CHAPTER 6

Database Space Management

In This Chapter

Managing storage space

Allocating and managing storage space

Estimating space requirements

In a popular TV series, space is the final frontier, and it's full of surprises for the adventurous explorer. For the Oracle DBAs, space — in the form of disk storage space — is more often a big worry, one that hopefully won't lead to any surprises. As an Oracle DBA, you have to concern yourself with storage space at a variety of levels. You need to know how much the database as a whole uses. You also need to know, or be able to find out, how much storage space is used by any given table, index, or other object. Not only do you need to know how much storage space is being used now, but you also need to forecast your needs for the future. Storage space, and how Oracle manages it, is the subject of this chapter.

Managing Storage Space

Oracle8i uses logical structures to manage a database's storage space. This section discusses how storage space for a table is managed logically by way of segments, extents, and blocks. Of course, the logical structures must eventually be mapped onto physical structures, and we'll look at this as well. As you learn these structures, you'll gain an appreciation for all the work that Oracle does behind the scenes to manage the storage of your data.

Understanding storage related terms

Logical database structures provide a conceptual framework for managing data that is somewhat independent of the underlying physical hardware. One of the major benefits to using a database is that it frees you from having to remember the physical details of how your data is stored. The database software handles that for you. There are five types of logical structures:

- **Database objects** — A database object is a logical object containing the data that you want to store in a database. A table is such an object, as is an index.

- **Segments** — A segment represents a set of extents that contain data for a database object. Segments are stored in tablespaces.
- **Tablespaces** — A tablespace is the logical equivalent of a file.
- **Blocks** — A block is the smallest logical unit of storage managed by Oracle.
- **Extents** — An extent is a contiguous allocation of data blocks within a tablespace.

These logical structures allow you to manage your storage for your database in a platform-independent manner. No matter whether you are running Oracle8i on NT, VMS, or UNIX, you always manage disk storage in terms of objects, segments, tablespaces, extents, and blocks — the five items listed previously.

Aside from platform independence, logical structures provide you with flexibility. You can change the underlying physical structure of a database (by moving files around, for example) without changing its logical structure. Thus, your applications, which only see the logical structure, are completely unaffected by the changes.

Database objects

Tables and indexes are the two primary examples of *database objects*. They are the reason you have the database in the first place. You have tables of information that you want to store and retrieve, and you create indexes on those tables to facilitate retrieving that information.

Tables and indexes store data; thus, they are the database object types you are most concerned about relative to understanding how Oracle logically manages your data using tablespaces, extents, and data blocks. The one other database object that you should be aware of here is the cluster. A *cluster* is a database object that combines related tables together in such a way that related rows from one table are stored next to the related rows from the other table.

Segments

Tables, indexes, and clusters are stored in *segments*. For a nonpartitioned object, you always have exactly one segment. Partitioned objects get one segment per partition. Figure 6-1 shows the relationship between objects and segments for both a partitioned and a nonpartitioned object.

The nonpartitioned table in Figure 6-1 is stored in one segment. The other table is stored in two segments, one for each partition. Segments stand between an object and the tablespace or tablespaces used to store that object. Tablespaces, in turn, get you to files.

