CHAPTER 7
Effective Schema Design

Your application will live and die based on its physical implementation. Choose the wrong data structures, and performance will be crippled and flexibility limited. Choose the correct data structures, and you could have great performance. In this chapter, we will look at some of the things you need to consider when designing your schema. Then we will focus on table types, some useful index types, and finally, compression.

Fundamental Schema Design Principles

In this section, we'll consider some of the fundamental approaches that you should adopt when designing your schema. We'll cover integrity checking, datatype selection, and optimization for queries.

Let the Database Enforce Data Integrity

We touched on this subject way back in the "It's a Database, Not a Data Dump" section in Chapter 1. There, I relayed the story of the consultant who wanted to remove all referential integrity from the database and do it in the application. I explained that this was a really bad idea, for the following reasons:

- Yours will not be the last application to want to use this data. If the rules governing the data integrity are hidden away in client applications, they will not be enforced uniformly, and they will be very hard to change.
- It is slower to manage data integrity on the client. It requires additional round-trips to the server, and it must be done on the server as well.
- It takes orders of magnitude more code to manage integrity on the client.

I have a theory as to why developers are sometimes inclined to do this on the client: It has to do with the DBA-versus-developer philosophy instead of the DBA-and-developers-working-together approach. If the integrity of the data is managed in the database, the developers feel they have lost control. If the integrity of the data is managed in the application, the developers feel they have regained that control. This is a shortsighted perspective. Neither the developer nor the DBA own the data. The data is the property of someone else: the end users. It is certainly not in the best interests of the end users to obfuscate the business rules and hide them in the client application.

Why You Want to Keep Referential Integrity in the Database

Data-integrity rules must be placed into the database to ensure that they are consistently implemented and enforced. Application-specific rules—rules that apply only to the data in the context of the application's use of it—may be in the application.

Here are the reasons you want data integrity enforced by the database whenever possible:

- **Features such as query rewrite are thwarted.** These features are rendered less-than-useful if you do not tell the database about the relationships between objects, unless you declaratively spell out the rules regarding the data. In the section on QUERY_

REWRITE_ENABLED in Chapter 6, you saw an example of how a materialized view could not be used until the database was told about the relationship between the tables, the NOT NULL column, and the primary keys.

- **The data integrity will be compromised at some point in time**. Virtually every developed system that chooses to enforce foreign key constraints outside the database has orphaned child rows (child rows without parents). If you have such a system, just run this query: select foreign_key_columns from child_table MINUS select primary_key_columns from parent. You may be surprised to find some in your own database! Run a couple of checks on your data, and you might find NULL values where none should be and data that does not conform to your business rules (out-of-range data, for example). These arise from an inconsistent application of the business rules.
- **Server-enforced integrity is blindingly fast.** Is it slower than not using data integrity? Yes, of course it is. It adds an extra step, and it means that more code is executed in the server. Is it faster than you can code it yourself? Yes, it is. Is it applied consistently regardless of the application? Yes, it is.
- **The database provides more information.** By keeping the integrity rules in the database, your system is infinitely more self-documenting. A simple query tells you what relates to what and how.

As a last reason for using server-enforced integrity, let's consider what happens when you try to do it yourself.

The Problem with Do-It-Yourself Integrity Checks

Client-side enforcement of constraints is problematic at best, if you can do it at all. Here, we'll look at an example of the problems that can arise.

> "I hope someone can help me and, therefore, thank you in advance if you can. I am using Oracle9 and the EMP table. I need to ensure that no department is to have more than eight employees and fewer than three, except when a transaction reduces the number of employees to 0."

Interestingly, this user posed this question to many individuals in different forums, and the most frequent response was something along the lines of the following trigger:

```
SQL> create or replace trigger xxx
     after delete or update or insert on emp
     declare
     begin

      for irec in (select deptno, count(*) emps
                 from emp
                 group by deptno
                 having count(*) <3
                     or count(*) >8)
```

```
 11   loop
 12       RAISE_APPLICATION_ERROR(-20000, 'Department '
 13       ||irec.deptno || ' has '||irec.emps||' employees!');
 14   end loop;
 15  end;
 16  /
```

In various forms, this trigger was implemented. Various optimizations were suggested, such as only checking the DEPTNO values that were inserted/updated/deleted (because the performance of the preceding trigger would be questionable if EMP were large), but the answers were pretty much uniform: Use a trigger and count.

The problem is that the trigger doesn't work! My solution to this problem was a bit more involved, but it would work in all situations. He needs to serialize access to employee records at the DEPTNO level during insert, delete, and update operations to the DEPTNO column, and use database check constraints. Then use a trigger to maintain that value.

When you need to enforce a simple rule, such as a department in the EMP table must have three to eight employees or zero employees, a trigger-and-count solution may seem like it should work. The trigger could just look for any DEPTNO in EMP that does not have a count of employees between 3 and 8. And indeed, when we perform a single-user test, such a trigger does appear to work:

```
ops$tkyte@ORA920> select deptno, count(*),
  2         case when count(*) NOT between 3 and 8 then '<<<===='
  3              else null
  4          end
  5     from emp
  6    group by deptno
  7   /

    DEPTNO   COUNT(*) CASEWHE
---------- ---------- -------
        10          3
        20          5
        30          6
```

Our initial data is valid; all of the aggregate counts fall within our required range. If we attempt to delete three employees from Department 20, our trigger kicks in and saves us:

```
SQL> **delete from emp where empno in ( 7369, 7566, 7788 );**
delete from emp where empno in ( 7369, 7566, 7788 )
*
ERROR at line 1:
ORA-20000: Department 20 has 2 employees!
```

Now, however, we'll simulate two simultaneous sessions modifying the table. We need to use two SQLPlus sessions in order to do this. I opened the first SQLPlus session and entered:

```
ops$tkyte@ORA920> delete from emp where empno in ( 7369, 7566 );
2 rows deleted.
```

Now, in the second session, I executed:

```
ops$tkyte@ORA920> delete from emp where empno = 7788;
1 row deleted.
ops$tkyte@ORA920> commit;
Commit complete.
```

Now, we have a pending deletion of two employees (7369 and 7566) and a committed deletion of a single employee (7788). All three of these employees work in Department 20. When we commit that first transaction, there will be only two people left in Department 20, which is in violation of our business logic. *Remember, our trigger has already fired and will not fire again.* The data has apparently been validated, but, actually, when we commit that first transaction, our data will be corrupted, our data integrity shattered. Return to that first session and enter:

```
ops$tkyte@ORA920> commit;
Commit complete.

ops$tkyte@ORA920> select deptno, count(*),
  2         case when count(*) NOT between 3 and 8 then '<<<===='
  3              else null
  4          end
  5    from emp
  6   group by deptno
  7  /

    DEPTNO   COUNT(*) CASEWHE
---------- ---------- -------
        10          3
        20          2 <<<====
        30          6
```

The reason the trigger fails to enforce our business logic as we expect is because this solution does not account for the fact that Oracle provides nonblocking reads, consistent queries, and multiversioning. (For details on the Oracle multiversioning model, see the Oracle9i Release 2 *Concepts Guide*, Chapter 20, "Data Concurrency and Consistency.") Even though our first transaction was still in progress (while it was not committed), the second transaction was not blocked from reading rows that were involved in that transaction. Our first transaction deleted two employees, which left three remaining, so all is well. Our second transaction was not aware of the uncommitted changes made by the first transaction, so the trigger saw four rows left in DEPTNO 20, and that was fine. When we committed, the data integrity was lost.

Note that in many other databases, this logic would appear to work. The reason is that the query against EMP in databases that do not provide nonblocking reads (Microsoft SQL Server and DB2, for example) would have blocked on the changed rows. In our example, the result would have been a self-deadlock, because the second delete would have hung on the read of the EMP table. This is yet another reason why database independence is something that is truly hard to achieve generically: The algorithms must be implemented for the database you are using.

To solve the problem, we can use serialization and server-enforced integrity. There are a couple of ways to do this. Here, we'll use the DEPT table, where we'll maintain an aggregate column EMP_COUNT. It looks like this:

```
ops$tkyte@ORA920> alter table dept
  2  add emp_count number
  3  constraint must_be_between_3_8
  4  check(emp_count between 3 and 8 OR emp_count = 0)
  5  deferrable initially deferred;
Table altered.

ops$tkyte@ORA920> update dept
  2     set emp_count = (select count(*)
  3                        from emp
  4                       where emp.deptno = dept.deptno )
  5  /
4 rows updated.

ops$tkyte@ORA920> alter table dept
  2  modify emp_count NOT NULL;
Table altered.
```

Now, we have an aggregate column in the DEPT table that we will use to maintain a count of employees. Additionally, this column has a declarative check constraint on it that verifies the count of employees is either 0 or between 3 and 8, as specified in the rule. Lastly, we made this constraint `deferrable initially deferred`, meaning it will not be validated until we commit by default. The reason for doing that is to permit multistatement transactions to execute without error. For example, we could update a row setting the DEPTNO from 20 to 30, reducing the number of employees in DEPTNO 20 to 2 but immediately follow that with an update of another row from DEPTNO 30 to 20, increasing the number of employees back to 3. If the constraint were validated after each and every statement, this sort of logic would fail.

Now, we need a trigger to maintain that value:

```
ops$tkyte@ORA920> create trigger emp_dept_cnt_trigger
  2  after insert or update or delete on emp
  3  for each row
  4  begin
  5      if ( updating and :old.deptno = :new.deptno )
  6      then
  7          return; -- no change
  8      end if;
  9      if ( inserting or updating )
 10      then
 11          update dept set emp_count = emp_count+1
 12           where deptno = :new.deptno;
 13      end if;
 14      if ( updating or deleting )
 15      then
```

```
        update dept set emp_count = emp_count-1
         where deptno = :old.deptno;
    end if;
end;
/
Trigger created.
```

NOTE
For ease of example, the SQL for the trigger is presented in-line. It would be advantageous to place the SQL into a stored procedure, especially in this case. Not only would we benefit from reducing the soft parsing our application performs, but we would also be able to get away with a single UPDATE statement.

If we are updating and did not change the DEPTNO column, we just return; no changes need to be made. Otherwise, we update the DEPT table and increment or decrement the EMP_COUNT column as needed, since we are using foreign keys. We are assured that the UPDATE statements update at least and at most a single row in DEPT. (You cannot possibly have a child row in EMP with a DEPTNO that does not exist in DEPT, because the database is enforcing that for us.)

The update of the *single row* in DEPT also causes *serialization* at the DEPTNO level; that is, only one transaction can insert into DEPTNO=20 at any point in time. All other sessions trying to insert, delete, or transfer where DEPTNO=20 will block on that update. All of the other DEPTNO operations are available for modification; just not this one. We can still update other EMP rows in DEPTNO=20, as long as we do not change their DEPTNO. This serialization is what allows our constraint to be effective.

As another example, consider a primary key/foreign key relationship. The code you would write in order to use client-side enforcement of constraints would need to do the following:

Action	**Reaction**
Insert into a child table or update a child table's foreign key value.	Lock the parent row for that foreign key (most people would simply select without locking). Note that this will serialize at the parent table row level.
Update the parent primary key or delete from parent table.	Lock the entire child table. Then look for child rows before you update or delete that parent row. That is your only recourse. You must lock the entire child table to prevent any rows that reference this primary key from being created. Oracle itself can bypass this child table lock and do it more efficiently, but only because it does it internally.

Neither of these approaches is very good for concurrency. Oracle can do much better on its own. You will notice that many sessions can insert into a child table simultaneously if you use declarative integrity, and many sessions can modify the parent table using declarative integrity. If you do it yourself, *and you do it correctly*, you will be forced to serialize frequently, greatly decreasing concurrency and, hence, scalability in your database.

Don't believe anyone when they say, "Don't use integrity in the database. It is too slow." They either don't enforce data integrity at all or they have been doing it wrong (and, hence, have a false sense of security). If they did it right, their system is running many times slower than it should be! It is all about benchmarking and doing it right.

Does that Mean that the Client Should Not Do Integrity Checking Itself?

It can be very useful for the client to do integrity checking for a variety of reasons. The important thing to remember, however, is that it must ultimately *still be performed in the database*. Client-side integrity is not a substitute for server-side integrity; it is a complementary addition. Here are some of the salient reasons for client-side integrity:

- **Better end-user experience** Users can discover as they are typing that the data they entered doesn't stand a chance of being entered into the database. They do not need to wait until they select Save.
- **Reduced resource usage on the server** By preempting bad data on the client, you do not make the server perform work that will ultimately need to be undone.

But, just as there are pros to most everything, there are cons as well. The major disadvantage with client-side integrity is that you now have two places where the rules are, and if they change over time, you must ensure they are changed in both locations. It would be frustrating for a client to not be able to input information the database should accept because the application is working from old rules. It would be equally as frustrating to work on a set of data and feel confident that it is valid, only to discover as you save the data from the application that the database will reject it. Good configuration management and software engineering principles will reduce the chances of having this happen.

Use the Correct Datatype

Using the correct datatype seems like common sense, but virtually every system I look at does one of the following:

- Uses a string to store dates or times
- Uses a string to store numbers
- Uses VARCHAR2(4000) to store all strings.
- Uses CHAR(2000) to store all strings, wasting tons of space and forcing the use of a lot of `trim` function calls
- Puts text in a BLOB (raw) type

I have a very simple rule: Put dates in dates, numbers in numbers, and strings in strings. Never use a datatype to store something other than what it was designed for, and use the most specific type possible. Furthermore, only compare dates to dates, strings to strings, and numbers to numbers. When dates and numbers are stored in strings, or stored using inappropriate lengths, your system suffers:

- You lose the edit upon insertion to the database, verifying that your dates are actual dates and numbers are valid numbers.
- You lose performance.
- You potentially increase storage needs.
- You definitely decrease data integrity.

How many of you know what ORA-01722 or ORA-01858 errors are off the top of your head? I bet many of you do, because they are so prevalent in systems where numbers are stored in strings (ORA-01722: invalid number) and dates in strings (ORA-01858: a non-numeric character was found where a numeric was expected).

How Data Integrity Decreases

Using an incorrect datatype is wrong for many reasons, but the first and foremost is *data integrity*. Systems that use strings for dates or numbers will have some records with dates that are not valid and numbers that are not numbers. It is just the nature of the game here. If you permit any string in your date field, at some point you will get dirty data in there.

Without data-integrity rules in place, the integrity of your data is questionable. I've needed to write the functions to convert strings to dates but return NULL when the date won't convert. I've also needed to try one of five date formats to see if I can get the date to convert. Can you look at 01/02/03 and tell what date that is? Is that *yy/mm/dd*, *dd/mm/yy*, or something else?

How Performance Suffers

Beyond the obvious data-integrity issues associated with incorrect datatypes, there are other subtle issues. To demonstrate, we'll use an example of a table with two date columns. One will be stored in a string using YYYYMMDD and the other as a DATE type. We will index these values and analyze the tables completely.

```
ops$tkyte@ORA920> create table t
  2  as
  3  select to_char(to_date('01-jan-1995','dd-mon-yyyy')+rownum,'yyyymmdd') str_date,
  4         to_date('01-jan-1995','dd-mon-yyyy')+rownum date_date
  5    from all_objects
  6  /
Table created.

ops$tkyte@ORA920> create index t_str_date_idx on t(str_date);
Index created.

ops$tkyte@ORA920> create index t_date_date_idx on t(date_date);
Index created.

ops$tkyte@ORA920> analyze table t compute statistics
```

```
  2  for table
  3  for all indexes
  4  for all indexed columns;
Table analyzed.
```

Now, let's see what happens when we query this table using the string date column and the real date column. Pay close attention to the Cost and Card= component of the plan:

```
ops$tkyte@ORA920> set autotrace on explain
ops$tkyte@ORA920> select * from t
  2 where str_date between '20001231' and '20010101';

STR_DATE DATE_DATE
-------- ---------
20001231 31-DEC-00
20010101 01-JAN-01


Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=406 Bytes=6090)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T' (Cost=5 Card=406 Bytes=6090)
   2    1     INDEX (RANGE SCAN) OF 'T_STR_DATE_IDX' (NON-UNIQUE)
              (Cost=3 Card=406)

ops$tkyte@ORA920> select * from t where date_date between
to_date('20001231','yyyymmdd') and to_date('20010101','yyyymmdd');

STR_DATE DATE_DATE
-------- ---------
20001231 31-DEC-00
20010101 01-JAN-01


Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=1 Bytes=15)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T' (Cost=3 Card=1 Bytes=15)
   2    1     INDEX (RANGE SCAN) OF 'T_DATE_DATE_IDX' (NON-UNIQUE)
              (Cost=2 Card=1)
```

So, what happened there? Well, the optimizer understands VARCHAR2 types and it understands DATE types. The optimizer knows that between the two DATE items, December 31, 2000, and January 1, 2001, there is only one day. The optimizer also *thinks* that between the two string items, '20001231' and '20010101', there are a whole bunch of values. The cardinality is thrown off.

But, so what? What do we care if the cardinality is wrong? It won't affect our output—the answer. That is correct, but it could have some impact on our overall performance. Consider a different query against the same data, asking for effectively the same result set:

```
ops$tkyte@ORA920> select * from t
  2 where str_date between '20001231' and '20060101';

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=12 Card=2034 Bytes=30510)
   1    0   TABLE ACCESS (FULL) OF 'T' (Cost=12 Card=2034 Bytes=30510)

ops$tkyte@ORA920> select * from t where date_date between
to_date('20001231','yyyymmdd') and to_date('20060101','yyyymmdd');

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=12 Card=1823 Bytes=27345)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'T' (Cost=12 Card=1823 Bytes=27345)
   2    1     INDEX (RANGE SCAN) OF 'T_DATE_DATE_IDX'
              (NON-UNIQUE) (Cost=6 Card=1823)
```

NOTE
As discussed in Chapter 6, different database parameter settings will influence the cost of various operations. You may need to increase the range of dates to see the same effect as shown in this example, but it will happen at some threshold.

This time, the fact that we hid a date in a string has a serious side effect. Our query plan has changed. We are now full-scanning for the string date but index-range scanning for the DATE type date. So, besides the fact that there is nothing stopping someone from inserting 20009950 as a date value into our field, the use of a string has withheld valuable information from the database. We lose all around.