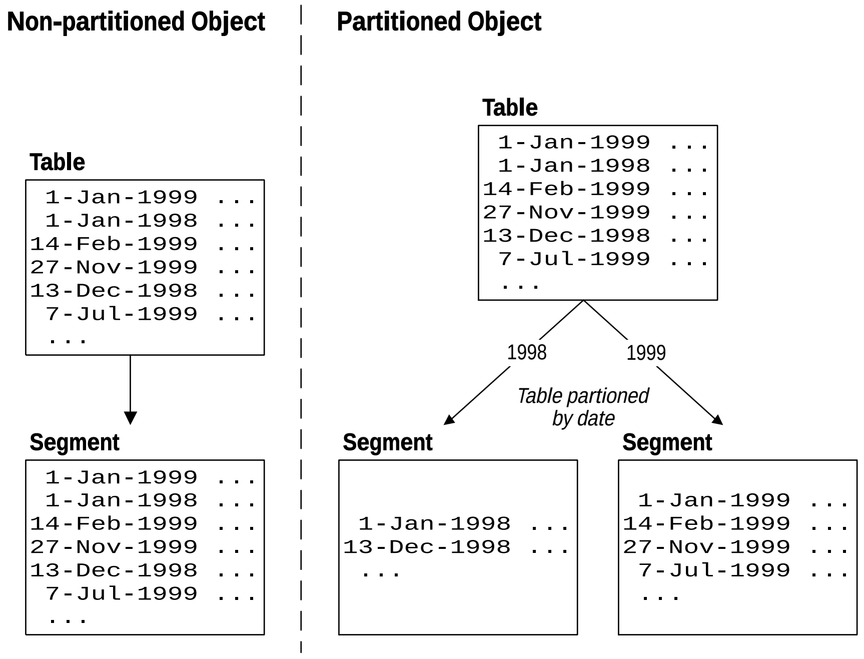

Figure 6-1: Objects have segments.

Tablespaces

You could consider a *tablespace* to be the logical equivalent of a data file. Rather than link each table and index directly to the file where the table or index data is stored, Oracle instead has you link each table and index segment to a tablespace. Data for a tablespace is then stored in one or more files. Figure 6-2 diagrams how the whole scheme looks.

Having a tablespace standing between a segment and the file used to store data for that segment provides you with some flexibility that you wouldn't otherwise have. For one thing, it allows you to transparently add files to a tablespace whenever you need more space. You don't have to do anything special to have Oracle use those files to store data for segments assigned to that tablespace. Another option is to move the files for a tablespace without having to change the definitions of objects stored within that tablespace.

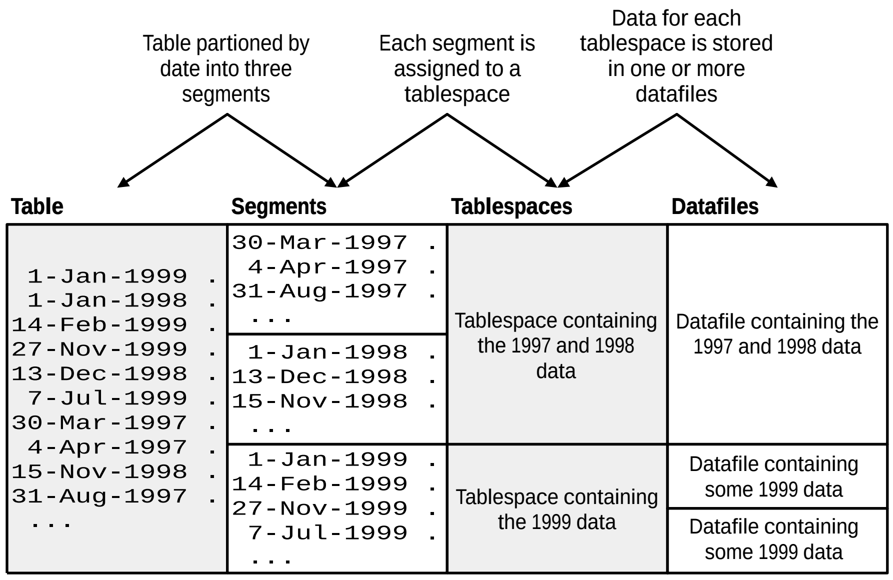
Table partioned by date into three segments
Each segment is assigned to a tablespace
Data for each tablespace is stored in one or more datafiles
Table
Segments
Tablespaces
Datafiles
1-Jan-1999 .
1-Jan-1998 .
14-Feb-1999 .
27-Nov-1999 .
13-Dec-1998 .
7-Jul-1999 .
30-Mar-1997 .
4-Apr-1997 .
15-Nov-1998 .
31-Aug-1997 .
...
30-Mar-1997 .
4-Apr-1997 .
31-Aug-1997 .
...
1-Jan-1998 .
13-Dec-1998 .
15-Nov-1998 .
...
1-Jan-1999 .
14-Feb-1999 .
27-Nov-1999 .
7-Jul-1999 .
...
Tablespace containing the 1997 and 1998 data
Tablespace containing the 1999 data
Datafile containing the 1997 and 1998 data
Datafile containing some 1999 data
Datafile containing some 1999 data

Figure 6-2: A table's segments are stored in a tablespace.

Data blocks

Data blocks are the smallest, most fundamental units of storage within a database. They are the smallest units of storage that Oracle can allocate to an object. Don't confuse Oracle data blocks with operating system blocks. An *operating system block* is the smallest unit that the operating system can read from or write to disk. Oracle data blocks are the smallest units that Oracle will read from or write to disk. Figure 6-3 diagrams how a tablespace is subdivided into Oracle data blocks and how those blocks relate to operating system blocks.

When Oracle reads data from disk, the amount of data it reads is always a multiple of the data block size. The data block size should always be an integer multiple of the operating system block size. Otherwise, you waste I/O and space because Oracle can't split an operating system block in half.

Extents

An *extent* is a contiguous allocation of data blocks. When Oracle needs to allocate space to an object, it doesn't just add one block. Instead, it allocates a whole group of blocks, and that group is referred to as an extent. Oracle does this to avoid having to constantly assign blocks one at a time to rapidly growing tables. You control the size of the extents that Oracle allocates to an object, and you should set the size large enough so that the object doesn't need to be extended often.

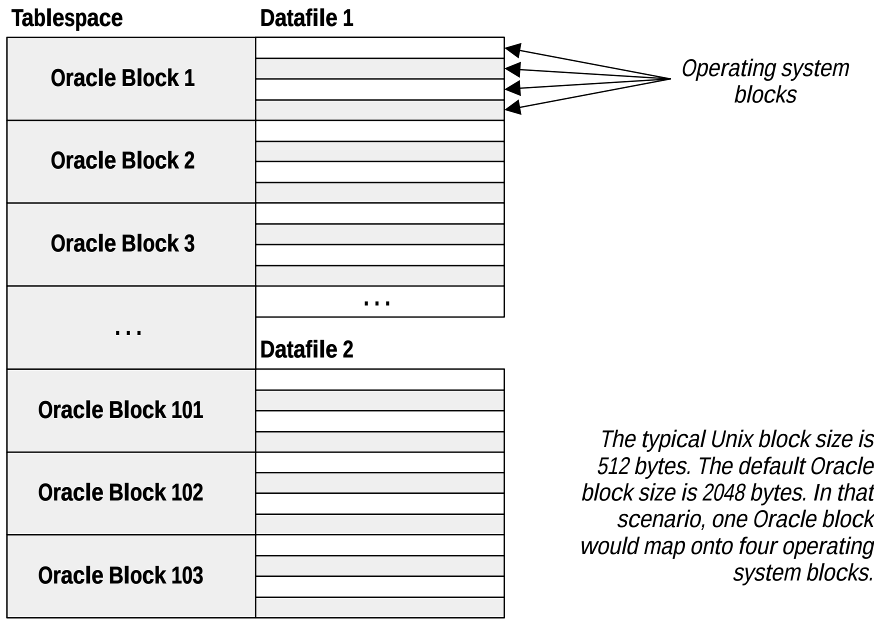
Tablespace
Datafile 1
Oracle Block 1
Oracle Block 2
Oracle Block 3
...
...
Datafile 2
Oracle Block 101
Oracle Block 102
Oracle Block 103
Operating system blocks
The typical Unix block size is 512 bytes. The default Oracle block size is 2048 bytes. In that scenario, one Oracle block would map onto four operating system blocks.

Figure 6-3: A tablespace contains many Oracle data blocks.

Figure 6-4 diagrams how extents figure into the space-allocation picture. Starting on the left, you have a table that is partitioned into two segments, each of which is assigned to a tablespace. Within each tablespace, the segments each have two extents. Extents are made up of Oracle data blocks. The tablespaces are each mapped onto two physical datafiles, and you can see how the Oracle datablocks are related to the operating system blocks.

You can see in Figure 6-4 that none of the extents crosses a datafile boundary. This is a rule that Oracle enforces. The blocks in an extent must be adjacent to each other, and they must also be in the same datafile.