How You Might Increase Your Storage Requirements

In addition to using the proper base datatype such as number, date, or string, you should also use the most specific type you can. For example, use VARCHAR2(30) for a field that is up to 30 characters in length; do *not* use VARCHAR2(4000).

"I work with a modelers group. My modeler would like to define every VARCHAR2 field with the maximum length, which means that a table with 20 VARCHAR2 fields will all be defined with a maximum of 2000 or 4000 bytes. I tried to talk to him about the reasons we identify data with correct lengths and names in order to understand what we have in our database. He told me that it doesn't matter, since Oracle just stores the length, etc., and there is no overhead. I don't believe this is true, but have been jumping between so many databases that I cannot find a document on the internals of Oracle storage. Can you help me out here with this question?"

My gut response was, "This is your data modeler, my goodness!" They are the ones who are supposed to be telling you that it is vital to use the appropriate length when defining fields! That is their *job*. Let's just forget about things like storage for a minute, why don't we ask him:

- What is going to happen when users pull this up in a query tool that formats each field based on the width of the column in the database? They'll see one column and need to scroll way over to see the second, the third, and so on.
- Say the code prepares a query that selects ten columns that are VARCHAR2. The developers, for performance, would like to array fetch (very important). They would like to array fetch say 100 rows (very typical). So, you have 4,000 × 10 × 100 = almost 4MB of RAM the developers must allocate! Now, consider if that were ten VARCHAR2(80) fields (it's probably much smaller than that). That's about 78KB. Ask the data modeler how much RAM he is willing to kick in for this system.
- Now, the developers start to build a data-entry screen to put data into the database. Wow, that code field can be 4,000 characters long and that first name can be 4,000 characters long. How the heck is anyone going to know what sort of data can really go in there?

Tell your data modeler group members that they need to consider the length as a constraint. Just as they use primary and foreign keys, they should use the proper and correct length on fields. You can always expand a field via a command like alter table t modify c varchar2(bigger_number). There is no reason to use the maximum length everywhere. It will hurt the applications you develop, because they will mistakenly allocate many megabytes of RAM. Just think of the array fetch example with an application server. Now, it's not just 4MB; it's 4MB × number of connections. You are talking some real memory here for a *single query,* and you'll be doing a lot of them at the same time.

A CHAR(2000) will consume 2,000 bytes of storage whether you put in the letter *a*, the string 'hello world', or 2,000 characters. A CHAR is always blank-padded. Additionally, are you thinking about using an occasional index in your system? If so, beware of storage-related problems. Consider the following:

```
tkyte@ORA817.US.ORACLE.COM> create table t ( a varchar2(4000), b varchar2(4000));
Table created.

tkyte@ORA817.US.ORACLE.COM> create index t_idx on t(a);
create index t_idx on t(a)
                       *
ERROR at line 1:
ORA-01450: maximum key length (3218) exceeded
```

NOTE
In Oracle9i, the maximum key length is larger, but the restriction still exists. For example, an index on T(a,b) would raise: ORA-01450: maximum key length (6398) exceeded in Oracle9i with an 8KB blocksize.

My system has an 8KB blocksize. I would need to use at least a 16KB blocksize to index a *single* column, but even then, if I tried to create a concatenated index on T(A,B), it would fail there!

The same holds true for your numbers and the new Oracle9i TIMESTAMP datatypes: When appropriate, use scales and precisions on those fields in order to better define your data integrity and to give applications that much more information about the data itself.

In short, never be tempted to use anything other than a DATE or TIMESTAMP datatype to hold a date value, and never be tempted to use a VARCHAR2 to hold a number. Use the appropriate and correct type for each to ensure maximum performance, and more important, to protect your data integrity.

Optimize to Your Most Frequently Asked Questions

If you have a system you know will be executing some given query or set of queries hundreds of times per minute, or even per second, that system must be designed, optimized, and built around those queries.

For example, my workplace once had an internal system called Phone. You could telnet into any email machine (back when email was character mode), and on the command line, type phone <*search string*>. It would return data like this:

```
$ phone tkyte
TKYTE    Kyte, Tom     703/555 4567  Managing Technologies RESTON:
```

When the Web exploded in about 1995/1996, our group wrote a small web system that loaded this phone data into a table and let people search it. Now that it was in a database and had a little GUI to go with it, it started becoming the de-facto standard within the company for looking up information about people. Over time, we started adding more data and more fields to it. It really started to catch on.

At some point, we decided to add a lot more fields to the system and rebuild it with more features. The first thing we did, based on our knowledge of how people would use this simple little system, was to design the tables to hold this data. We had a read-only repository of data that people would be searching, and it needed to be fast. This system was growing in popularity every day and was threatening to consume our machine with its resources. We had a single 67-column, 75,000-row table upon which we wanted to perform a simple string search against various fields. So, if a user put in *ABC*, it would find *ABC* in the email address, or the first name, or the last name, or the middle name, and so on. Even worse, it would be interpreted as %ABC%, and the data was in mixed case.

There wasn't an index in the world that could help us here (we tried them all), so we built our own. Every night, as we refreshed the data from our human resources system (a complete refresh of data), we would also issue the following after we were finished:

```
CREATE TABLE FAST_EMPS
PCTFREE 0
CACHE
AS
SELECT upper(last_name)||'/'||upper(first_name)||'/' …. || '/' ||
                  substr( phone, length(phone)-4) SEARCH_STRING,
       rowid row_id
  FROM EMPLOYEES
/
```

In effect, we built the most dense, compact table possible (PCTFREE 0) and asked that it be cached, if possible. Then we would query:

```
select *
  from employees
 where rowid in ( select row_id
                    from fast_emp
                   where search_string like :bv
                     and rownum <= 500 )
```

This query would always full-scan the FAST_EMP table, which is what we wanted, because that table was, in fact, our "index." Given the types of questions we were using, that was the only choice. Our goal from the outset was to minimize the amount of data that would be scanned, to limit the amount of data people would get back, and to make it as fast as possible. The query shown here accomplishes all three goals.

The FAST_EMP table is typically always in the buffer cache. It is small (less than 8% the size of the original table) and scans very fast. It has already done the work of the case-insensitive searching for us once (instead of once per query) by storing the data in uppercase. It limits the number of hits to 500 (if your search is broader than that, refine it; you'll *never* look at 500 hits). In effect, that table works a lot like an index, since it stores the ROWID in EMPLOYEES. There are no indexes employed on this system in order to do this search—just two tables.

Before settling on this design, we tried a couple of alternative approaches:

- We tried using a fast full scan on a function-based index (close, but not as fast).
- We tried interMedia text (not useful due to the %ABC% requirement).
- We tried having just an extra field in the base EMPLOYEES table (wiped out the buffer cache because it was too big to full-scan).

It may seem foolish to have spent so much time on this detail. However, this one query is executed between 150,000 and 250,000 times a day. This is two to three times a second, every second, all day long, assuming a constant flow of traffic (and we cannot assume that, since we frequently have spikes in activity, as most systems do). If this single query performed poorly, our entire system would fall apart, and it is just one of thousands of queries we need to do. By determining where our weak points would be—the lowest hanging fruit, so to say—and concentrating on them, we were able to build an application that scales very well. If we had tried the tuning-after-the-fact principle, we would have found ourselves rewriting after the fact.

The point is that when developing a system, you need to give a lot of thought to how people will actually use that system. Consider how physical organization can matter, and realize the impact a simple 100% increase in logical I/O might have on your system's performance.

Overview of Table Types

Steve Adams (http://www.ixora.com.au/), a very bright guy who knows a lot about Oracle, has been known to say, "If a schema has no IOTs or clusters, that is a good indication that no thought has been given to the matter of optimizing data access." His point is that most systems use only two data structures in the Oracle database: heap-based tables (the kind you get from a simple `create table t ( x int)`) and B*Tree indexes (the kind you get from a simple `create index t_idx on t(x)`).

Oracle provides a wealth of other types of data structures you can use to store and access your data. In the following sections, we'll take a look at each of the different types, starting with tables and then moving onto indexes.

Let's define each type of table that we're going to consider before getting into the details.

- **Heap-organized table** This is a standard database table. Data is managed in a heap-like fashion. As data is added, the first free space found in the segment that can fit the data will be used. As data is removed from the table, it allows space to become available for reuse by subsequent insert and update operations. This is the origin of the name *heap* as it refers to tables like this. A *heap* is a bunch of space, and that space is used in a somewhat random fashion.
- **B*Tree Index Clustered table** Two things are achieved with this type of table. First, many tables may be stored physically joined together. Normally, one would expect data from only one table to be found on a database block. With clustered tables, data from many tables may be stored together on the same block. Second, all data that contains the same cluster key value will be physically stored together. The data is clustered around the cluster key value. A cluster key is built using a B*Tree index.
- **Hash-clustered table** This type is similar to a clustered table, but instead of using a B*Tree index to locate the data by cluster key, the hash cluster hashes the key to the cluster to determine the database block on which to store the data. In a hash cluster, the data is the index (metaphorically speaking). This would be appropriate for data that is read frequently via an equality comparison on the key. There are two types of hash-clustered tables: single-table and multiple-table hash clusters. We'll focus on single-table hash clusters here, because the clustered tables examples will show the use of multiple-table clusters.
- **Index-organized table (IOT)** Here, a table is stored in an index structure. This imposes physical order on the rows themselves. Unlike in a heap, where the data is stuffed wherever it might fit, in an IOT, the data is stored in sorted order according to the primary key.
- **External table** This is a new type with Oracle9i Release 1. As the name suggests, external tables give us the ability to store data in flat files outside the Oracle database. You cannot modify data in these table types; you can only query them. Also, they cannot be conventionally indexed.

For details about the basic Oracle table types, see the Oracle9i Release 2 *Concepts Guide*, Chapter 10, which has a "Tables" section.

We're not going to consider the heap-organized tables directly, as they are the standard table type that most people use all the time. However, we will use them in our benchmarks, to show the performance advantages that the other types can sometimes bring. Let's get started by considering clustered tables.

B*Tree Index Clustered Tables

Clusters are segments in Oracle that do two things for us:

- They physically colocate data together by a common key. The data is not sorted; it's just physically stored together by key. If you have a cluster built on a number key, all of the data with the key value of 10 would be physically colocated together, optimally on the same database block, if it all fits.
- They allow us to store data from multiple database tables in the same physical database block. For example, the row for DEPTNO=10 from the DEPT table may be on the same block as the rows from the EMP table for that same DEPTNO. Consider this a method of prejoining data.

Immediately, we can see some potential advantages of using clusters. The physical colocation of data means that we can store rows from many tables on the same database block. The fact that all tables share one cluster key index means that we have a decreased need for indexes. (In fact, hash clusters remove the need for indexes completely, because the data is the index.)

In a nutshell, clusters allow us to limit the amount of I/O the system needs to perform in order to answer our queries. If we clump together data that is used together, the database will need to perform less-physical I/O and perhaps even less-logical I/O to answer our questions. However, you should be aware of two basic limitations of clusters:

- You cannot do direct-path loading into a cluster.
- You cannot partition clustered tables.

So, if you know you need to do either of the preceding with a table, a cluster is not the right choice.

NOTE
I would argue that direct-path loading is something you seriously reconsider. Many times, people feel that using direct-path loading will speed up loading, when, in fact, the conventional-path load process would satisfy their needs. Then they would find that the advantages of using one of Oracle's more sophisticated storage devices would pay off thousands of times during the day as people access the data!

You may not realize it, but you use clusters every day in Oracle, because the main tables in the data dictionary are stored in clusters (B*Tree clusters). To see them, use this query:

```
ops$tkyte@ORA920> select cluster_name, owner, table_name
  2    from dba_tables
  3   where cluster_name is not null
  4   order by cluster_name
  5  /
```

```
CLUSTER_NAME                   OWNER TABLE_NAME
------------------------------ ----- ------------------------------
C_COBJ#                        SYS   CDEF$
                               SYS   CCOL$

C_FILE#_BLOCK#                 SYS   SEG$
                               SYS   UET$

C_MLOG#                        SYS   MLOG$
                               SYS   SLOG$

C_OBJ#                         SYS   CLU$
                               SYS   COL$
…
                               SYS   LIBRARY$
                               SYS   NTAB$
…
```

The C_OBJ# cluster itself has 16 tables in it. That means there could be rows from up to 16 tables stored on a single database block, all prejoined by their common key. When Oracle needs to find information about an object (a table, for example), it may need to do a single physical I/O to read in all of the information from 16 different tables!

In the remainder of this section, we'll walk through a cluster implementation. We'll discuss relevant considerations for the effective use of clusters, and we'll benchmark their performance against a more traditional implementation using conventional heap tables. As you will see as you read through this section, there are many situations where a cluster should be considered.

Create Clusters

Oracle supports B*Tree clusters and hash clusters. The difference between these two types lies in *how* the data is accessed:

- The B*Tree cluster uses a conventional B*Tree index to store a key value and block address where the data can be found. So, much like using an index on a table, Oracle would look up the key value in the cluster index, find the database block address the data is on, and go there to find the data.
- The hash cluster uses a hashing algorithm to convert the key value into a database block address, thus bypassing all I/O except for the block read itself. With a hash cluster, the data is the index itself. Effectively, this means that there will optimally be one logical I/O used to perform a lookup.

We'll start with how to create a B*Tree cluster, and then discuss how to create a hash cluster.

Create a B*Tree Cluster

Creating a B*Tree cluster object is fairly straightforward:

```
ops$tkyte@ORA920> create cluster user_objects_cluster_btree
  2  ( username varchar2(30) )
  3  size 1024
  4  /
Cluster created.
```

Normally, you expect a table to be the physical storage object (segment), but in this case, the cluster itself is the storage object. The cluster is the object that can have storage parameters; the tables created in a cluster cannot have storage parameters. The cluster is the object that specifies other parameters such as INITRANS, PCTFREE, PCTUSED, and so on.

The sample cluster we created will be "keyed" off a VARCHAR2(30) column called USERNAME. This implies that all the data in any table added to this cluster will be physically organized by some VARCHAR2(30) column, and that column will be a username or object owner. There is nothing to stop us from using *any* VARCHAR2(30) column as the key, but it would be bad practice to create a cluster on a key named USERNAME, and then later add tables to the cluster using something other than a username as the key.

The other interesting thing to note here is the SIZE parameter. This is used to tell Oracle that we expect about 1,024 bytes of data to be associated with each cluster key value. Oracle will use that to compute the maximum number of cluster keys that could fit per block. Given that I have an 8KB blocksize, Oracle will fit up to seven cluster keys per database block; that is, the data for up to seven distinct USERNAME key values would tend to go onto one block. As soon as I insert the eighth username, a new block will be used. That does not mean that the data is stored sorted by username here. It just means that all data relating to any given username will typically be found on the same block. As we'll see a little later, both the size of the data and the order in which the data is inserted will affect the number of keys we can store per block.

Next, we'll create our cluster key index:

```
ops$tkyte@ORA920> create index user_objects_idx
  2  on cluster user_objects_cluster_btree
  3  /
Index created.
```

This creates a B*Tree index that will contain an entry for each unique key value we put into the cluster and a pointer to the first block containing data for that key value. This index is not optional; it must be created in order to actually store data in the cluster. That is because the entire concept of the B*Tree cluster is predicated on storing like data together. In order to know where to put and where to get the rows in the cluster, Oracle needs this index. It tells Oracle that the data for KEY=*X* is on block *Y*.

The cluster index's job is to take a cluster key value and return the block address of the block that contains that key. It is a primary key, in effect, where each cluster key value points to a single block in the cluster itself. So, when you ask for the data for username SCOTT, Oracle will read the cluster key index, determine the block address for that, and then read the data.

Create Hash Clusters

To create a hash cluster instead of a B*Tree cluster, add a HASHKEYS parameter and skip the creation of the cluster key index. Here is the example from the previous section created as a hash cluster:

```
ops$tkyte@ORA920> create cluster user_objects_cluster_hash
  2  ( username varchar2(30) )
```

```
     hashkeys 100
     size 3168
     /
Cluster created.
```

In this CREATE CLUSTER statement, you see two key elements: HASHKEYS and SIZE.

HASHKEYS specifies the number of unique key values we expect over time. Here, we set it to 100. It does not limit us to only 100 distinct usernames; that is just the size of the hash table Oracle will set up. If we exceed that value, we will necessarily begin to get "collisions" in our hash table. This will not prevent us from working, but it will decrease the efficiency of our hash table.

SIZE has the same meaning as it did in our B*Tree cluster example. Here, since a hash cluster preallocates enough space to hold HASHKEYS/TRUNC(BLOCKSIZE/SIZE) bytes of data, we want to be more careful about the sizing. So, for example, if you set your SIZE to 1,500 bytes and you have a 4KB block size, Oracle will expect to store 2 keys per block. If you plan on having 1,000 hash keys, Oracle will allocate 500 blocks. In this case, since I have an 8KB blocksize, I more or less directed Oracle to allocate 50 blocks of storage (two keys per block).

Create Tables in a Cluster

Let's move on and create two tables in our B*Tree cluster (the process is the same for the hash cluster). We will use a parent/child table built from the DBA_USERS/DBA_OBJECTS tables:

```
ops$tkyte@ORA920> create table user_info
      ( username, user_id, account_status,
        lock_date, expiry_date, default_tablespace,
        temporary_tablespace, created, profile )
      cluster user_objects_cluster_btree(username)
      as
      select username, user_id, account_status,
             lock_date, expiry_date,
             default_tablespace, temporary_tablespace,
             created, profile
        from dba_users
       where 1=0
      /
Table created.

ops$tkyte@ORA920> create table users_objects
      ( owner, object_name, object_id, object_type,
        created, last_ddl_time, timestamp, status )
      cluster user_objects_cluster_btree(owner)
      as
      select owner, object_name, object_id, object_type, created,
             last_ddl_time, timestamp, status
        from dba_objects
       where 1=0
      /
Table created.
```

So, here we've created two empty tables. The first, USER_INFO, will use its column named USERNAME to organize itself. The table USERS_OBJECTS will use its column OWNER. That means all of the data in the USER_INFO table with a USERNAME value of *X* will physically go in the database in about the same location as all of the data in the USERS_OBJECTS table when OWNER = *X*.