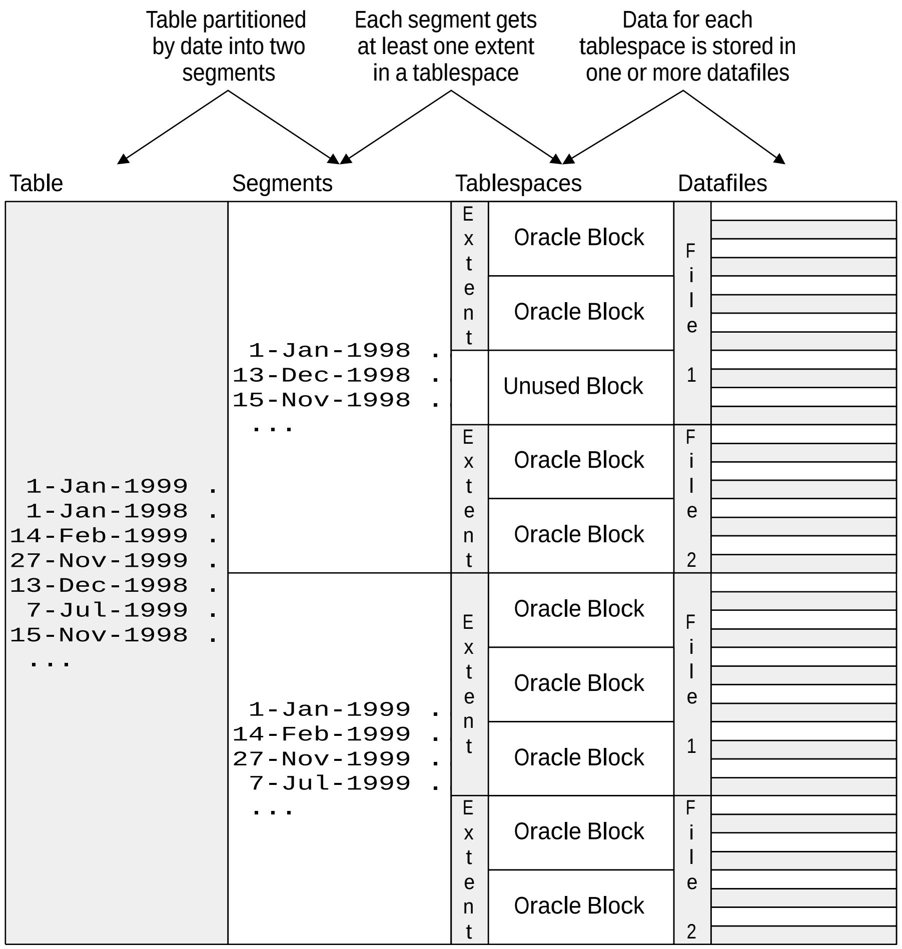

Figure 6-4: Extents in the overall space-management picture

Allocating and Managing Storage Space

To manage storage in an Oracle database, you need to be able to specify storage options for the tablespaces that you create. You also need to be able to specify storage options for the objects that you create in those tablespaces. In general, you need to know how to do the following:

- ✥ Create a tablespace
- ✥ Set tablespace options, such as the minimum extent, whether to autoextend, and so forth.

- Set default storage parameters for a tablespace
- Set the storage parameters for an object
- Add space when needed
- Coalesce extents

Each of these topics is covered in one the following sections.

Creating a tablespace

You use the `CREATE TABLESPACE` command to create a tablespace. At a minimum, when creating a tablespace, you need to tell Oracle how large to make the tablespace and where to put it. For example:

```
CREATE TABLESPACE CHECKUP_HISTORY
    DATAFILE '/m01/oradata/BIBDB/checkup_history.dbf'
    SIZE 1000M;
```

This statement creates a file named checkup_history.dbf that is 1,000MB in size. That file will be used to hold all data stored within the `CHECKUP_HISTORY` tablespace.

Tablespace Storage Parameters

When you create a tablespace, Oracle allows you to specify default storage parameters to be used for objects that you later create and assign to that tablespace. The five parameters that you can set at the tablespace level are the following:

`INITIAL`	The `INITIAL` parameter sets the size of the initial extent that is allocated to each new object when you create it.
`NEXT`	The `NEXT` parameter sets the size to use for subsequent extents. Often, the `INITIAL` and `NEXT` settings are identical.
`MINEXTENTS`	The `MINEXTENTS` parameter defines the minimum number of extents to allocate when a new object is created. Most of the time, it makes sense to set `MINEXTENTS` to 1.
`MAXEXTENTS`	The `MAXEXTENTS` parameter defines the maximum number of extents that Oracle will allow an object to have. You can supply a specific number, or you can use the keyword `UNLIMITED`.
`PCTINCREASE`	The `PCTINCREASE` parameter establishes a percentage by which to increase the `NEXT` value each time a new extent is added for an object. It's usually best to set this to 0.

For example, to specify that objects created and assigned to a tablespace have extent sizes of 50MB, and that no limit is placed on the number of extents for an object, you could issue the following command:

```
CREATE TABLESPACE checkup_history
    DATAFILE '/m01/oradata/BIBDB/checkup_history.dbf'
    SIZE 1000M
DEFAULT STORAGE (    INITIAL 50M
                     NEXT 50M
                     MINEXTENTS 1
                     MAXEXTENTS UNLIMITED
                     PCTINCREASE 0);
```

Storage parameter values specified at the tablespace level represent defaults. They are used only when you create a new object in the tablespace without specifying storage parameters specifically for the object. The table created by the following command, for example, will inherit the default storage parameters shown previously:

```
CREATE TABLE checkup_history (
     CHECKUP_NO          NUMBER(10,0) NOT NULL,
     ID_NO               NUMBER(10,0),
     CHECKUP_TYPE        VARCHAR2(30),
     CHECKUP_DATE        DATE,
     DOCTOR_NAME         VARCHAR2(50)
     ) TABLESPACE checkup_history;
```

Since no `STORAGE` clause was given in the `CREATE TABLE` statement, the `checkup_history` table will inherit its storage parameter settings from the tablespace. An initial extent of 50MB will be allocated immediately, and future extents of 50MB will be allocated as needed. Figure 6-5 illustrates this allocation of space.

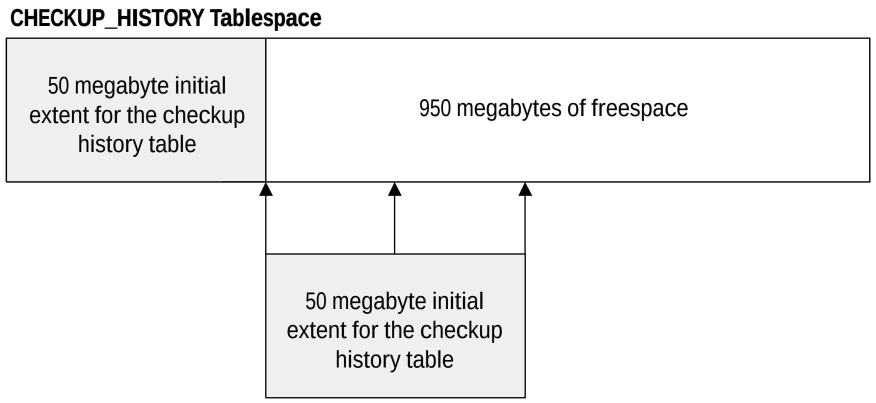
CHECKUP_HISTORY Tablespace
50 megabyte initial extent for the checkup history table
950 megabytes of freespace
50 megabyte initial extent for the checkup history table

Figure 6-5: The initial extent for the checkup_history table

The tablespace default storage parameters are referenced when you create a new object in the tablespace. Subsequently changing the parameters at the tablespace level won't affect objects that have already been created.

The Minimum Extent Size

When many objects with different extent sizes are stored in a tablespace, it can become fragmented. The upcoming section, "Coalescing Extents," talks more about fragmentation. For now, though, realize that fragmentation may create extents of free space that aren't big enough to be very useful. Figure 6-6 illustrates how this might happen.