We could create the equivalent heap tables using the same CREATE TABLE AS SELECT statements without the CLUSTER clause. We are now ready to load data into our cluster tables.

Use Clusters

Loading data into tables illustrates one of the key considerations in gauging the potential effectiveness of clusters in your application. Some of the other useful aspects of clusters we'll look at include reducing I/O, increasing buffer cache efficiency, eliminating index blocks, and creating read-only lookup tables.

Control How the Data Is Loaded

We will use a multitable insert to populate both objects at the same time:

```
ops$tkyte@ORA920> insert
when (r=1) then
into user_info
( username, user_id, account_status, lock_date,
  expiry_date, default_tablespace, temporary_tablespace,
  created, profile )
values
( username, user_id, account_status, lock_date,
  expiry_date, default_tablespace, temporary_tablespace,
  user_created, profile )
when (1=1) then
into users_objects
( owner, object_name, object_id, object_type, created,
  last_ddl_time, timestamp, status )
values
( owner, object_name, object_id, object_type, obj_created, last_ddl_time,
  timestamp, status )
select a.username, a.user_id, a.account_status, a.lock_date,
       a.expiry_date, a.default_tablespace,
       a.temporary_tablespace, a.created user_created, a.profile,
       b.owner, b.object_name, b.object_id, b.object_type,
       b.created obj_created,
       b.last_ddl_time, b.timestamp, b.status,
       row_number() over (partition by owner order by object_id) r
  from dba_users a, dba_objects b
 where a.username = b.owner
   and a.username <> 'SYS'
/

4749 rows created.

ops$tkyte@ORA920> analyze cluster user_objects_cluster_btree compute statistics;
```

```
Cluster analyzed.

ops$tkyte@ORA920> analyze table user_info compute statistics
  2 for table for all indexes for all indexed columns;
Table analyzed.

ops$tkyte@ORA920> analyze table users_objects compute statistics
  2 for table for all indexes for all indexed columns;
Table analyzed.
```

You're probably wondering why we did the insert in this complicated fashion—with a join with ROW_NUMBER and such—rather than simply using two INSERT statements. While the latter option is indeed simpler, it may not achieve the goal of clustering data by key. Remember that when we created the B*Tree cluster we used this code:

```
ops$tkyte@ORA920> create cluster user_objects_cluster_btree
2  ( username varchar2(30) )
3  size 1024
4  /
Cluster created.
```

That told Oracle, "There will be about 1KB of data associated with each key value, so go ahead and store up to seven username keys per block, or fewer if you need to." If we had inserted the data from DBA_USERS first into this table, there would be one row per user, with a small amount of data. We would easily get seven username keys per block. However, on my system, the DBA_OBJECTS data varied widely in size from 0.5KB to 139KB of data per key. So, when we inserted that data, the required clustering of data would not have been achieved. There is not enough room for seven keys per block every time. For some USERNAME values, there will be one key per block.

Our goal is to store this information together, and in order to do that, we need to load the data more or less together. This is an important thing to understand about clusters and why they work for something like the Oracle data dictionary so well. Consider how the data dictionary gets populated. You execute DDL such as:

```
create table t
( x int,
  y date,
  z varchar2,
  constraint check_cons check ( x > 0 ),
  constraint t_pk primary key(x,y) );

create index t_z_idx on t(z);
```

In the real data dictionary, there is a CLUSTER C_OBJ# object that contains tables such as CLU$ (cluster information), COL$ (column information), TAB$ (table information), IND$ (index information), ICOL$ (index column information), and so on. When you create a table and its related information, Oracle populates all of those objects *at about the same time*. In general, the amount of information (number of rows) does not go up or down very much, since you do not drop or add columns very often, the number of constraints is generally fixed, and so on. In this case, all of the information related to an object goes in once—on object creation—and tends to all go into the same block(s) together.

By loading all of the data for a given cluster key at the same time, we pack the blocks as tightly as possible and start a new block when we run out of room. Instead of Oracle putting up to seven cluster key values per block, it will put as many as will fit.

To see this in action, let's first look at the degree of colocation we achieved by loading the data at the same time. We'll do this by looking at the ROWIDs of the data and breaking down the rows by file and block. We'll quickly be able to see how the data is organized.

```
ops$tkyte@ORA920> break on owner skip 1
ops$tkyte@ORA920> select owner, arfno||'.'||ablock arowid,
                         brfno||'.'||bblock browid, count(*)
       from (
     select b.owner,
            dbms_rowid.rowid_relative_fno(a.rowid) arfno,
            dbms_rowid.rowid_block_number(a.rowid) ablock,
            dbms_rowid.rowid_relative_fno(b.rowid) brfno,
            dbms_rowid.rowid_block_number(b.rowid) bblock
       from user_info a, users_objects b
      where a.username = b.owner
            )
      group by owner, arfno, ablock, brfno, bblock
      order by owner, arfno, ablock, bblock
     /

OWNER        AROWID          BROWID            COUNT(*)
------------ --------------- --------------- ----------
A            9.271           9.271                    1
```

This shows that for the OWNER = A, the row in USER_INFO is in file 9, block 271. Additionally, the row for USERNAME = A, the row in USERS_OBJECTS, is also in file 9, block 271. The individual rows from each table are physically colocated on the same database block. Continuing on:

```
AQ           9.271           9.271                   10

BIG_TABLE    9.271           9.271                   36

CSMIG        9.271           9.271                   14

CTXSYS       9.271           9.268                  101
             9.271           9.269                   34
             9.271           9.271                   31
             9.271           9.272                   97
```

This shows that for the OWNER = AQ data, the single row from USER_INFO is colocated on the same database block with ten rows from USERS_OBJECTS. The data is, in effect, prejoined.

We see the same thing with BIG_TABLE: The single row in USER_INFO is colocated with 36 rows from USERS_OBJECTS. This is also true for CSMIG. CTXSYS, however, is a bit different. CTXSYS has about 260 objects, and this many objects simply will not fit on a single block.

However, notice that the rows are still nicely clustered together, packed rather tightly on four blocks by the cluster key. So, even though they do not fit on a single block, the database took care to put them into as few blocks as possible.

Suppose we had instead loaded these tables in this less-thought-out fashion:

```
ops$tkyte@ORA920> insert
  2  into user_info
  3  ( username, user_id, account_status, lock_date,
  4    expiry_date, default_tablespace, temporary_tablespace,
  5    created, profile )
  6  select a.username, a.user_id, a.account_status, a.lock_date,
  7         a.expiry_date, a.default_tablespace,
           a.temporary_tablespace,
  8         a.created user_created, a.profile
  9    from dba_users a
 10   where a.username <> 'SYS'
 11  /
44 rows created.

ops$tkyte@ORA920> insert
  2  into users_objects
  3  ( owner, object_name, object_id, object_type, created,
  4    last_ddl_time, timestamp, status )
  5  select b.owner, b.object_name, b.object_id, b.object_type,
           b.created obj_created,
  6         b.last_ddl_time, b.timestamp, b.status
  7    from dba_users a, dba_objects b
  8   where a.username = b.owner
  9     and a.username <> 'SYS'
 10   order by object_type, object_name
 11  /
4717 rows created.
```

Arguably, we have the same data here, but the physical organization is quite different. Here, Oracle first received the USER_INFO data. This was quite small, so Oracle put seven key values per block on average. Then it received the USERS_OBJECTS data sorted in a fashion that ensured that the OWNER column was not coming in sorted order; this data was scrambled. Now, when we look at a sample of how colocated the data is, we find this:

```
OWNER        AROWID          BROWID            COUNT(*)
------------ --------------- --------------- ----------
A            9.269           9.377                    1
```

Already, we can see that the location of the data is not as orderly as before. Oracle inserted the single row for OWNER = A into block 269. Later on, when the USERS_OBJECTS row that corresponds to that entry came to be inserted, Oracle discovered that block was full; hence, it could not place the data from USERS_OBJECTS on that same block. The data from these two rows is now spread across two different blocks.

Continuing on, we can see the full extent of the damage:

```
AQ             9.268          9.268                   8
               9.268          9.388                   1
               9.268          9.532                   1
```

The data for AQ is spread out on three discrete blocks now, instead of all ten rows on one block as before.

```
BIG_TABLE      9.272          9.272                   8
               9.272          9.347                   5
               9.272          9.372                   1
               9.272          9.379                  14
               9.272          9.381                   7
               9.272          9.536                   1
```

The data for BIG_TABLE is spread out over six blocks, as opposed to one.

```
CSMIG          9.269          9.384                   8
               9.269          9.533                   6

CTXSYS         9.269          9.269                  53
               9.269          9.362                   2
               9.269          9.363                   2
               9.269          9.371                  21
               9.269          9.372                  54
               9.269          9.373                   4
               9.269          9.374                  23
               9.269          9.381                  38
               9.269          9.386                   4
               9.269          9.389                   6
               9.269          9.390                   7
               9.269          9.391                   2
               9.269          9.534                  47
```

And CTXSYS, which used to be nicely packed on 4 blocks, is now on 13!

What you should conclude from this example is that clusters will be most effective in environments where we can control how the data is loaded. For example, clusters are appropriate for warehousing-type operations where you reload from time to time and hence can control how the data is loaded to achieve maximum clustering. B*Tree clusters are also appropriate for update (transactional) systems if you tend to insert the parent and many (or most or all) of the child records at about the same time, similar to what happens when Oracle processes a CREATE TABLE statement. Most of the parent and child rows in the data dictionary are inserted at about the same time. B*Tree clusters are not as appropriate if the data arrives randomly, out of cluster key order, most of the time.

B*Tree Clusters Can Reduce I/O and Increase Buffer Cache Efficiency

By buffer cache efficiency, I do not necessarily mean a cache hit ratio. Rather, by storing related information together, on the same blocks, we can increase the efficiency of our buffer cache. Instead of needing to manage 50 blocks in response to our query, the buffer cache may need to manage only 1 or 2 blocks. It may not be evidenced as a better cache hit ratio (although it probably will be), and that is not the point really. A high cache hit ratio by itself doesn't mean the system is performing well. A reduced logical count and an SGA that needs less block buffer cache to achieve the same cache hit would indicate good performance.

Now, we are ready to test the efficiency of our B*Tree cluster. In order to do this, we'll assess the performance of our nicely packed cluster against the equivalent heap tables, using SQL_TRACE and TKPROF.

One of the things we want to observe is how much less, or more, physical I/O our cluster implementation will be compared to the heap table approach. So, the first thing to do is flush the buffer cache of all data related to the tablespace is which our cluster is stored. In order to do this, we simply take the tablespace offline, and then put it back online:

```
ops$tkyte@ORA920> alter tablespace users offline;
Tablespace altered.

ops$tkyte@ORA920> alter tablespace users online;
Tablespace altered.
```

Now for the benchmark code itself. We will retrieve all of the data from the two tables, cluster key by cluster key. We will do this ten times in all, just to have it run many times, over and over.

NOTE
The / CLUSTER */ in the query is not a hint. It is only there so that when we look at the TKPROF report, we can distinguish our two implementations easily.*

```
ops$tkyte@ORA920> alter session set sql_trace=true;
Session altered.

ops$tkyte@ORA920> begin
  2      for x in ( select username from all_users )
  3      loop
  4          for i in 1 .. 10
  5          loop
  6              for y in ( select a.username,
  7                                a.temporary_tablespace,
  8                                b.object_name ,
  9                                b.object_type /* CLUSTER */
 10                           from user_info a, users_objects b
 11                          where a.username = b.owner
 12                            and a.username = X.USERNAME )
 13              loop
 14                  null;
 15              end loop;
 16          end loop;
 17      end loop;
 18  end;
 19  /
```

```
PL/SQL procedure successfully completed.

ops$tkyte@ORA920> alter session set sql_trace=false;
Session altered.
```

Now that we have our report for the B*Tree cluster, we need to rebuild the tables as conventional heap tables. We use the same CREATE TABLE AS SELECT statements but remove the CLUSTER clause. Additionally, we add one index to each table.

```
ops$tkyte@ORA920> create index user_info_username_idx on user_info_heap(username);
Index created.

ops$tkyte@ORA920> create index user_objects_owner_idx on
  2  users_objects_heap(owner);
Index created.
```

On the heap tables, we use A.USERNAME = *bind_variable* and join by USERNAME to OWNER. We load these tables using the same multitable INSERT statement and run the same block of benchmark code, using /* HEAP */ in place of /* CLUSTER */ in the query. Now, in reviewing the TKPROF report, we see the following:

```
select a.username, a.temporary_tablespace,
       b.object_name , b.object_type /* CLUSTER */
  from user_info a, users_objects b
 where a.username = b.owner
   and a.username = :b1

call     count       cpu elapsed disk   query    current     rows
------- ------  -------- ------- ---- ------- ----------  -------
Parse        1      0.00    0.00    0       0          0        0
Execute    450      0.06    0.05    0       0          0        0
Fetch    47620      1.63    1.50   55   49730          0    47170
------- ------  -------- ------- ---- ------- ----------  -------
total    48071      1.69    1.56   55   49730          0    47170

Rows     Row Source Operation
-------  ---------------------------------------------------
  47170  MERGE JOIN CARTESIAN
  47170   TABLE ACCESS CLUSTER USERS_OBJECTS
    350    INDEX UNIQUE SCAN USER_OBJECTS_IDX
  47170   BUFFER SORT
    350    TABLE ACCESS CLUSTER USER_INFO
    350     INDEX UNIQUE SCAN USER_OBJECTS_IDX
******************************************************************
select a.username,
       a.temporary_tablespace,
       b.object_name ,
       b.object_type /* HEAP */
  from user_info_heap a, users_objects_heap b
 where a.username = b.owner
```

```
   and a.username = :b1

call     count       cpu elapsed disk   query    current     rows
------- ------  -------- ------- ---- ------- ----------  -------
Parse        1      0.00    0.00    0       0          0        0
Execute    450      0.06    0.06    0       0          0        0
Fetch    47620      2.32    2.10   74   95930          0    47170
------- ------  -------- ------- ---- ------- ----------  -------
total    48071      2.39    2.17   74   95930          0    47170

Rows     Row Source Operation
-------  ---------------------------------------------------
  47170  MERGE JOIN CARTESIAN
  47170   TABLE ACCESS BY INDEX ROWID USERS_OBJECTS
  47170    INDEX RANGE SCAN USER_OBJECTS_OWNER_IDX
  47170   BUFFER SORT
    350    TABLE ACCESS BY INDEX ROWID USER_INFO
    350     INDEX RANGE SCAN USER_INFO_USERNAME_IDX
```

NOTE
You may be wondering how the CPU time could exceed the elapsed time. Here, it is because Oracle was timing a lot of very short events (47,620 fetches). Each fetch was timed individually. Any fetch that took less time to complete than the unit of measurement would appear to happen "instantly," in 0.00 seconds. Generally, this error is not relevant, but when you time many thousands of very short events, it will creep in and become visible, as it did here.

The B*Tree cluster in this case did significantly less logical I/O (QUERY column) and less physical I/O. There were fewer blocks that needed to be read in from disk to accomplish the same task. If you scale this example up to hundreds of thousands or millions of records, you'll get the benefits of reduced physical I/O and increased buffer cache efficiency.

Instead of needing to read two or three index structures and blocks from multiple locations, you have one index structure (the cluster key index) to read. Also, in the best case, you have a single database block—one that contains all of the rows from all of the tables you need— to physically read into cache. You also have fewer index blocks and table blocks to cache. Whereas the conventional table query would have index blocks from two indexes (say three blocks at least per index) and table blocks from two tables (say one block at least for each table) for about eight blocks in cache, a cluster may have two or three index blocks plus one table block. You may be able to fit twice as much data into the cache, simply by having related data that is accessed together stored together.

Hash Clusters Eliminate Index Blocks

If we run through the same example from the previous section using a hash cluster rather than a B*Tree cluster, the TKPROF report for the cluster query shows the following:

```
select a.username, a.temporary_tablespace,
       b.object_name , b.object_type /* CLUSTER */
```

```
  from user_info a, users_objects b
 where a.username = b.owner
   and a.username = :b1

call     count       cpu elapsed disk   query current     rows
------- ------  -------- ------- ---- ------- -------  -------
Parse        1      0.00    0.00    0       0       0        0
Execute    450      0.05    0.05    0       4       0        0
Fetch    47610      1.67    1.46   61   49230       0    47160
------- ------  -------- ------- ---- ------- -------  -------
total    48061      1.73    1.51   61   49234       0    47160

Rows     Row Source Operation
-------  ---------------------------------------------------
  47160  MERGE JOIN CARTESIAN
  47160   TABLE ACCESS HASH USERS_OBJECTS
  47160   BUFFER SORT
    350    TABLE ACCESS HASH USER_INFO
```

If you recall, the totals for the B*Tree cluster and the conventional tables were, respectively:

```
call     count       cpu elapsed disk   query current     rows
------- ------  -------- ------- ---- ------- -------  -------
total    48071      1.69    1.56   55   49730       0    47170
total    48071      2.39    2.17   74   95930       0    47170
```

So, that is on par with the B*Tree cluster. However, here, the data is, in effect, the index. Oracle did not scan any indexes to find the data. Rather, it would take the USERNAME value passed into the query via the `:b1` bind variable, hash it to a block address, and go to that block.

If a B*Tree index can increase the buffer cache utilization by simply storing similar data together, a hash cluster goes one step further by eliminating the index blocks altogether. You can fit more relevant data in your buffer simply because you have less relevant data.

Single-Table Hash Clusters Are Useful for Read-Only Lookup Tables

A *single-table hash cluster* is a special case of a cluster, whereby only one table may exist in the cluster at any point in time. Oracle is able to optimize the access to this table, since it knows that the blocks contain data from exactly one table. Therefore, it is easier to process these blocks, because Oracle no longer has the overhead of remembering which rows come from which table.

Where would a single-table hash cluster be useful? Almost anywhere you do a lookup (turn a code into a value). Normally, each lookup would involve two or three index accesses, followed by a table access by ROWID. With a single-table hash cluster, those three or four logical I/Os may be turned into a single logical I/O.