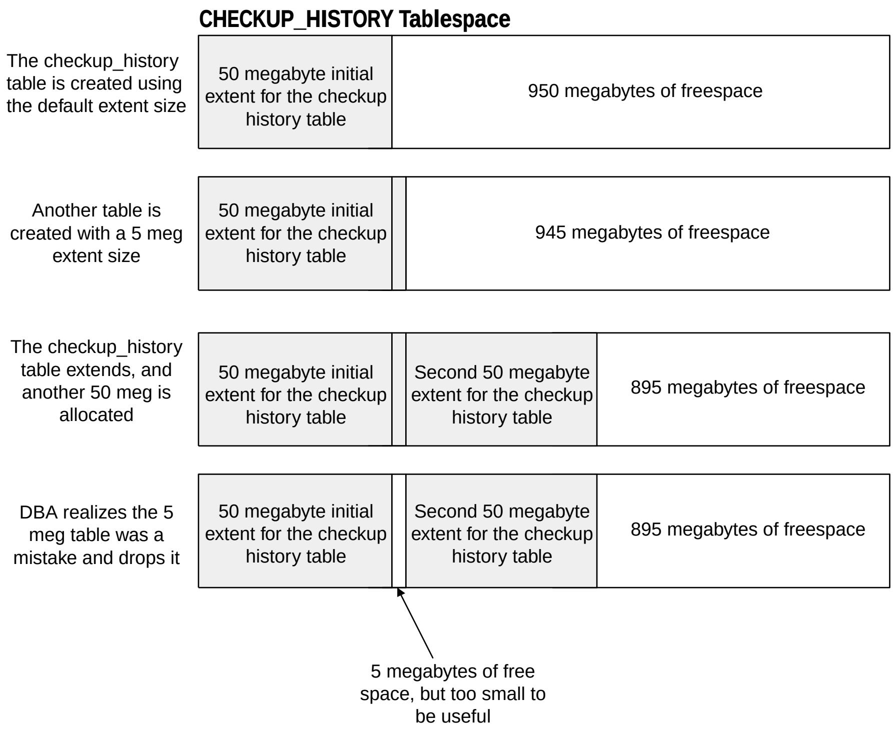
CHECKUP_HISTORY Tablespace
The checkup_history table is created using the default extent size
50 megabyte initial extent for the checkup history table
950 megabytes of freespace
Another table is created with a 5 meg extent size
50 megabyte initial extent for the checkup history table
945 megabytes of freespace
The checkup_history table extends, and another 50 meg is allocated
50 megabyte initial extent for the checkup history table
Second 50 megabyte extent for the checkup history table
895 megabytes of freespace
DBA realizes the 5 meg table was a mistake and drops it
50 megabyte initial extent for the checkup history table
Second 50 megabyte extent for the checkup history table
895 megabytes of freespace
5 megabytes of free space, but too small to be useful

Figure 6-6: Events leading up to fragmentation and a uselessly small extent

To help you prevent a tablespace from being fragmented into extents that are too small to be useful, Oracle allows you to specify a minimum extent size for a tablespace. The following statement creates a tablespace and specifies a minimum extent size of 25MB:

```
CREATE TABLESPACE checkup_history
    DATAFILE '/m01/oradata/BIBDB/checkup_history.dbf'
    SIZE 1000M
    MINIMUM EXTENT 25M
DEFAULT STORAGE (    INITIAL 50M
                     NEXT 50M
                     MINEXTENTS 1
                     MAXEXTENTS UNLIMITED
                     PCTINCREASE 0);
```

Oracle will not allow any extents less than 25MB to be allocated in the `checkup_history` tablespace created with this statement. With respect to the sequence of events shown in Figure 6-6, the command to create a table with an initial size of 5MB will succeed, but Oracle will silently allocate a 25MB initial extent, because that is the minimum size allowed.

By preventing extents of such a small size from ever being created, you ensure that even if fragmentation occurs, the fragmented extents will at least be of a usable size.