To demonstrate this use, we'll set up a small test case. We'll use DBA_OBJECTS again and, knowing that we'll want to put about 50,000 lookup items in our single-table hash cluster, we'll create the cluster as follows:

```
ops$tkyte@ORA920> create cluster object_id_lookup
  2  ( object_id number )
```

```
single table
hashkeys 50000
size 100
/
Cluster created.
```

Here, we used the keyword SINGLE TABLE to indicate to Oracle we want to use this special-case table. Additionally, we used HASHKEYS 50000 to indicate the desired size of our hash table. Lastly, we know the rows are about 100 bytes on average for the data we are going to load into this hash table, so we used SIZE 100.

Now, we are ready to create a single-table hash cluster table as well as a heap lookup table in order to see the difference between the two as far as I/O goes. To do this, we'll populate 50,000 rows into each, using DBA_OBJECTS two times with a UNION ALL. Since my DBA_OBJECTS view has about 35,000 rows in it, this gave me more than 50,000 rows, so I used ROWNUM to generate just the 50,000 rows I wanted to test with:

```
ops$tkyte@ORA920> create table single_table_hash_cluster
( owner, object_name, object_id,
  object_type, created, last_ddl_time,
  timestamp, status )
cluster object_id_lookup(object_id)
as
select owner, object_name, rownum,
       object_type, created, last_ddl_time,
       timestamp, status
  from ( select * from dba_objects
         union all
         select * from dba_objects)
where rownum <= 50000
/
Table created.

ops$tkyte@ORA920> create table heap_table
( owner, object_name, object_id,
  object_type, created, last_ddl_time,
  timestamp, status,
  constraint heap_table_pk
    primary key(object_id) )
as
select owner, object_name, rownum,
       object_type, created, last_ddl_time,
       timestamp, status
  from ( select * from dba_objects
         union all
         select * from dba_objects)
where rownum <= 50000
/
Table created.
```

To compare the runtime characteristics of the two implementations, we'll query each and every record from both tables three times apiece. So, we'll do 150,000 lookups against each table in a loop:

```
ops$tkyte@ORA920> alter session set sql_trace=true;
Session altered.

ops$tkyte@ORA920> declare
  2      l_rec single_table_hash_cluster%rowtype;
  3  begin
  4      for iters in 1 .. 3
  5      loop
  6          for i in 1 .. 50000
  7          loop
  8              select * into l_rec
  9                from single_table_hash_cluster
 10               where object_id = i;
 11
 12              select * into l_rec
 13                from heap_table
 14               where object_id = i;
 15          end loop;
 16      end loop;
 17  end;
 18  /

PL/SQL procedure successfully completed.
```

When we look at the TKPROF report for these runs, we discover the following:

```
SELECT * from single_table_hash_cluster where object_id = :b1

call     count    cpu elapsed disk   query current    rows
------- ------  ----- ------- ---- ------- -------  -------
Parse        1   0.00    0.00    0       0       0        0
Execute 150000  13.60   14.01    0       2       0        0
Fetch   150000  12.93   12.64    0  189924       0   150000
------- ------  ----- ------- ---- ------- -------  -------
total   300001  26.54   26.65    0  189926       0   150000

Rows     Row Source Operation
-------  ---------------------------------------------------
 150000  TABLE ACCESS HASH SINGLE_TABLE_HASH_CLUSTER

**********************************************************
SELECT * from heap_table where object_id = :b1

call     count    cpu elapsed disk   query current    rows
------- ------  ----- ------- ---- ------- -------  -------
Parse        1   0.00    0.00    0       0       0        0
```

```
Execute 150000  16.36   17.86    0       0       0        0
Fetch   150000  11.02   11.05    0  450000       0   150000
------- ------  ----- ------- ---- ------- -------  -------
total   300001  27.38   28.91    0  450000       0   150000

Rows     Row Source Operation
-------  ---------------------------------------------------
 150000  TABLE ACCESS BY INDEX ROWID HEAP_TABLE
 150000   INDEX UNIQUE SCAN HEAP_TABLE_PK (object id 43139)
```

Note that the runtimes (CPU column) are more or less the same. If you ran this test over and over as I did, you would find the averages of the CPU times to be the same over time. The numbers of executes, parses, fetches, and rows are identical. These two tables gave the same output given the same inputs. The difference is all about I/O here. The QUERY column (consistent mode gets) values are very different.

The query against the single-table hash cluster did 42% of the logical I/Os that the conventional heap table performed. In this case, this did not show up in the runtimes (they appear equivalent) but as you've heard me say more than once in this book, "A latch is a lock, a lock is a serialization device, and serialization devices greatly inhibit concurrency and scalability." Anytime we can significantly reduce the amount of latching we perform, we will naturally increase our scalability and performance in a multiple-user environment. If instead of using SQL_TRACE and TKPROF, we used Runstats on the above test, we would find that the single-table hash cluster uses about 85% of the cache buffers chains latches (latches used to protect the buffer cache; we use fewer of them solely because we use the buffer cache less).

For applications that have read-only/mostly lookup tables, or those where the number of reads of a table by a key value is enormous as compared to the modifications, a single-table hash cluster can be the difference between simply performing and exceeding performance requirements.

Clusters Wrap-Up

B*Tree clusters have their pros and cons. We've reviewed many of their advantages:

- Physical colocation of data
- Increased buffer cache efficiency
- Decreased logical I/Os
- Decreased need for indexes, as all tables share one cluster key index

However, B*Tree clusters do have some limitations. Along with their inability to do direct-path loading or partitioning, there are some general disadvantages:

- Correctly sizing clusters requires thought. Setting a SIZE value too large will waste space; setting a SIZE value too small can remove the main benefit of the cluster (physical clustering of data). In order to correct an incorrectly sized cluster, you would need to rebuild the cluster.
- Insertions into a cluster must be controlled or controllable, or else the clustering effect may deteriorate.

- Clustered tables are slower to insert into than conventional tables, since the data has a location it must go into. A heap table can just add data wherever it pleases; it does not need to work hard to keep the data in some specific location. That does not mean that clustered tables are not appropriate for read/write applications. Consider the data dictionary of Oracle itself!

So, think about where you may be able to use these data structures. They can be used in a wide variety of applications, especially after you understand their benefits as well as their limitations. Many tables are never direct-path loaded or partitioned and do have controlled patterns of insertion. For these objects, consider a cluster.

Index-Organized Tables (IOTs)

An IOT is basically a table stored in an index. It is similar in nature to a B*Tree cluster in that data is stored physically colocated by a key value, but it differs in the following ways:

- There is a single data structure, a single index structure, whereas a B*Tree cluster has an index and a data segment.
- The data is stored sorted by key, unlike a B*Tree cluster where the data is organized by key value, but the keys themselves are not stored sorted.
- Sizing the IOT is somewhat easier than sizing a cluster. You do not need to estimate the maximum number of keys as with a hash cluster, and you have more flexibility in how you size the amount of data stored by key.
- IOTs are very useful in a couple of implementation areas. One is as association tables—tables you use in a many-to-many relationship. The other is tables where physical colocation of data is important, but the order of insertion is unpredictable, or the data arrives in an order that makes it impossible for a cluster to maintain the data colocated over time.

Use IOTs as a Space-Saving Alternative to Association Tables

Association tables generally are comprised of two columns or two keys and are used to relate two tables together. In the Oracle data dictionary, you could think of DBA_TAB_PRIVS as such an association object between DBA_USERS and DBA_OBJECTS. A single user may have one or more privileges on a given object; that given object may have one or more users that have privileges on it.

For association tables, normally you would set up a structure such as this:

```
create table association
( primary_key_table1,
  primary_key_table2,
  <possibly some columns pertaining to the relationship> );

create index association_idx1 on
association(primary_key_table1, primary_key_table2 );
```

```
create index association_idx2 on
association(primary_key_table2, primary_key_table1 );
```

So, you would have three structures: a table and two indexes. The indexes allow you to traverse either from TABLE1 to all related TABLE2 rows, or vice versa. In most implementations, the table itself is never even accessed; it is a redundant data structure that is considered a necessary evil, as it just consumes space. In some relatively rare cases, it contains additional data specific to the relationship, but this data is usually small in volume.

Using an IOT, we can get the same effect:

```
create table association
( primary_key_table1,
  primary_key_table2,
  <possibly some columns pertaining to the relationship>,
  primary key(primary_key_table1,primary_key_table2) )
organization index;

create index association_idx on
association(primary_key_table2);
```

We've removed the need for the table. Not only that, but if we need to retrieve the information pertaining to the relationship between two rows in TABLE1 and TABLE2, we've eliminated the TABLE ACCESS BY ROWID step. Note that the secondary index on the ASSOICATION table did not include both columns. This is a side effect of using an IOT the logical ROWID that is used for IOT's in the index structure and the value for primary_key_table1 is already there! We can see that with this small example:

```
ops$tkyte@ORA920> create table t
  2  ( a int,
  3    b int,
  4    primary key (a,b)
  5  )
  6  organization index;
Table created.

 ops$tkyte@ORA920> create index t_idx on t(b);
Index created.

ops$tkyte@ORA920> set autotrace traceonly explain
ops$tkyte@ORA920> select a, b from t where b = 55;

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=26)
   1    0   INDEX (RANGE SCAN) OF 'T_IDX' (NON-UNIQUE)
            (Cost=2 Card=1 Bytes=26)
```

Notice how there is no "TABLE ACCESS BY ROWID" step there. Even though column A is requested in the query, Oracle knows that it can get the value of column A from the logical ROWID in the index structure and hence does not go to the table to get it. Using this fact can save you much disk space in the creation of your secondary indexes on the IOT.

Use IOTs to Colocate Randomly Inserted Data

In addition to being a space-saving device, by obviating the need for some redundant tables, IOTs excel in their ability to physically colocate related information for fast access. If you recall, one of the downsides to a cluster is that you need to have some control over the arrival of the data in order to optimize its physical colocation. IOTs do not suffer from this condition, because they will readjust themselves structurally in order to accommodate the data as it is inserted.

Consider an application that frequently retrieves a list of documents owned by a given user. In the real world, the user would not insert all of the documents he or she will ever own in a single session. This is a dynamic list of unpredictable size that will constantly be changing as the user adds and removes documents in the system. Therefore, in a traditional heap table, the rows that represent this user's documents would be scattered all over the place. Consider what would happen when you run a query such as this:

```
select * from document table where username = :bind_variable
```

Oracle would use an index to read many dozens of blocks from all over the table. If we used an IOT to physically cluster the data together, this would not happen. We can observe this behavior with a simple simulation and AUTOTRACE. For this example, we'll set up a pair of tables: one using an IOT and the other using a heap-based implementation.

```
ops$tkyte@ORA920> create table iot
  2  ( username varchar2(30),
  3    document_name varchar2(30),
  4    other_data       char(100),
  5    constraint iot_pk
  6    primary key (username,document_name)
  7  )
  8  organization index
  9  /
Table created.

ops$tkyte@ORA920> create table heap
  2  ( username varchar2(30),
  3    document_name varchar2(30),
  4    other_data       char(100),
  5    constraint heap_pk
  6    primary key (username,document_name)
  7  )
  8  /
Table created.
```

We use a CHAR(100) just to make the average width of a row in these tables about 130 bytes or so, since a CHAR(100) will always consume 100 characters of storage (it is a fixed-width datatype). So, the only difference between these two tables (besides their names) is the addition of the ORGANIZATION INDEX clause. That instructs Oracle to store the table data in an index segment instead of a table segment, so that all of the data for that table will be stored in an index structure.

Next, we populate these tables with some sample data. We construct a loop that adds 100 documents for each user in the ALL_USERS table. We do this in a fashion that emulates real life, in that the documents for a given user are added not all at once, but rather over time after many other documents have been added by other users.

```
ops$tkyte@ORA920> begin
    for i in 1 .. 100
    loop
        for x in ( select username
                     from all_users )
        loop
            insert into heap
            (username,document_name,other_data)
            values
            ( x.username, x.username || '_' || i, 'x' );

            insert into iot
            (username,document_name,other_data)
            values
            ( x.username, x.username || '_' || i, 'x' );
        end loop;
    end loop;
    commit;
end;
/
PL/SQL procedure successfully completed.
```

And now we are ready for our performance comparison. Here, we will read all of the data from our table, user by user; that is, for USER1, we'll read out all of the rows that correspond to that user, and then for USER2, and so on. Additionally, we'll do this reading in two ways—one time using BULK COLLECT and the next using single-row fetches—just to see the difference that array processing can have on performance and scalability, and to show how significantly different this IOT can be from a heap table. Our benchmark routine is as follows:

```
ops$tkyte@ORA920> alter session set sql_trace=true;
Session altered.

ops$tkyte@ORA920> declare
    type array is table of varchar2(100);
    l_array1 array;
    l_array2 array;
    l_array3 array;
begin
for i in 1 .. 10
loop
```

```
    for x in (select username from all_users)
    loop
        for y in ( select * from heap single_row
                    where username = x.username )
        loop
            null;
        end loop;
        for y in ( select * from iot single_row
                    where username = x.username )
        loop
            null;
        end loop;
        select * bulk collect
          into l_array1, l_array2, l_array2
          from heap bulk_collect
         where username = x.username;
        select * bulk collect
          into l_array1, l_array2, l_array2
          from iot bulk_collect
         where username = x.username;
    end loop;
end loop;
end;
/
PL/SQL procedure successfully completed.
```

The TKPROF report for the single-row fetches shows the following statistics:

```
select * from heap single_row where username = :b1

call     count    cpu elapsed disk   query current     rows
------- ------  ----- ------- ---- ------- -------  -------
Parse        1   0.00    0.00    0       0       0        0
Execute    440   0.05    0.05    0       0       0        0
Fetch    44440   1.50    1.42    0   88886       0    44000
------- ------  ----- ------- ---- ------- -------  -------
total    44881   1.56    1.48    0   88886       0    44000

Rows     Row Source Operation
-------  ---------------------------------------------------
  44000  TABLE ACCESS BY INDEX ROWID HEAP
  44000   INDEX RANGE SCAN HEAP_PK (object id 43271)
***********************************************************
select * from iot single_row where username = :b1

call     count    cpu elapsed disk   query current     rows
------- ------  ----- ------- ---- ------- -------  -------
Parse        1   0.00    0.00    0       0       0        0
Execute    440   0.07    0.05    0       0       0        0
Fetch    44440   1.11    0.99    0   44987       0    44000
```

```
------- ------  ----- ------- ---- ------- -------  -------
total    44881   1.18    1.04    0   44987       0    44000

Rows     Row Source Operation
-------  ---------------------------------------------------
  44000  INDEX RANGE SCAN IOT_PK (object id 43273)
```

The heap table, with its read-the-index-then-the-table approach, will do at least twice the I/O row by row by row. It must read the index block, and then read the table block. The IOT approach, on the other hand, simply reads the index and is finished. So, this is pretty good. Any day we can cut in half the number of I/Os our system must perform is a good day. Can it be even better? Yes, it can.

In Chapter 2, we used AUTOTRACE to demonstrate the effect that different array sizes (how many rows Oracle fetches in response to each fetch request) could have on an application. We saw that we can significantly reduce the I/O performed by a query if we fetch many rows at a time instead of fetching single rows. Here, we can really show the power of the IOT over the heap table in this regard. Reading further on in the TKPROF report we see this:

```
SELECT * from heap bulk_collect where username = :b1

call     count    cpu elapsed disk   query current     rows
------- ------  ----- ------- ---- ------- ------- --------
Parse        1   0.00    0.00    0       0       0        0
Execute    440   0.06    0.05    0       0       0        0
Fetch      440   0.49    0.48    0   36100       0    44000
------- ------  ----- ------- ---- ------- ------- --------
total      881   0.55    0.54    0   36100       0    44000

Rows     Row Source Operation
-------  ---------------------------------------------------
  44000  TABLE ACCESS BY INDEX ROWID HEAP
  44000   INDEX RANGE SCAN HEAP_PK (object id 43271)
***********************************************************
SELECT * from iot bulk_collect where username = :b1

call     count    cpu elapsed disk   query current     rows
------- ------  ----- ------- ---- ------- -------  -------
Parse        1   0.00    0.00    0       0       0        0
Execute    440   0.06    0.05    0       0       0        0
Fetch      440   0.24    0.24    0    2110       0    44000
------- ------  ----- ------- ---- ------- -------  -------
total      881   0.31    0.30    0    2110       0    44000

Rows     Row Source Operation
-------  ---------------------------------------------------
  44000  INDEX RANGE SCAN IOT_PK (object id 43273)
```

The IOT using bulk fetches did less than 6% of the I/O of the query against the heap table using the same bulk collect. When running this test with Runstats instead of TKPROF, I discovered the heap table approach used *600% more latches*. These are some serious differences here!

IOTs Wrap-Up

Much like clusters, IOTs are useful data structures for physically organizing your data on disk with the goal of improved access times. They have the same good attributes as clusters do:

- Physical colocation of data
- Increased buffer cache efficiency
- Decreased logical I/Os
- Decreased need for indexes; the table is the index

IOTs also have some advantages that clusters do not offer:

- The data is actually stored sorted by primary key, which may be of great benefit.
- Since an index is a fairly sophisticated data structure, it has the ability to move rows around (something a cluster cannot do), which allows it to better pack the related data together. The order of insertion, something to consider with clusters, it not as much of a factor with IOTs.
- They can be rebuilt (reorganized) online if the rare need to do this arises.

However, they do have some limitations. Along with the inability to do direct-path loading, there are some general disadvantages of using IOTs:

- As with clustered tables, IOT tables are slower to insert into than conventional tables, since the data has a location it must go into.
- You may need to consider the overflow segment. There are issues with very wide tables in an IOT structure. They are generally best suited for tall, "skinny" tables, but with some foresight, they can work with wide tables as well.

External Tables

External tables are new with Oracle9i Release 1. They add to the growing list of ORGANIZATION clauses on tables. The ORGANIZATION EXTERNAL clause is used for a table whose data is stored outside the Oracle database.

External tables give us the ability to select from flat files, either delimited files or fixed-width positional files. You cannot modify these table types; you can just query them. They cannot be conventionally indexed, because the rows have no ROWIDs. Because you freely edit the files at anytime, moving the rows around, Oracle cannot index them in any fashion.

I recommend using external tables only for the loading and merging of data, not as a replacement for true database tables. If you find yourself querying external tables in an application, you are most likely misusing them. If you find yourself using them to load and reload data on an ongoing basis, you've found their purpose in life.

Set Up External Tables

External tables rely on DIRECTORY objects in Oracle. You create a DIRECTORY object using a SQL DDL command such as this:

```
ops$tkyte@ORA920> create or replace directory data_dir as '/tmp/'
  2  /
Directory created.
```

This sets up a directory DATA_DIR that points to the /tmp/ directory in the file system.

The next step is to create a table that tells Oracle how to read this file:

```
ops$tkyte@ORA920> create table external_table
  2  (EMPNO NUMBER(4) ,
  3   ENAME VARCHAR2(10),
  4   JOB VARCHAR2(9),
  5   MGR NUMBER(4),
  6   HIREDATE DATE,
  7   SAL NUMBER(7, 2),
  8   COMM NUMBER(7, 2),
  9   DEPTNO NUMBER(2)
 10  )
 11  ORGANIZATION EXTERNAL
 12  ( type oracle_loader
 13    default directory data_dir
 14    access parameters
 15    ( fields terminated by ',' )
 16    location ('emp.dat')
 17  )
 18  /
Table created.
```

Most of this looks like a typical CREATE TABLE statement, until you get to the part in bold. This code should look familiar to anyone who uses SQLLDR, because it looks a bit like a control file used by that tool. In fact, if we compare the preceding listing to the equivalent SQLLDR control file, we see a lot of similarities:

```
LOAD DATA
INFILE /tmp/emp.dat
INTO TABLE emp
REPLACE
FIELDS TERMINATED BY ','
(empno,ename,job,mgr,hiredate,sal,comm,deptno)
```

We listed the column names, we have a FIELDS TERMINATED BY clause, and we have a location from which to read the data (INFILE)—all of the things an external table has.

The similarities to SQLLDR are not by chance. The relationship between SQLLDR and external tables is very close in Oracle9i—so close in fact that SQLLDR may be used as a front-end to external tables. What I mean by that is, if you have many existing control files for SQLLDR and would like to take advantage of external tables, you can do so easily using the EXTERNAL_TABLE= GENERATE_ONLY parameter to SQLLDR. It will convert a control file into a CREATE TABLE statement for you, easing the migration from SQLLDR to external tables. For example, suppose you have the following SQLLDR control file named T.CTL:

```
LOAD DATA
INFILE *
INTO TABLE emp
REPLACE
FIELDS TERMINATED BY '|'
( empno ,ename ,job ,mgr ,hiredate ,sal ,comm ,deptno)
```

You can run the following command:

```
$ sqlldr / t.ctl external_table=generate_only
SQL*Loader: Release 9.2.0.3.0 - Production on Fri Jul 4 12:42:35 2003
Copyright (c) 1982, 2002, Oracle Corporation.  All rights reserved.
```

When you edit the resulting T.LOG file, you'll discover in it this create table statement (I edited some columns out to make it small):

```
CREATE TABLE "SYS_SQLLDR_X_EXT_EMP"
(
  EMPNO NUMBER(4),
…
  DEPTNO NUMBER(2)
)
ORGANIZATION external
(
  TYPE oracle_loader
  DEFAULT DIRECTORY SYS_SQLLDR_XT_TMPDIR_00000
  ACCESS PARAMETERS
  (
    RECORDS DELIMITED BY NEWLINE CHARACTERSET US7ASCII
    BADFILE 'SYS_SQLLDR_XT_TMPDIR_00000':'t.bad'
    LOGFILE 't.log_xt'
    READSIZE 1048576
    SKIP 7
    FIELDS TERMINATED BY "|" LDRTRIM
    REJECT ROWS WITH ALL NULL FIELDS
    (
      EMPNO CHAR(255) TERMINATED BY "|",
…
      DEPTNO CHAR(255) TERMINATED BY "|"
    )
  )
  location
  (
    't.ctl'
  )
)REJECT LIMIT UNLIMITED
```

So, if you are ever stuck on the syntax for creating an external table, but you knew how to use SQLLDR to do it, or you have many control files to migrate to this new facility, this method is something to consider trying.

But, getting back to the original example, now, all we need is some data. In order to demonstrate this easily, we'll use a utility I have posted on the Internet at http://asktom.oracle.com/~tkyte/flat/index.html. There, I have three different data unloaders available for download: one that uses SQL*Plus, another that uses a Pro*C version, and a third using PL/SQL implementation. Each has its uses, but for this example, the easiest to use is the SQL*Plus version called `flat` (for flat file). Here, we'll unload the EMP table to /tmp/:

```
ops$tkyte@ORA920> host flat scott/tiger emp > /tmp/emp.dat
ops$tkyte@ORA920> host head /tmp/emp.dat
7369,SMITH,CLERK,7902,17-DEC-80,800,,20
7499,ALLEN,SALESMAN,7698,20-FEB-81,1600,300,30
7521,WARD,SALESMAN,7698,22-FEB-81,1250,500,30
…
```

As you can see, we have a simple delimited file from which to work. To use it, all we need do is this:

```
ops$tkyte@ORA920> select ename, job, hiredate
  2    from external_table
  3  /

ENAME      JOB       HIREDATE
---------- --------- ---------
SMITH      CLERK     17-DEC-80
ALLEN      SALESMAN  20-FEB-81
WARD       SALESMAN  22-FEB-81
…
```

```
MILLER     CLERK     23-JAN-82

14 rows selected.
```

Every time we execute this query, Oracle will open, read, and close that file in the operating system. None of the "blocks" are buffered in the conventional sense with an external table. Oracle always reads the data from the disk into the session space, bypassing the buffer cache. Since Oracle does not own this data, it must be read from the disk each time is it accessed. This can make interpreting a TKPROF or AUTOTRACE report against an external table a bit misleading, because the I/O for the external table (logical or physical) will not be accurate; it is different from normal Oracle block I/O.

So, very basically, that is all there is to get started with external tables. There is one other important note to beware of: The file to be read must be visible on the database server machine. It cannot be a file on your local PC client. It cannot be a file on an application server separate from the database machine. The database processes themselves must be able to see that file, open that file, and read that file. If they cannot, they will not be able to make use of it at all.

On Unix, this is somewhat easy to get around: You can simply use NFS mounts and make other file systems available to all processes on the database server. So, if you wanted a file that physically resides on your application server to be accessible on the database machine, you would export that file system from the application server and mount it on the database machine.

If you are using Windows shares, the rules are a bit different. Windows is primarily a single-user operating system. The file systems you see when you log in are not the file systems that every process on that machine can see. Each user has his or her own set of network drives that may or may not be visible to that user. Under Windows, this takes a bit of work to set up, to make it so the database server process can actually see a remote file system. For information about the necessary Windows-level setup for making a network drive visible to the user who runs the database itself, see Oracle Support Note 45172.1, available on http://metalink.oracle.com.

Modify External Tables

You cannot use an UPDATE, an INSERT, or a DELETE statement on an external table. To modify external tables, you must use an operating system tool or a tool like UTL_FILE, which can write operating system files. However, as soon as data is added or modified in the operating system file, Oracle will see it immediately.

```
ops$tkyte@ORA920> host echo -
>'1234,KYTE,DBA/DEV,7782,23-JAN-2003,1300,,10' >> /tmp/emp.dat

ops$tkyte@ORA920> select ename, job, hiredate
  2    from external_table
  3   where ename = 'KYTE'
  4  /

ENAME      JOB       HIREDATE
---------- --------- ---------
KYTE       DBA/DEV   23-JAN-03
```

Use External Tables for Direct-Path Loading

One of the main uses of SQLLDR over the years has been its high-speed direct-path load capability. Well, SQLLDR is no longer the only tool in our shed, because external tables do this quite nicely as well. You can use one of the following methods to perform direct-path loads:

- Script it on the command line with SQLLDR.
- Use an INSERT statement.
- Use a CREATE TABLE AS SELECT statement.

NOTE
In the past, I was frequently asked how one could invoke SQLLDR, Oracle's loading tool, from a stored procedure or trigger. It was difficult at best, because you needed to set up a Java stored procedure or C-based external procedure in order to execute a host command and run the SQLLDR program. This is no longer the case. Now, in order to invoke SQLLDR, all you need to do is run a SELECT statement; SQLLDR has been burned into the Oracle database itself. In fact, it would be fair to say that running SQLLDR from the command line is old-fashioned. Using an INSERT INTO statement or CREATE TABLE AS SELECT or MERGE command to load the data transactionally in the database is the way to go.

Let's set up a test with BIG_TABLE (ALL_OBJECTS copied repeatedly so that it has more than a million rows). We will use one of the data unloaders from http://asktom.oracle.com/~tkyte/flat/index.html to create a simple flat-file extract of these rows. Then we will direct-path load this data using all three methods. The SQLLDR control file looks like this:

```
LOAD DATA
INFILE big_table.dat
INTO TABLE big_table
truncate
FIELDS TERMINATED BY ','
trailing nullcols
(
owner ,object_name ,subobject_name ,object_id
,data_object_id ,object_type
,created date 'dd-mon-yyyy hh24:mi:ss'
,last_ddl_time date 'dd-mon-yyyy hh24:mi:ss'
,timestamp ,status ,temporary ,generated ,secondary
)
```

And here is the external table definition:

```
create table big_table_external
( OWNER                 VARCHAR2(30), OBJECT_NAME         VARCHAR2(30),
  SUBOBJECT_NAME        VARCHAR2(30), OBJECT_ID           NUMBER,
  DATA_OBJECT_ID        NUMBER,       OBJECT_TYPE         VARCHAR2(18),
  CREATED               DATE,         LAST_DDL_TIME       DATE,
  TIMESTAMP             VARCHAR2(19), STATUS              VARCHAR2(7),
  TEMPORARY             VARCHAR2(1),  GENERATED           VARCHAR2(1),
  SECONDARY             VARCHAR2(1)
)
ORGANIZATION EXTERNAL
( type oracle_loader
  default directory data_dir
  access parameters
  (
    fields terminated by ','
    missing field values are null
    ( owner ,object_name ,subobject_name ,object_id ,data_object_id
      ,object_type,created date 'dd-mon-yyyy hh24:mi:ss'
      ,last_ddl_time date 'dd-mon-yyyy hh24:mi:ss'
      ,"TIMESTAMP" ,status ,temporary ,generated ,secondary
    )
  )
    location ('big_table.dat')
  )
```

The results of repeated runs are shown in Table 7-1. (These results are from a NOARCHIVELOG mode database; if you are using an ARCHIVELOG mode database, make sure that BIG_TABLE is set to NOLOGGING to reproduce similar findings.)

Method	**CPU**	**Elapsed**	**Rows**
SQLLDR direct=true	29	42	1,833,792
External table INSERT /*+APPEND*/	33	38	1,833,792
External table CREATE TABLE AS SELECT	32	37	1,833,792
External table INSERT (conventional path)	42	130	1,833,792
SQLLDR (conventional path)	50	410	1,833,792

TABLE 7-1. *Comparing direct-path load methods*

These results show that using an external table adds some nominal overhead compared with using SQLLDR when in direct-path mode. In conventional-path loads, external tables really start to shine. But also consider these advantages of external tables (long-time users of SQLLDR will recognize these limits):

- You can join an external table to another table during a load to do lookups directly in the load itself (less postprocessing).
- You can use any SQL predicate you can dream of to filter the data. The ability to filter the data in SQLLDR is somewhat primitive by comparison.
- You can do a direct-path load from a remote client without needing to telnet into the server itself.
- You can invoke SQLLDR functionality from within a stored procedure simply by using INSERT.

As you can see, the advantages and ease of using external tables over SQLLDR from the command line are compelling.

Use External Tables for Parallel Direct-Path Loading

In the past, performing direct-path parallel loads using command-line SQLLDR was a bit of a pain. In order to accomplish this, you needed to script it all yourself. You decided the degree of parallelism, you split up the input files, you started *N* copies of SQLLDR in parallel, and you monitored them all. With external tables, this becomes as easy as this:

```
Alter table external_table parallel;
Create table T as select * from external_table;
```

That is it. If you are using parallel automatic tuning, Oracle will determine the degree of parallelism given the current system load, Oracle will split up the input files, and Oracle will start and monitor *N* parallel query slaves for you to process the load. This can be quite convenient for performing mass bulk loads of data.

Use External Tables for Merging

The technique for merging described here is a combination of another new feature in Oracle9i Release 1 and later and external tables. Oracle9i Release 1 added the MERGE command to the SQL language. This is the ability to take two tables and merge them together by a common key. If a record exists in both tables, you can process it as an update. If a record exists in only the table you are merging from, you can insert it into the table you are merging into.

For example, suppose you have an existing table of employee information and another group gives you a file of updates. This file of updates contains both updates to existing records and data for new hires. In the past, you needed to process the changes something like this:

- Using SQLLDR, load the file into a staging table.
- For each record in the staging table, attempt to update the existing record.
- If no rows were updated, insert the record as new.

Not only is that a lot of code—a lot of procedural processing—it is definitely not the fastest way to perform this operation. Processing sets of data is best done a set at a time. Row-at-a-time procedural code is almost always slower than a single SQL statement doing the same thing.

Enter external tables and the MERGE command. As an example, we'll use the same EMP and EXTERNAL_TABLE as we did earlier in setting up external tables, except that we'll modify the existing EMP table data as follows:

```
ops$tkyte@ORA920> delete from emp where mod(empno,2) = 1
  2  /
4 rows deleted.

ops$tkyte@ORA920> update emp set sal = sal/2
  2  /
10 rows updated.
```

Here, we've removed four of the existing employees from our database and modified the salaries of the rest. Now, suppose that someone gives us that delimited file of employee information, which we place into /tmp/emp.dat. We've already set up the external table, so we are ready to reconcile what is in the flat file with what is in the database. To accomplish this, we simply do the following:

```
ops$tkyte@ORA920> merge into EMP e1
  2  using EXTERNAL_TABLE e2
  3  on ( e2.empno = e1.empno )
  4  when matched then
  5   update set e1.sal = e2.sal
  6  when not matched then
  7   insert (empno, ename, job, mgr, hiredate, sal, comm, deptno)
  8   values ( e2.empno, e2.ename, e2.job,
               e2.mgr, e2.hiredate, e2.sal, e2.comm, e2.deptno )
  9  /
14 rows merged.
```

And that is it. The file has been synchronized with the database table. For any record that existed both in the file as well as the database table, we updated the salary column in the database table from the file. Likewise, for any record that was in the file but was not found in the table, we inserted the record.

Handle Errors with External Tables

External tables are a little lacking in handling rejected or bad records. SQLLDR suffered from the same problem. With SQLLDR, you more or less needed to manually inspect the log file to see if the load was successful.

With external tables, you can set up named log and bad files. By default, the log and bad files will be written to the same directory as the input file itself, but you are free to change this. You can also set up the name of a log or bad file so that it is in some well-known, named location. Here is an example:

```
ops$tkyte@ORA920> create table external_table
(EMPNO NUMBER(4) ,
 ENAME VARCHAR2(10),
 JOB VARCHAR2(9),
 MGR NUMBER(4),
 HIREDATE DATE,
 SAL NUMBER(7, 2),
 COMM NUMBER(7, 2),
 DEPTNO NUMBER(2)
)
ORGANIZATION EXTERNAL
( type oracle_loader
  default directory data_dir
  access parameters
  (
    records delimited by newline
    badfile data_dir:emp_external_table
    fields terminated by ','
  )
  location ('emp.dat')
)
reject limit unlimited
/

Table created.
```

Now, if some records in the input file EMP.DAT are invalid, instead of raising an error when we select from the external table, Oracle will write them to a bad file (emp_external_table.bad) in our directory. But how can we programmatically inspect these bad records and the log that is generated? Fortunately, that is easy to do by using yet another external table. Suppose we set up this external table:

```
ops$tkyte@ORA920> create table emp_external_table_bad
( text1 varchar2(4000) ,
  3    text2 varchar2(4000) ,
  4    text3 varchar2(4000)
  5  )
  6  organization external
  7  (type oracle_loader
  8   default directory data_dir
  9   access parameters
 10   (
 11     records delimited by newline
 12     fields
 13     missing field values are null
 14     ( text1 position(1:4000),
 15       text2 position(4001:8000),
 16       text3 position(8001:12000)
 17     )
 18   )
 19   location ('emp_external_table.bad')
 20  )
 21  /
```

This is just a table that can read any file without failing on a datatype error, as long as the lines in the file are less than 12,000 characters (simply add more fields if your input files are wider than 12,000 characters).

Let's add a row to our flat file to see how this works. Here, the record we add has yyy in the year portion of the date, so that record will be rejected:

```
ops$tkyte@ORA920> host echo '1234,KYTE,DBA/DEV,7782,23-JAN-yyy,1300,,10' >> /tmp/emp.dat

ops$tkyte@ORA920>
ops$tkyte@ORA920> select ename, job, hiredate
  2    from external_table
  3   where ename = 'KYTE'
  4  /
no rows selected
```

We can clearly see the rejected records via a simple query:

```
ops$tkyte@ORA920> select substr(text1,1,60) from emp_external_table_bad
  2  /

SUBSTR(TEXT1,1,60)
------------------------------------------------------------
1234,KYTE,DBA/DEV,7782,23-JAN-yyy,1300,,10
```

A COUNT(*) would tell us how many records were rejected. Another external table created on the log file associated with this external table would tell us why the record was rejected. You would need to go one step further to make this a repeatable procedure, however. There are two approaches:

- Capture the COUNT(*) from the bad file before you access the external table and make sure it is unchanged after you query it.
- Use UTL_FILE and reset the bad file—truncate it, in effect—by simply opening it for write and closing it.

In that fashion, you'll be able to tell if the bad records in the bad file were generated by you just recently or if they were left over from some older version of the file itself and are not meaningful.