The Autoextend Option

Normally, when you create the initial datafile for a tablespace, the size of that datafile is fixed. Adding more space to the tablespace requires that you create more datafiles. Historically, that was the way it worked — period. However, in one of the Oracle7 releases, Oracle began allowing you to create datafiles that can extend in size automatically. You do this by specifying `AUTOEXTEND ON` when you create a tablespace, or when you add a datafile to a tablespace. Following is an example showing a tablespace being created with a 100MB datafile that will grow in 100MB chunks:

```
CREATE TABLESPACE checkup_history
    DATAFILE 'e:\Oracle\Oradata\jonathan\checkup_history.dbf'
    SIZE 100M
    AUTOEXTEND ON NEXT 100M MAXSIZE UNLIMITED
    MINIMUM EXTENT 25M
DEFAULT STORAGE (   INITIAL 50M
NEXT 50M
MINEXTENTS 1
MAXEXTENTS UNLIMITED
PCTINCREASE 0);
```

Turning the autoextend option on for your datafiles truly puts your database's storage space management on automatic. The tablespace created with the command just shown can grow in 100MB chunks until you run out of disk space, and objects created within the tablespace can grow in 50MB chunks, also until you run out of disk space.

Locally Managed Tablespaces

New with Oracle8i is the ability to create locally managed tablespaces. With regular tablespaces, referred to now as *dictionary-managed tablespaces,* the data dictionary keeps track of how space has been allocated. A *locally managed tablespace* is one where the space management is done using bitmaps that are stored within the tablespace itself, hence, the term *local.* The result is faster extent allocation as objects grow.

The `LOCAL` keyword is used to create a locally managed tablespace, and two options exist for managing extents: `AUTOALLOCATE` and `UNIFORM`. The following two statements illustrate these options:

```
CREATE TABLESPACE checkup_history
    DATAFILE '/m01/oradata/BIBDB/checkup_history.dbf'
    SIZE 1000M
    EXTENT MANAGEMENT LOCAL AUTOALLOCATE;

CREATE TABLESPACE checkup_history
    DATAFILE '/m01/oradata/BIBDB/checkup_history.dbf'
    SIZE 1000M
    EXTENT MANAGEMENT LOCAL UNIFORM SIZE 50M;
```

The `AUTOALLOCATE` option causes the tablespace to be created with a bitmap containing one bit for each data block. Oracle controls extent size. When you create an object in the tablespace, Oracle will start out by allocating a 64KB extent to that object. As the object grows in size, more 64KB extents will be added. Oracle will eventually increase the size of extents allocated to that object.

The `UNIFORM` clause allows you to manage storage space in extents that are larger than one block. All extents must be the same size, hence, the term *uniform*. In this example, the uniform extent size is 50MB, and the bitmap will contain one bit for each 50MB of storage space in the tablespace.

Note The uniform extent size will be rounded up to the nearest integer multiple of the database block size.

Setting storage parameters for an object

You've seen how to use the `DEFAULT STORAGE` clause to specify the default storage characteristics for objects created within a tablespace. You can use a similar clause when you create an object that allows you to override those default characteristics.

Dictionary-Managed Tablespaces

You can use the STORAGE clause when you create an object to specify the storage parameters to use for that object. For example, the following statement creates a table with a 15MB initial extent. Subsequent extents will be 5MB each.

```
CREATE TABLE checkup_history (
     CHECKUP_NO         NUMBER(10,0) NOT NULL,
     ID_NO              NUMBER(10,0),
     CHECKUP_TYPE       VARCHAR2(30),
     CHECKUP_DATE       DATE,
     DOCTOR_NAME        VARCHAR2(50)
     ) TABLESPACE checkup_history_local
       STORAGE (INITIAL 15M NEXT 5M);
```

Anything that you specify in the STORAGE clause for an object overrides the corresponding DEFAULT STORAGE value that you specify at the tablespace level. The same values, INITIAL, NEXT, PCTINCREASE, MINEXTENTS, and MAXEXTENTS, which you can specify in the DEFAULT STORAGE clause, may also be used in the STORAGE clause.