Indexing Techniques

Oracle has a wide variety of indexing techniques that you might consider for optimizing data access:

- **B*Tree** The conventional, standard type of index that everyone uses.
- **Reverse-key index** A B*Tree index where the bytes are reversed. This is used to randomly distribute modifications to an index structure over the entire structure that may otherwise happen to a single block with a monotonically increased value (like a sequence).
- **Descending index** A B*Tree index where one or more of the fields is stored in descending sort order. For example, you can have an index on a table T(C1,C2 DESC,C3). The values for C1 and C3 in the index will be sorted in ascending fashion, whereas C2 will be sorted in descending order.
- **IOT** A table stored in an index, as described earlier in the "Index-Organized Tables (IOTs)" section.
- **B*Tree cluster index** An index that must be created to store data in a cluster, as described in the "Create Clusters" section earlier in the chapter.
- **Bitmap index** An index in which a single index entry may point to many rows using a bitmap. Normally, in a B*Tree index, a single index entry points to a single row. With a bitmap index, a single index entry may point to many hundreds of rows or more.
- **Function-based index (FBI)** An index on a complex calculation or a built-in or user-defined function. Instead of indexing ENAME in the EMP table, you could, for example, index UPPER(ENAME) to provide for fast, case-insensitive searches.
- **Domain index** An index you could build yourself (if you feel you have a superior or data-specific indexing technique). Also, the developers at Oracle have created domain indexes on nonstructured datatypes such as text, video, audio, image, and spatial data.

For details about the basic Oracle index types, see the Oracle9i Release 2 *Concepts Guide*, Chapter 10, the "Indexes" section.

In this section, we'll look at using two index types: FBIs and domain indexes.

Use FBIs—Think Outside the Box

FBIs were added to Oracle8i in version 8.1 of the database. They are currently available with the Oracle Enterprise and Personal Editions (but not the Standard Edition). FBIs give you the ability to index computed columns and use these indexes in a query. In a nutshell, this capability allows you to do the following:

- Have case-insensitive searches or sorts.
- Search on complex equations.
- Extend the SQL language efficiently by implementing your own functions and operators and then searching on them.

The nicest qualities of FBIs are that they are easy to implement and provide immediate, *transparent* value. They can be used to speed up existing applications without changing any of their logic or queries. It is not often that you can say such a thing about a database feature.

Here, we'll look at a couple of unique uses of FBIs. These are things that perhaps the inventors of this feature did not anticipate but are nonetheless not only possible but natural, once you start thinking about them.

"Thinking outside the box" is a popular saying. It means to not be confined to conventional wisdom, to tried-and-true methods—to go beyond where you've gone before. To think of new, unusual, creative solutions using the tools you have at your disposal. It is what the engineers did in Houston, Texas when Apollo 13 declared, "Houston, we have a problem." They spent time in a room with nothing more than the astronauts in space had and thought outside the box in order to solve the problem at hand.

Oracle has a lot of features and functionality. The documentation is very good at telling us how things work and what functions and features are available. But the documentation will never give us the creative solution to the particular problem we are facing. Coming up with that creative solution is our job: To put the disparate pieces together, in some unique, not necessarily anticipated fashion and to solve our technical problem in the simplest form possible. To that end, we will use two concrete examples that build on a pair of facts. These are the documented facts:

- A B*Tree index will never have an entirely NULL entry. If all of the columns in the index key are NULL, there will not be an entry in the B*Tree for that column. If you create an index on a million-row table using a column that is NULL for every row, that index will be empty.
- We have FBIs that can incorporate complex logic in them.

Using these two facts, we will set about to solve two common issues simply and elegantly. These problems are how to index selectively and how to achieve selective uniqueness.

Index Selectively

Frequently, I hear questions like, "How can I index only some of the rows in a table?" and "Can I index a specific WHERE clause but not for all of the rows in the table?" Many times, this has to do with creating an index for a Y/N-type column—a processed flag, for example. You might have a table with a flag indicating whether a record has been processed, and the vast preponderance of records have Y as their value; only a few rows have an N to indicate that they are awaiting processing. You are looking for an index that will find the N records, without indexing the Y records.

What immediately pops into most people's heads at this point is, "Aha, this is a bitmap index we are looking for." They base that on what they've heard about bitmap indexes. A bitmap index is useful for indexing low-cardinality data, and what could be lower cardinality than just Y and N values? The problem is that these solutions are generally needed in a system that performs a lot of single-row updates to the indexed column in a transactional system. The column being indexed will be updated from N to Y, and many people are inserting into or updating this table concurrently. Those attributes preclude a bitmap index from even being considered. Bitmap indexes cannot be used in environments where the indexed value is modified frequently or the underlying table is concurrently modified by many sessions. So, some other solution must be found.

Here's where the FBI comes in handy. We have a requirement to index a subset of rows that satisfies some criteria. We do not want to index the remaining rows. We can index functions, and we know that entirely NULL entries are not made in B*Tree indexes. We can achieve our goal easily if we create an index like this:

```
Create index selective_index on table_name(
     Case when <some criteria is met> then 'Y'
     Else NULL
     End )
```

Let's walk through an example. We'll start by creating a table with many rows, all with a PROCESSED_FLAG column set to Y:

```
ops$tkyte@ORA920> create table t as
  2  select 'Y' processed_flag, a.* from all_objects a;
Table created.
```

Now, for programming simplicity, we'll create a view that will hide the complexity of our indexed column.

```
ops$tkyte@ORA920> create or replace view v
  2  as
  3  select t.*,
  4         case when processed_flag = 'N' then 'N'
  5                   else NULL
  6          end processed_flag_indexed
  7    from t;
View created.
```

This approach works well with most FBIs. Rather than have the developers or end users need to know the exact function to use, they just use a column in a view; that column has the function in it. Also, if the logic ever changes, you can simply re-create the index and update the view, and all the applications will be fixed immediately.

Now, we create the FBI using the same function:

```
ops$tkyte@ORA920> create index t_idx on
  2  t( case when processed_flag = 'N' then 'N'
  3          else NULL
  4      end );
Index created.
```

NOTE
If you cannot (or do not want to) use CASE, DECODE works for most purposes. For example, instead of CASE, we could have indexed `decode( processed_flag, 'N', 'N', NULL )` *instead.*

We'll use the ANALYZE INDEX VALIDATE STRUCTURE command to see how many rows are currently in the index:

```
ops$tkyte@ORA920> analyze index t_idx validate structure;
Index analyzed.

ops$tkyte@ORA920> select name, del_lf_rows, lf_rows, lf_blks
  2    from index_stats;

NAME                           DEL_LF_ROWS    LF_ROWS    LF_BLKS
------------------------------ ----------- ---------- ----------
T_IDX                                    0          0          1
```

As you can see, it starts with nothing in the index: zero leaf rows (LF_ROWS) and no delete leaf row (DEL_LF_ROWS) entries.

Let's see what happens after we do an update operation, setting 100 rows to N:

```
ops$tkyte@ORA920> update t set processed_flag = 'N'
  2    where rownum <= 100;
100 rows updated.

ops$tkyte@ORA920> analyze index t_idx validate structure;
Index analyzed.

ops$tkyte@ORA920> select name, del_lf_rows, lf_rows, lf_blks
  2    from index_stats;

NAME                           DEL_LF_ROWS    LF_ROWS    LF_BLKS
------------------------------ ----------- ---------- ----------
T_IDX                                    0        100          1
```

We can see the index has 100 entries. We know that the table has thousands of rows, but our index is very small and compact.

Now, since we must be using the cost-based optimizer (CBO) in order to use FBIs (and since we should be using the CBO for all things anyway), we'll analyze our structures.

```
ops$tkyte@ORA920> analyze table t compute statistics
  2  for table
  3  for all indexes
  4  for all indexed columns
  5  /
Table analyzed.
```

Then we'll simulate some processing to access the first unprocessed record, process it, and then update its flag. We'll do this two times, using AUTOTRACE the second time to see how efficient this is.

```
ops$tkyte@ORA920> column rowid new_val r

ops$tkyte@ORA920> select rowid, object_name
  2    from v
  3   where processed_flag_indexed = 'N'
  4     and rownum = 1;

ROWID              OBJECT_NAME
------------------ ------------------------------
AAAKlgAAJAAAAI8AAA /1005bd30_LnkdConstant

ops$tkyte@ORA920> update v
  2      set processed_flag = 'Y'
  3    where rowid = '&R';
old   3:   where rowid = '&R'
new   3:   where rowid = 'AAAKlgAAJAAAAI8AAA'
1 row updated.

ops$tkyte@ORA920> set autotrace on
ops$tkyte@ORA920> select rowid, object_name
  2    from v
  3   where processed_flag_indexed = 'N'
  4     and rownum = 1;

ROWID              OBJECT_NAME
------------------ ------------------------------
AAAKlgAAJAAAAI8AAB /10076b23_OraCustomDatumClosur


Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=2700)
   1    0   COUNT (STOPKEY)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'T' (Cost=2 Card=100
Bytes=2700)
   3    2       INDEX (RANGE SCAN) OF 'T_IDX' (NON-UNIQUE) (Cost=1 Card=100)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          2  consistent gets
          0  physical reads
…
          1  rows processed
```

There are two consistent gets—two logical I/Os—one to read the very small index and one to read the table block. In reality, we would expect between two and three I/Os per query execution, depending on the number of index entries made over time.

Now, we'll just update this row, set it to processed, and check our index:

```
ops$tkyte@ORA920> update v
  2      set processed_flag = 'Y'
  3    where rowid = '&R';
old   3:   where rowid = '&R'
new   3:   where rowid = 'AAAKlgAAJAAAAI8AAB'
1 row updated.

ops$tkyte@ORA920> analyze index t_idx validate structure;
Index analyzed.

ops$tkyte@ORA920> select name, del_lf_rows, lf_rows, lf_blks
  2    from index_stats;

NAME                           DEL_LF_ROWS    LF_ROWS    LF_BLKS
------------------------------ ----------- ---------- ----------
T_IDX                                    2        100          1
```

We can see that there are still 100 entries in there; however, two of them are deleted, ready to be reused. Along comes another session and inserts some unprocessed data:

```
ops$tkyte@ORA920> insert into t
  2  select 'N' processed_flag, a.* from all_objects a
  3  where rownum <= 2;
2 rows created.

ops$tkyte@ORA920> analyze index t_idx validate structure;
Index analyzed.

ops$tkyte@ORA920> select name, del_lf_rows, lf_rows, lf_blks
  2    from index_stats;

NAME                           DEL_LF_ROWS    LF_ROWS    LF_BLKS
------------------------------ ----------- ---------- ----------
T_IDX                                    0        100          1
```

And the whole cycle begins again. Over time, this index will remain more or less at a steady state, with some deleted entries that will be reused by subsequent inserts. The index will stay small and will index only those rows that match our predicate encoded in the CASE statement (which can be quite complex).

By putting together two facts in a manner you will not see documented in the Oracle documentation set, we've solved a seemingly complex problem easily.

Implement Selective Uniqueness

Another challenging problem is how to achieve "selective uniqueness." When a column in a table has a certain value, how can you make sure that other columns in that table are unique for that subset of rows?

> "I have a table project (`project_ID number primary key, teamid number, job varchar2(100), status varchar2(8)`). STATUS may have the values ACTIVE, meaning it is currently active, and INACTIVE, meaning it is archived. I was told to enforce a unique rule: the job has to be unique in the same TEAMID for the active projects. It means TEAMID and JOB have to be unique while STATUS=ACTIVE. What is the best way to do this?"
>
> The question went on to suggest that a trigger of some sort or some procedural code might be the solution. Well, I had just used an FBI recently to solve something slightly different. Here, he was not trying to index a subset of rows. In fact, he did not want to index at all, but rather enforce a unique condition selectively: "This data must be unique when this condition is met." Okay, let's look at some useful facts:
>
> - You can selectively index rows.
> - Indexes can be unique and hence can be used to enforce uniqueness.
> - Attempting to perform referential integrity on your own in a multiversioning, read-consistent database is a recipe for disaster.
>
> Putting it together, all you need to do to solve this problem is uniquely index TEAMID, JOB *when* STATUS=ACTIVE.
>
> An index to enforce a unique condition selectively might look something like this:

```
ops$tkyte@ORA920> create table project
  (project_ID number primary key,
   teamid number,
   job varchar2(100),
   status varchar2(20) check (status in ('ACTIVE', 'INACTIVE'))
  );
Table created.

ops$tkyte@ORA920> create UNIQUE index
  job_unique_in_teamid on project
  ( case when status = 'ACTIVE' then teamid else null end,
    case when status = 'ACTIVE' then job    else null end
  )
  /
Index created.
```

Now, when we try to insert a duplicate TEAMID/JOB pair in an active set of projects, we receive this error:

```
ops$tkyte@ORA920> insert into project(project_id,teamid,job,status)
  2  values( 1, 10, 'a', 'ACTIVE' );
1 row created.

ops$tkyte@ORA920> insert into project(project_id,teamid,job,status)
  2  values( 2, 10, 'a', 'ACTIVE' );
insert into project(project_id,teamid,job,status)
*
ERROR at line 1:
ORA-00001: unique constraint (OPS$TKYTE.JOB_UNIQUE_IN_TEAMID) violated
```

As soon as we retire (make inactive) that first row, the insertion of that second row goes through as expected:

```
ops$tkyte@ORA920> update project
  2     set status = 'INACTIVE'
  3    where project_id = 1
  4      and teamid = 10
  5      and status = 'ACTIVE';
1 row updated.

ops$tkyte@ORA920> insert into project(project_id,teamid,job,status)
  2  values( 2, 10, 'a', 'ACTIVE' );
1 row created.
```

We also have the nice effect that the index is only on active projects, so it is consuming as little space as possible. This implementation was possible only by thinking "outside the box" a bit.

NOTE
If at all possible, uniqueness should be enforced via a UNIQUE constraint. Unique constraints can be used by the optimizer during query rewrites, whereas a simple unique index won't be used. Unique constraints add metadata to the data dictionary that may be used by many tools as well. However, due to the fact that a UNIQUE constraint will not accept a function like the one in this section, a unique index was our only course of action.

Use Domain Indexes

Application domain indexes are what Oracle calls *extensible indexing*. It is how you or I could create a custom index structure that works just like an index supplied by Oracle. When someone issues a CREATE INDEX statement using your index type, Oracle will run your code to generate the index. If someone analyzes the index to compute statistics on it, Oracle will execute your code to generate statistics in whatever format you care to store them in. When Oracle parses a query and develops a query plan that may make use of your index, Oracle will ask you how costly this function is to perform as it is evaluating the different plans. Application domain indexes, in short, give you the ability to implement a new index type that does not exist in the database yet.

I personally have not often found the need to build a new exotic type of index structure. I see this particular feature as being of use mostly to third-party solution providers that have innovative indexing techniques. However, you can do some interesting things, and we'll also take a quick look at how you can create indexes.

Domain Indexes Can Implement Third-Party Indexing Techniques

Domain indexes allow some third-party company to create an index to use with your system. For example, if you inserted a fingerprint into the database via a BLOB type, some external software would be invoked to index this fingerprint data. It would store the point data related to the fingerprint in database tables, in clusters, or perhaps externally in flat files—wherever made the most sense. You would now be able to input a fingerprint into the database and find other fingerprints that matched it, just as easily as you `select * from t where x between 1 and 2` using SQL.

Let's consider an example. Say you developed software that analyzed images stored in the database and produced information about the images, such as the colors found in them. You could use a domain index to create your own image index. As images were added to the database, your code would be invoked to extract the colors from the images and store them wherever you wanted to keep them. At query time, when the user asked for all blue images, Oracle would ask you to provide the answer from your index when appropriate.

The best example of this is Oracle's own text index. This index is used to provide keyword searching on large text items. Oracle Text introduces its own index type:

```
ops$tkyte@ORA8I.WORLD> create index myindex on mytable(docs)
2  indextype is ctxsys.context
3  /
Index created.
```

It also uses its own operators in the SQL language:

```
select * from mytable where contains( docs, 'some words' ) > 0;
```

It will even respond to commands such as the following:

```
ops$tkyte@ORA8I.WORLD> analyze index myindex compute statistics;
Index analyzed.
```

Oracle will participate with the optimizer at runtime to determine the relative cost of using a text index over some other index or a full scan. The interesting thing about all of this is that you or I could have developed this index. The implementation of the text index was done without inside kernel knowledge. It was implemented using the documented and exposed API for doing these sorts of things.

The Oracle database kernel is not aware of how the text index is stored (it's stored in many physical database tables per index created). Oracle is not aware of the processing that takes place when a new row is inserted. Oracle Text is really an application built on top of the database, but in a wholly integrated fashion. To you and me, it looks just like any other Oracle database kernel function, but it is not.

Create Indexes

Knowing what is available as far as domain indexes go can prove very useful. If you need specialized information, it's possible that the database already stores that for you.

For example, Oracle has the ability to natively index point data—latitudes and longitudes in a simple sense—and provide spatial searches based on them, to be able to order data by them. It is a nontrivial problem in a relational database when you get down to it. Consider that latitude and longitude are two attributes. You want to sort all of the data in the database by distance from a given point. Well, you cannot sort by latitude and then longitude, nor can you do the opposite. You really need to sort by some derived, complex value. This is where spatial data comes into play. You can use spatial information to find all of the rows that are close to some point, sort them, and find the distance between the points represented by a pair of rows and so on.

The point I'm making here is not how to use spatial data. For that, you must read the documentation and spend a little while with the capability. Rather, my point is that the database contains many features and functions that are used to store and retrieve data quickly and efficiently. If you are aware of this functionality, you can use it.

> "I have a function F_DIST written in PL/SQL to compute the distance between two points. When I call this from SQL in my query, I've discovered that it takes over one-half hour to execute. Now, if I "inline" the function—I do not use the PL/SQL call, but rather do the function inline using acos/sin and so in—it runs in about 15 seconds or so. My questions are 1) Why is the PL/SQL so much slower? and 2) Is there a way to speed this up more?"
>
> This is an interesting question for two reasons. First, why did the PL/SQL function-based approach take so long, and second, why were they reinventing the wheel? After a little research (I got a hold of their data to play with), what we discovered was that with their data set, there were 612,296 invocations of the PL/SQL function F_DIST. Now, it took about 2,083 seconds to run this query on my system (about the same as theirs), meaning that we called that PL/SQL function 294 times a second. That is a context switch from the SQL engine to the PL/SQL engine 294 times every second. Each callout was taking at most 0.003 second, which by itself is very small but adds up when you do it 612,296 times! Anytime we can "inline" the code directly in SQL, we should, because it will almost certainly be faster than calling out to PL/SQL time and time again.
>
> Now for the second part, reinventing the wheel and making it go faster. I don't like doing the first item, and I really enjoy it when the second item is easy to achieve. In this case, not only can we make it go faster, we can do it much easier. My solution was to augment the tables to have a spatial column, and then use that in the query. Here is the code we needed to add to their schema:

```
alter table b add
(geometry mdsys.sdo_geometry);

update b a
   set a.geometry =
    mdsys.sdo_geometry(2001,null,
         mdsys.sdo_point_type(a.longitude,
                              a.latitude,
                              null),
         null, null)
  where latitude is not null;

insert into user_sdo_geom_metadata
values ('B','GEOMETRY',
   mdsys.sdo_dim_array(
      mdsys.sdo_dim_element
      ('X',-180,180,.00005),
      mdsys.sdo_dim_element
      ('Y',-90,90,.00005)), null);

create index b_sidx on b(geometry)
indextype is mdsys.spatial_index;
```

That is for table B in their example. For table C, we added a GEOMETRY column, registered it with the USER_SDO_GEOM_METADATA table, and then created a spatial index on it as well. Now, their query becomes simply:

```
select substr(Z2.ZIPP, 1, 7) ZIP
  from A PT, B TA, C Z2
 where ( PT.REGION || PT.AREA ||
         PT.DISTRICT || PT.LOCATION) = :str
   and PT.LOCATION_ID = TA.LOCATION_ID
   AND MDSYS.LOCATOR_WITHIN_DISTANCE
     (Z2.AVG_GEOM, TA.GEOMETRY,
      'distance = '||:dist||', units=mile') = 'TRUE';
```

We found that same query runs in less than one second, versus more than 15 seconds using straight SQL and one-half hour using PL/SQL called from SQL.

If I had a dollar for every time someone “invented” a text-indexing technique for indexing documents in the database, I would be rich. If I had the money spent by their employers funding the development, redevelopment, maintenance, bug fixing, and so on for these custom-developed techniques, I would be perhaps the richest person alive.

What happens when these folks are shown that the functionality they painstakingly coded to parse and keyword index documents already existed, was fully integrated with the optimizer, had administrative ease of use (such as allowing rebuilding), and had many of the native database features like partitioning and parallel processing? They are sometimes filled with amazement, other times dismay, and many times, disbelief. In all cases, using native functionality would have greatly improved development productivity, maintenance, and performance.

Compression

We’ve discussed ways in which you can have Oracle physically organize your data for you—using B*Tree clusters, hash clusters, IOTs, and the like. The last storage-related option we’ll look at is data compression. Oracle supports compression of both index structures (since Oracle8i Release 1) as well as database tables (since Oracle9i Release 2). The compression algorithm for both segment types is similar in that it relies on factoring out repetitive information and storing that repeating data once on a block, instead of once per occurrence on the block.

The compression of indexes and tables is a little different and can be used in different situations. For example, compressed indexes can be used on tables that undergo constant modifications, but table compression would not make sense in that case due to its implementation. We'll discuss index key compression and then table compression.

Use Index Key Compression

Index key compression in Oracle allows you to factor out the repetitive leading edge of an index key and store the leading-edge values once per leaf block instead of once per row per leaf block.

Consider the database view DBA_OBJECTS. Suppose we copy that table and create an index on OWNER, OBJECT_TYPE, and OBJECT_NAME, to be able to quickly retrieve rows for a specific object owned by someone. Normally, the index would store values like this:

User1,table,t1	User1,table,t2	User1,table,t3	User1,table,t4	User1,table,t5
User1,table,t6	User1,table,t7	User1,index,i1	User1,index,i2	… and so on…

Using index key compression, Oracle could instead store the values conceptually, like this:

User1,table	T1	T2	T3	T4
T5	T6	User1,index	I1	I2…

In effect, Oracle could store significantly more data per index leaf block than an uncompressed index could store.

Before looking at an example, let's go over a couple of facts about compressed indexes. Here are some of their advantages:

- A compressed index consumes less disk space, so it provides some storage savings.
- A compressed index can reduce the amount of physical I/O your system performs.
- A compressed index increases the buffer cache efficiency. There are quite simply fewer blocks to cache. The index blocks are cached compressed as well as stored compressed.

And here are some of the disadvantages of compressed indexes:

- The compressed index puts more row entries per block, increasing contention on these already compact data structures even more. If before you were putting 200 rows per leaf block—200 possible rows people might update in concurrent sessions—you are now putting in about 400 rows.
- A compressed index requires slightly more CPU time to process at runtime because the structure is more complex. You may notice this during both insert and select operations.

Bear in mind these are simply considerations. It does not mean that there will be more contention. It does not mean that you will even notice the additional CPU time, if any, being used. They are things to watch out for and to test against as you implement this feature.

Let's take a look at this in action. We'll create the compressed and uncompressed indexes on this data and compare not only their size but their performance as well.

We start by creating a copy of DBA_OBJECTS, which we will quadruple in volume. That will mean that our index on (OWNER, OBJECT_TYPE, OBJECT_NAME) will point to about four rows per key value typically.

```
ops$tkyte@ORA920> create table t1
  2  as
  3  select * from dba_objects;
Table created.

ops$tkyte@ORA920> insert /*+ append */ into t1 select * from t1;
30975 rows created.

ops$tkyte@ORA920> commit;
Commit complete.

ops$tkyte@ORA920> insert /*+ append */ into t1 select * from t1;
61950 rows created.

ops$tkyte@ORA920> commit;
Commit complete.

ops$tkyte@ORA920> create index uncompressed_idx
  2  on t1( owner,object_type,object_name );
Index created.

ops$tkyte@ORA920> analyze table t1 compute statistics
  2  for table
  3  for all indexes
  4  for all indexed columns;
Table analyzed.
```

Now, since we cannot create another index on these same columns in this table, we'll copy the table again and index the same data, this time using index key compression on all three key values:

```
ops$tkyte@ORA920> create table t2
  2  as
  3  select * from t1;
Table created.

ops$tkyte@ORA920> create index compressed_idx
  2  on t2( owner,object_type,object_name )
3  COMPRESS 3;
Index created.
```

```
ops$tkyte@ORA920> analyze table t2 compute statistics
  2  for table
  3  for all indexes
  4  for all indexed columns;
Table analyzed.
```

We are ready to measure the space differences between these two indexes. For this, we'll use the INDEX_STATS dynamic view. This view is populated with details about an index structure after an ANALYZE INDEX VALIDATE STRUCTURE statement. This view has, at most, one row at a time, so we'll copy the rows out as we validate the indexes.

```
ops$tkyte@ORA920> analyze index uncompressed_idx validate structure;
Index analyzed.

ops$tkyte@ORA920> create table index_stats_copy as select * from index_stats;
Table created.

ops$tkyte@ORA920> analyze index compressed_idx validate structure;
Index analyzed.

ops$tkyte@ORA920> insert into index_stats_copy select * from index_stats;
1 row created.
```

Now, we are ready to compare them. For a side-by-side comparison, we can use the following SQL:

```
select 'HEIGHT',
       max(decode(name,'UNCOMPRESSED_IDX',HEIGHT,null)),
       max(decode(name,'UNCOMPRESSED_IDX',to_number(null),HEIGHT))
  from index_stats_copy
 union all
select 'BLOCKS',
       max(decode(name,'UNCOMPRESSED_IDX',BLOCKS,null)),
       max(decode(name,'UNCOMPRESSED_IDX',to_number(null),BLOCKS))
  from index_stats_copy
 union all
…(query for each column)…
```

But, if you don't feel like typing that in, this bit of code generates it for you:

```
ops$tkyte@ORA920> variable x refcursor
ops$tkyte@ORA920> declare
  2      l_stmt long;
  3  begin
  4      for x in ( select '''' || column_name || '''' quoted,
  5                        column_name
  6                   from user_tab_columns
  7                  where table_name = 'INDEX_STATS_COPY'
  8                    and column_name not in
  9                        ('NAME','PARTITION_NAME') )
```

```
    loop
        l_stmt := l_stmt || ' select ' || x.quoted || ' name,
                    max(decode(name,''UNCOMPRESSED_IDX'',' ||
                    x.column_name || ',null)) uncompressed,
                    max(decode(name,''UNCOMPRESSED_IDX'',
                        to_number(null),' || x.column_name ||
                        ')) compressed
                    from index_stats_copy union all';
    end loop;
    l_stmt :=
    'select name, uncompressed, compressed,
            uncompressed-compressed diff,
            decode(uncompressed,0,
               to_number(null),
               round(compressed/uncompressed*100,2)) pct
       from ( ' ||
         substr( l_stmt, 1,
                 length(l_stmt)-length(' union all') ) ||
            ') order by name';
    open :x for l_stmt;
end;
/

PL/SQL procedure successfully completed.

ops$tkyte@ORA920> print x

NAME                 UNCOMPRESSED COMPRESSED       DIFF        PCT
-------------------- ------------ ---------- ---------- ----------
BLKS_GETS_PER_ACCESS   5.50934125 5.50934125          0        100
BLOCKS                        896        512        384      57.14
BR_BLKS                         6          3          3         50
BR_BLK_LEN                   8028       8028          0        100
BR_ROWS                       841        379        462      45.07
BR_ROWS_LEN                 37030      14367      22663       38.8
BTREE_SPACE               6780800    3061044    3719756      45.14
DEL_LF_ROWS                     0          0          0
DEL_LF_ROWS_LEN                 0          0          0
DISTINCT_KEYS               30831      30831          0        100
HEIGHT                          3          3          0        100
LF_BLKS                       842        380        462      45.13
LF_BLK_LEN                   7996       7992          4      99.95
LF_ROWS                    123900     123900          0        100
LF_ROWS_LEN               6021780    1362900    4658880      22.63
MOST_REPEATED_KEY              64         64          0        100
OPT_CMPR_COUNT                  3          3          0        100
OPT_CMPR_PCTSAVE               54          0         54          0
PCT_USED                       90         90          0        100
PRE_ROWS                        0      30972     -30972
PRE_ROWS_LEN                    0    1351112   -1351112
```

```
ROWS_PER_KEY            4.01868249 4.01868249          0        100
USED_SPACE                 6058810    2728379    3330431      45.03

23 rows selected.
```

Now we have a side-by-side comparison of the uncompressed index and compressed index. First, observe that we achieved better than a 50% compression ratio. Ignore the BLOCKS value for a moment (that is space allocated to the segment but includes unused blocks as well). Looking at the BR_BLKS (branch blocks) and LF_BLKS (leaf blocks, where the actual index keys and ROWIDs are stored), we see that we have 50% of the branch blocks and 45% of the leaf blocks. The compressed index is consuming about half of the disk space. Another way to look at this is that every block in the compressed index contains twice the amount of data. Not only do we save disk space, but we also just doubled the size of our buffer cache. Also consider that Oracle reads blocks not rows; hence, anything that reduces the number of blocks Oracle must manage potentially "increases" the capacity of our buffer cache.

The other interesting thing to notice here is the OPT_CMPR_COUNT (optimal key compression length) and OPT_CMPR_PCTSAVE (percent savings in space if you compress) values. They estimated that if we used index key compression on the uncompressed index, we would save 54% in space—a value that is accurate, as shown by the actual compressed index. You can use these values to see if rebuilding an index with compression would be paid back with significant space savings.

The last items we'll consider here are the PRE_ROWS (prefix rows, or actual key values that were factored out) and the PRE_ROWS_LEN (length of these prefix rows) values. These numbers represent the number of unique prefix keys in the index. These represent the data that is factored out of the index structure and stored once per leaf block instead of once per row. Here, we find a value of 30,972. This is not surprising, since the number of rows in my DBA_OBJECTS table was 30,975 (you can see that by the first INSERT /*+ APPEND */ output shown earlier—the first doubling of the table added that many rows). That is just showing that apparently there are 30,972 unique (OWNER, OBJECT_TYPE, OBJECT_NAME) combinations in the table. The PRE_ROWS_LEN value shows these factored-out values take about 1.3MB of storage.

For the raw numbers, we can run a somewhat simple test that takes each key value and reads the entire row using that key. Bear in mind that each key value is in the index about four times, so we are actually fetching each key value out about four times, for 16 rows.

```
ops$tkyte@ORA920> alter session set sql_trace=true;
Session altered.

ops$tkyte@ORA920> begin
         for x in ( select * from t1 )
         loop
             for y in ( select *
                          from t1
                         where owner = x.owner
                           and object_name = x.object_name
                           and object_type = x.object_type )
             loop
                 null;
             end loop;
         end loop;
         for x in ( select * from t2 )
         loop
             for y in ( select *
                          from t2
                         where owner = x.owner
                           and object_name = x.object_name
                           and object_type = x.object_type )
             loop
                 null;
             end loop;
         end loop;
     end;
     /
PL/SQL procedure successfully completed.
```

The TKPROF report for this run shows the following statistics:

```
select * from t1
where owner = :b3 and object_name = :b2 and object_type = :b1

call     count       cpu elapsed disk   query current     rows
------- ------  -------- ------- ---- ------- -------  -------
Parse        1      0.00    0.00    0       0       0        0
Execute 123900     16.54   17.42    0       0       0        0
Fetch   650828     39.69   41.09    0 1425552       0   526928
------- ------  -------- ------- ---- ------- -------  -------
total   774729     56.24   58.52    0 1425552       0   526928

Rows     Row Source Operation
-------  ---------------------------------------------------
 526928  TABLE ACCESS BY INDEX ROWID T1
 526928   INDEX RANGE SCAN UNCOMPRESSED_IDX (object id 43577)
**************************************************************
select * from t2
where owner = :b3 and object_name = :b2 and object_type = :b1

call     count    cpu elapsed disk   query current     rows
------- ------  ----- ------- ---- ------- -------  -------
Parse        1   0.00    0.00    0       0       0        0
Execute 123900  17.48   19.98    0       0       0        0
Fetch   650828  42.29   43.34    0 1425552       0   526928
------- ------  ----- ------- ---- ------- -------  -------
total   774729  59.77   63.33    0 1425552       0   526928

Rows     Row Source Operation
-------  ---------------------------------------------------
 526928  TABLE ACCESS BY INDEX ROWID T2
 526928   INDEX RANGE SCAN COMPRESSED_IDX (object id 43579)
```

As you can see, the CPU times are more or less the same (yes, they vary by about 3 CPU seconds, but that is only a 5% variation, so they are more or less equivalent). If we were to run this again, the numbers may well be reversed. This is due to the fact that the 39.69 CPU seconds reported by the fetch phase for the first query is an aggregation of 650,828 discrete timed events. If some of the events took less time to execute than the granularity of the system clock, they would report 0; others may report 1 unit of time. (In fact, I did another couple of runs just to test that this is reproducible, and in one occurrence, I observed CPU times of 60.06 and 60.97 CPU seconds for the uncompressed versus compressed timings.)

The TKPROF report also shows that the amount of logical I/O performed by each was the same. This is not surprising, given that the height of both indexes was the same (as observed by the preceding INDEX_STATS output). Therefore, given any key entry, it would logically take the same amount of logical I/Os to get the leaf entry. In this case, it took two logical I/Os on average to get the leaf block and one more logical I/O to retrieve the table data itself.

I also ran these tests using the Runstats framework. The salient output showed that the latching differences between the two approaches was negligible. In the following, RUN1 is the uncompressed index and RUN2 is the compressed index attempt:

```
      RUN1       RUN2       DIFF RUN1_PCT_OF_RUN2
---------- ---------- ---------- ----------------
   3622897    3630132       7235             99.8
```

Additionally, the wall clock times were not significantly different either:

```
12058 hsecs
12467 hsecs
run 1 ran in 96.72% of the time
```

The uncompressed index was only 3% faster by the wall clock—an amount you most likely would not notice in the real world. Remember also that the wall clock time is easily influenced by external events. Our computers do more than one thing at a time, after all.

Use Table Compression for Read-Only/Read-Mostly Tables

The ability to compress a table started in Oracle8i with IOTs. Since an IOT is really a special kind of index—one that supports index key compression—we could compress it using that technique.

True table-level compression for more general table types is new with Oracle9i Release 2. It is similar to index key compression in that it factors out repetitive information, but it differs in many ways. Table compression works on the aggregate-block level, not row by row as index key compression does; that is, when Oracle is building a database block, it is looking for repeating values across all columns and rows on that block. The values need not be on the leading edge of the row; they can be anywhere in the row.

The major difference between table and index compression is when each may be used. Index key compression works equally well on systems where the table is being modified frequently and on read-only or read-mostly systems. The index is maintained in its compressed state in both environments. However, table compression works well *only* in a read-only or read-mostly environment. It is not a feature you will use on your active transaction tables (although you can use it in your transactional systems, as you'll see). Table compression works only for bulk operations as well. A normal INSERT or UPDATE statement will not compress data. Instead, you need to use one of the following:

- CREATE TABLE AS SELECT
- INSERT /*+ APPEND */ (direct-path insert)
- SQLLDR direct=y (direct-path load)
- ALTER TABLE MOVE

Compressing a table does not prohibit you from using normal DML against it. It is just that the newly added or modified information will not be stored compressed. It would take a rebuild of that segment in order to compress it. So, table compression is most likely to be valuable in these situations:

- Large amounts of static reference information that is read-only or read-mostly
- Data warehousing environments where bulk operations are common
- Audit trail information stored in partitioned tables, where you can compress last month's auditing information at the beginning of a new month

Table compression should not be considered for most transactional tables where the data is heavily updated. There table compression would be defeated, as each update would tend to "decompress" the row(s).

Table compression in Oracle is achieved by factoring out repeating data found on a database block and creating a symbol table. It we started with a database block that looked like this (each cell in the table represents a row, the data is stored delimited in this conceptual depiction):

SCOTT,TABLE,EMP	SCOTT,TABLE,DEPT	SCOTT,TABLE,BONUS
SCOTT,INDEX,EMP_PK	SCOTT,INDEX,ENAME_IDX	SCOTT,INDEX,DEPT_PK
SCOTT,INDEX,EMP_DEPT	SCOTT,INDEX,DNAME_IDX	SCOTT,PROCEDURE,P1
SCOTT,PROCEDURE,P2	SCOTT,PROCEDURE,P3	SCOTT,PROCEDURE,P4

A compressed block would conceptually look like this:

SCOTT=<A>,TABLE=<B>,
INDEX=<C>,PROCEDURE=<D>

<A>,<B>,EMP	<A>,<B>,DEPT	<A>,<B>,BONUS	<A>,<C>,EMP_PK
<A>,<C>,ENAME_IDX	<A>,<C>,DEPT_PK	<A>,<C>,EMP_DEPT	<A>,<C>,DNAME_IDX
<A>,<D>,P1	<A>,<D>,P2	<A>,<D>,P3	<A>,<D>,P4

Here, the repeating values of SCOTT, TABLE, INDEX, and PROCEDURE were factored out, stored once on the block in a symbol table, and replaced in the block with simple indexes into the symbol table. This can result in significant savings in space.

Consider this simple example where we store DBA_OBJECTS in an uncompressed and a compressed table. Note that the PCTFREE on the compressed table will be 0, so we'll use the same PCTFREE on the uncompressed table to be fair.

```
ops$tkyte@ORA920> create table uncompressed
2  pctfree 0
3  as
  4  select *
  5    from dba_objects
  6   order by owner, object_type, object_name;
Table created.

ops$tkyte@ORA920> analyze table uncompressed
  2  compute statistics
  3  for table;
Table analyzed.

ops$tkyte@ORA920> create table compressed
2  COMPRESS
3  as
  4  select *
  5    from uncompressed
  6   order by owner, object_type, object_name;
Table created.

ops$tkyte@ORA920> analyze table compressed
  2  compute statistics
  3  for table;
Table analyzed.

ops$tkyte@ORA920> select cblks comp_blks, uncblks uncomp_blks,
  2         round(cblks/uncblks*100,2) pct
  3    from (
  4  select max(decode(table_name,'COMPRESSED',blocks,null)) cblks,
  5    max(decode(table_name,'UNCOMPRESSED',blocks,null)) uncblks
  6    from user_tables
  7   where table_name in ( 'COMPRESSED', 'UNCOMPRESSED' )
  8         )
  9  /

 COMP_BLKS UNCOMP_BLKS        PCT
---------- ----------- ----------
       217         395      54.94
```

This shows that we can store the same amount of information in 55% of the space. That is significant—we just cut our disk space needs in half for this table.

Notice that we put an ORDER BY on the CREATE TABLE COMPRESSED and ordered the table by data we knew to be compressible. We clumped together all of the SCOTT objects by type on the same blocks. This allows the compression routines to do their best job. Instead of storing SCOTT (or SYSTEM, or CTXSYS, or whatever) dozens or hundreds of times per block, we store it once. We also store TABLE and PACKAGE BODY once, and then point to them with a very small pointer.

You might think that is cheating; the data won't be sorted. I disagree. Table compression is useful only on read-only/read-mostly data. You have a great deal of control over the sorted order of that data. You load it into a warehouse, have an audit trail, or have reference information. All of this can be sorted. If your goal is to minimize disk use, you can achieve that.

Just for fun, let's see what happens if the data is somewhat randomly sorted:

```
ops$tkyte@ORA920> create table compressed
2  COMPRESS
3  as
  4  select *
  5    from uncompressed
6   order by dbms_random.random;
Table created.

ops$tkyte@ORA920> analyze table compressed
  2  compute statistics
  3  for table;
Table analyzed.

ops$tkyte@ORA920> select cblks comp_blks, uncblks uncomp_blks,
  2         round(cblks/uncblks*100,2) pct
  3    from (
  4  select max(decode(table_name,'COMPRESSED',blocks,null)) cblks,
  5    max(decode(table_name,'UNCOMPRESSED',blocks,null)) uncblks
  6    from user_tables
  7   where table_name in ( 'COMPRESSED', 'UNCOMPRESSED' )
  8         )
  9  /

 COMP_BLKS UNCOMP_BLKS        PCT
---------- ----------- ----------
       287         395      72.66
```

Even if the data in this particular table arrives randomly, we achieve about a 25% reduction in space needed. Your mileage may vary on this; I observed numbers between 65% and 80% during repeated runs. The results depended on how the data ended up. Using the statistics in the USER_TAB_COLUMNS table for the UNCOMPRESSED table, I noticed that a particularly wide column TIMESTAMP had a lot of repetitive values and no NULL values:

```
ops$tkyte@ORA920> analyze table uncompressed compute statistics;
Table analyzed.

ops$tkyte@ORA920> select column_name, num_distinct, num_nulls, avg_col_len
  2  from user_tab_columns
  3  where table_name = 'UNCOMPRESSED'
  4  /
```

```
COLUMN_NAME     NUM_DISTINCT  NUM_NULLS AVG_COL_LEN
--------------- ------------ ---------- -----------
OWNER                     37          0           5
OBJECT_NAME            19194          0          23
SUBOBJECT_NAME            46      30774           2
OBJECT_ID              30980          0           4
DATA_OBJECT_ID          2340      28603           2
OBJECT_TYPE               33          0           8
CREATED                 2143          0           7
LAST_DDL_TIME           2262          0           7
TIMESTAMP               2184          0          19
STATUS                     2          0           6
TEMPORARY                  2          0           1
GENERATED                  2          0           1
SECONDARY                  1          0           1

13 rows selected.
```

I rebuilt the compressed table sorting by TIMESTAMP this time, and the results were impressive:

```
ops$tkyte@ORA920> drop table compressed;
Table dropped.

ops$tkyte@ORA920> create table compressed
  2  COMPRESS
  3  as
  4  select *
  5    from uncompressed
  6   order by timestamp;
Table created.

ops$tkyte@ORA920> analyze table compressed
  2  compute statistics
  3  for table;
Table analyzed.

ops$tkyte@ORA920>
ops$tkyte@ORA920> select cblks comp_blks, uncblks uncomp_blks,
  2         round(cblks/uncblks*100,2) pct
  3    from (
  4  select max(decode(table_name,'COMPRESSED',blocks,null)) cblks,
  5    max(decode(table_name,'UNCOMPRESSED',blocks,null)) uncblks
  6    from user_tables
  7   where table_name in ( 'COMPRESSED', 'UNCOMPRESSED' )
  8         )
  9  /

 COMP_BLKS UNCOMP_BLKS        PCT
---------- ----------- ----------
       147         395      37.22
```

That table now takes one-third the space of the uncompressed version, due to factoring out that wide column and storing it once per block. This shows that a good understanding of the frequency of values in your data, the size of your data, and the ability to sort the data will allow you to achieve maximum compression, if that is your goal.

The examples in this section use different storage options to achieve different goals, and sometimes these goals are orthogonal to each other. For example, you might find that sorting by TIMESTAMP achieves maximum compression but you might also observe that people often query for all of the TABLES owned by some user. In that case, sorting by OWNER, OBJECT_TYPE would place the data people request in a query on the fewest number of blocks (all of the data for SCOTT's tables would tend to be collected on a small number of blocks, all colocated). So you have to make a trade-off here: Do you want to use as little of the disk as possible, but potentially spread SCOTT's objects over many blocks, or do you want to use a little more disk space, but have all of SCOTT's tables on the same blocks? There is no right or wrong answer. There is only the answer you come to after careful consideration of your goals and objectives—what you most want or need to have happen in your system.

Here are the three places where you might consider table compression:

- When you have large amounts of static reference information that is read-only or read-mostly
- In data warehousing environments where bulk operations are common
- For audit trail information stored in partitioned tables, where you can compress last month's auditing information at the beginning of a new month

Compressed Tables for Static Information and Data Warehousing

If you have large amounts of static reference information already loaded into your database, you can use compressed tables as follows:

- Analyze the table to find the best sort columns to achieve maximum compression (assuming that is your goal).
- Use CREATE TABLE COPY_OF_TABLE *compress* AS SELECT with the requisite ORDER BY.
- Drop the old uncompressed table and rename this copy.

It you are in a data warehouse situation and are bulk-loading data, you can follow the same basic procedure. For example, you can use DBMS_STATS against external tables. If you are unsure of the frequencies in the data you are loading, you can use the same approach for your database tables. Using the BIG_TABLE_EXTERNAL table we used in the examples in the "External Tables" section earlier in the chapter, we can do the following:

```
ops$tkyte@ORA920> exec dbms_stats.gather_table_stats -
> ( user, 'BIG_TABLE_EXTERNAL' );
PL/SQL procedure successfully completed.

ops$tkyte@ORA920> select column_name, num_distinct, num_nulls, avg_col_len
  2  from user_tab_columns
  3  where table_name = 'BIG_TABLE_EXTERNAL'
  /

COLUMN_NAME                     NUM_DISTINCT  NUM_NULLS AVG_COL_LEN
------------------------------ ------------ ---------- -----------
OWNER                                    28          0           6
OBJECT_NAME                           17130          0          24
SUBOBJECT_NAME                           31    1824766           2
OBJECT_ID                           1833792          0           6
DATA_OBJECT_ID                         1475    1737790           2
OBJECT_TYPE                              27          1           9
CREATED                                  13          0           8
LAST_DDL_TIME                            16          0           8
TIMESTAMP                           1833729         63           8
STATUS                                    2         63           6
TEMPORARY                                 2         63           2
GENERATED                                 2         63           2
SECONDARY                                 1         63           2

13 rows selected.
```

From that, we can deduce that maximum compression would be achieved by sorting on OBJECT_NAME. Because it has relatively few distinct values as compared to the number of rows in the table and the average row length is very wide, it would be the best candidate. We can load the table using this code:

```
ops$tkyte@ORA920> create table big_table_compressed
  COMPRESS
  as
  select * from big_table_external
  order by object_name;
Table created.
```

When I compared the sizes of a compressed versus uncompressed version of the same data, I found the compressed table used 4,831 blocks and the uncompressed version used 22,188—a 5-to-1 compression ratio!

Compress Auditing or Transaction History

Suppose that you have a sliding window of data in a partitioned table. For example, you are keeping seven years of audit information online in your database. Every month, you take the oldest month of data and drop it, you add a new partition to the table to accommodate new data, and now you want to also compress the latest full month's worth of data. We'll take a look at how you could accomplish this step by step. We'll start with our partitioned AUDIT_TRAIL_TABLE:

```
ops$tkyte@ORA920> CREATE TABLE audit_trail_table
  ( timestamp date,
    username  varchar2(30),
    action    varchar2(30),
    object    varchar2(30),
  6    message   varchar2(80)
  7  )
  8  PARTITION BY RANGE (timestamp)
  9  ( PARTITION jan_2002 VALUES LESS THAN
 10    ( to_date('01-feb-2002','dd-mon-yyyy') ) ,
 11    PARTITION feb_2002 VALUES LESS THAN
 12    ( to_date('01-mar-2002','dd-mon-yyyy') ) ,
 13    PARTITION mar_2002 VALUES LESS THAN
 14    ( to_date('01-apr-2002','dd-mon-yyyy') ) ,
 15    PARTITION apr_2002 VALUES LESS THAN
 16    ( to_date('01-may-2002','dd-mon-yyyy') ) ,
…
 37    PARTITION mar_2003 VALUES LESS THAN
 38    ( to_date('01-apr-2003','dd-mon-yyyy') ) ,
 39    PARTITION the_rest VALUES LESS THAN
 40    ( maxvalue )
 41  )
 42  /
Table created.

ops$tkyte@ORA920> create index partitioned_idx_local
  2  on audit_trail_table(username)
  3  LOCAL
  4  /
Index created.
```

This table has a partition for each month's worth of data, dating back to January 2002 and up through the end of March 2003. Now, assume it is the beginning of April and we have a month's worth of audit information stored uncompressed in the MAR_2003 partition. We would like to do the following:

- Get rid of the oldest data (slide that window of data to the right)
- Add a new partition just for April 2003
- Compress the data for March 2003, since we no longer will be adding to it (only querying it)

First, we'll generate some mock data for March 2003:

```
ops$tkyte@ORA920> insert into audit_trail_table
  2  select to_date( '01-mar-2003 ' ||
  3                  to_char(created,'hh24:mi:ss'),
  4                  'dd-mon-yyyy hh24:mi:ss' ) +
  5                  mod(rownum,31),
  6         owner,
  7         decode( mod(rownum,10), 0, 'INSERT',
  8               1, 'UPDATE', 2, 'DELETE', 'SELECT' ),
  9         object_name,
 10         object_name || ' ' || dbms_random.random
 11    from (select * from dba_objects
 12          UNION ALL
 13          select * from dba_objects
 14          UNION ALL
 15          select * from dba_objects
 16          UNION ALL
 17          select * from dba_objects)
 18  /
124072 rows created.

ops$tkyte@ORA920> analyze table audit_trail_table partition (mar_2003)
  2  compute statistics
  3  /
Table analyzed.

ops$tkyte@ORA920> select num_rows, blocks
  2    from user_tab_partitions
  3   where table_name = 'AUDIT_TRAIL_TABLE'
  4     and partition_name = 'MAR_2003'
  5  /

  NUM_ROWS     BLOCKS
---------- ----------
    124072       1440
```

Now, we'll use the column statistics to determine how we might want to sort this data to achieve our goal of maximum compression:

```
ops$tkyte@ORA920> select column_name, num_distinct, num_nulls, avg_col_len
  2    from USER_PART_COL_STATISTICS
  3   where table_name = 'AUDIT_TRAIL_TABLE'
  4     and partition_name = 'MAR_2003'
  5  /

COLUMN_NAM NUM_DISTINCT  NUM_NULLS AVG_COL_LEN
---------- ------------ ---------- -----------
TIMESTAMP         39763          0           7
USERNAME             37          0           5
ACTION                4          0           6
OBJECT            19199          0          23
MESSAGE          107394          0          34
```

We can see that MESSAGE is almost unique, but that OBJECT is not very selective and it is big. We decide to order by OBJECT and then USERNAME/ACTION based on this information. So, we'll create a table TEMP with the sorted, compressed data:

```
ops$tkyte@ORA920> drop table temp;
Table dropped.

ops$tkyte@ORA920> create table temp
```

```
  2  COMPRESS
  3  as
  4  select timestamp, username, action, object, message
  5    from audit_trail_table Partition(mar_2003)
  6   order by object, username, action
  7  /
Table created.
```

And we must index it, so it looks just like our partitioned table structurally:

```
ops$tkyte@ORA920> create index temp_idx on temp(username)
  2  /
Index created.

ops$tkyte@ORA920> analyze table temp
  2  compute statistics
  3  for table
  4  for all indexes
  5  for all indexed columns
  6  /
Table analyzed.
```

Now, we are ready to move data. We start by dropping the oldest partition, sliding the window to the right timewise.

```
ops$tkyte@ORA920> alter table audit_trail_table
  2  drop partition jan_2002;
Table altered.
```

Then we swap in our nicely compacted slice of data using the EXCHANGE PARTITION command.

```
ops$tkyte@ORA920> set timing on
ops$tkyte@ORA920> alter table audit_trail_table
  2  exchange partition mar_2003
  3  with table temp
  4  including indexes
  5  without validation
  6  /
Table altered.
Elapsed: 00:00:00.12
```

Note that this operation happens almost instantaneously. This is a simple DDL command that swaps names; the data isn't touched during this operation. Additionally, we can see the statistics are swapped as well:

```
ops$tkyte@ORA920> select blocks
  2    from user_tab_partitions
  3   where table_name = 'AUDIT_TRAIL_TABLE'
  4     and partition_name = 'MAR_2003'
  5  /

    BLOCKS
----------
       880
```

Instead of 1,440 blocks, we have 880, so the data is taking about 61% of the space it took originally.

All that is left now is to split the last partition at the end into a partition for April data and the rest.

```
ops$tkyte@ORA920> alter table audit_trail_table
  2  split partition the_rest
  3  at ( to_date('01-may-2003','dd-mon-yyyy') )
  4  into ( partition apr_2003, partition the_rest )
  5  /
Table altered.
```

Now, we are finished. It is interesting to note that March data is still modifiable here. We can insert, update, and delete that data. However, any rows we insert using a conventional INSERT or modify with an UPDATE will cause the affected data to be uncompressed. A table may consist of compressed blocks and uncompressed blocks. In fact, a single block may contain both compressed and uncompressed rows.

Compression Wrap-Up

Compression is a powerful tool for saving disk space, although care must be taken to trade this off with other physical storage considerations. If the location of data is key, data must be clustered in order to give the best performance, and compression might not be the way to go. On the other hand, if you need to store massive amounts of read-only data, such as audit trail information and data warehouse tables, compression may be the tool you need. Care must be taken to have the data physically bulk-loaded in a sorted order to maximize your compression ratios. Fortunately, simple CREATE TABLE AS SELECT statements make this a straightforward proposition.

In the area of performance, compressed tables fair well. There is the small overhead of decompressing the block, similar to the overhead you find with index key compression. However, the reduced I/O, more efficient use of the buffer cache, reduced latching, and other benefits outweigh any perceived overhead. To view a demonstration of a compressed table versus an uncompressed table, visit http://asktom.oracle.com/~tkyte/compress.html.

Summary

In this chapter, we covered many of the salient features of schema design. Unlike many other references on this topic, I didn't really go into data modeling or the normal forms. My goal was more pragmatic and more technical.

Many people confuse schema design with data modeling. To me, data modeling and schema design are related but separate events. One takes place before the other (data modeling and then physical schema design). The purity of the logical model is sometimes overcome by practical reality. For example, the people lookup example I used in this chapter is not something a logical data modeler would have come up with. Rather, the logical data model would have described the requirements, and the two-table people example would have been the physical, optimized realization of those requirements.

My fundamental schema principles are as follows:

- Let the database do your work. Use as many declarative statements as you can. Never implement procedurally what the database does declaratively.
- Always use the correct and most appropriate datatype. If you do not, you'll either be hating yourself over time if you need to maintain your own system, or the next people who need to do it will be using your name in vain for years to come.
- Optimize your database to efficiently and effectively answer your most frequently asked questions. I see many systems with pure logical models that stamped out a physical design—a physical design that looks nice on paper but doesn't respond very well in real life. Understand what will be asked of your system and design the system around that.

We looked at using various data structures in order to optimize access to data, paying the price at the time of insertion in order to optimize data retrieval. Remember that you generally insert a row of data once. You will query that data many times, so the price you pay during insertion will be repaid many times over, every time you retrieve the data. To design an efficient physical schema, you need to know what features and functions are available, how they work, and how to benchmark them under *your* conditions.