Locally Managed Tablespaces

When you are creating objects in a locally managed tablespace, the rules for extent allocation change. If you are using the AUTOALLOCATE option, Oracle will control the size and number of extents based on an internal algorithm. While the details of this algorithm haven’t been published, the following statements are true:

- Oracle will allocate an initial extent of 64KB by default.
- Oracle will allocate an initial extent of 64KB, even if you ask for less.
- If you ask for an initial extent greater than 64KB, Oracle will allocate enough 64KB extents to give you at least the amount of space that you asked for.
- Oracle will ignore any value that you supply for PCTINCREASE.
- Oracle will ignore any value that you supply for NEXT.
- Beginning with the 16th extent, Oracle will increase the extent size from 64KB to 1MB.
- Oracle will ignore any MAXEXTENTS value.

If you configure a locally managed tablespace to use uniform extents, then Oracle ensures that all extents in the tablespace are the same size. As with the AUTOALLOCATE option, enough uniform extents will be allocated to at least match the amount of space requested for the initial extent.

Just remember that when you use a uniform extent size, all extents will be that size. When you use the AUTOALLOCATE option, Oracle uses an internal algorithm to control the extent size.

Coalescing extents

Figure 6-6 illustrated how you can end up with a small orphan extent of free space in a tablespace. In cases where you have a large amount of extent creation and deletion, it’s possible to end up with several contiguous extents of free space. Figure 6-7 diagrams how this might look.

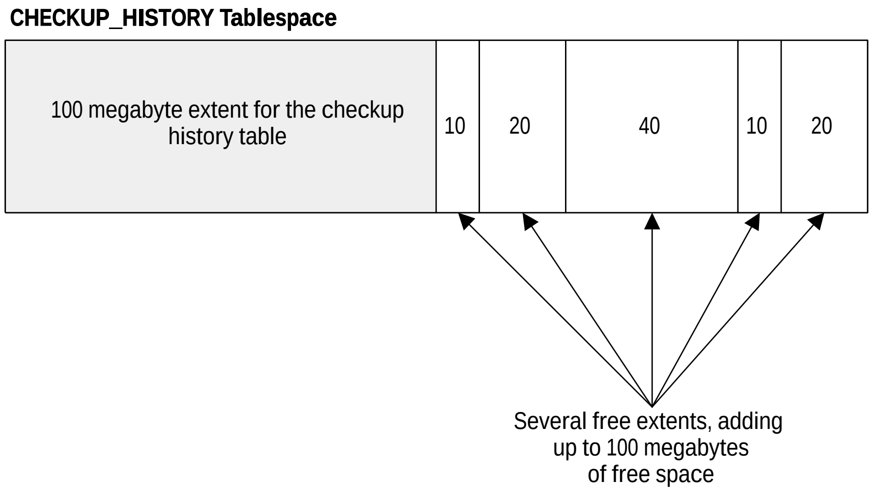
CHECKUP_HISTORY Tablespace
100 megabyte extent for the checkup history table
10
20
40
10
20
Several free extents, adding up to 100 megabytes of free space

Figure 6-7: Several contiguous extents of free space

The tablespace shown in Figure 6-7 contains 100MB of free space organized into several extents ranging in size from 10–40MB each. Sometimes this can present a problem. Even though there are 100MB of contiguous free space, it doesn’t appear that way because that space has been fragmented into several smaller pieces.

The solution to this problem of fragmented free space is to *coalesce* all the adjacent free extents into one large extent. There are two ways you can do this. One is to just wait. One of the functions of the SMON background process is to continuously scan for this type of situation and to coalesce adjacent free extents whenever it finds them. However, SMON only does this for tablespaces with a nonzero default `PCTFREE` setting. Oracle’s extent allocation algorithm is also supposed to do this for you. However, if you run into problems allocating storage space and you need the space right away, you can manually coalesce the extents. The following `ALTER TABLESPACE` command will do that:

```
ALTER TABLESPACE tablespace_name COALESCE;
```

Figure 6-8 shows the same tablespace as the one shown in Figure 6-7, but after the free space has been coalesced. Notice that there is now one large 50MB extent of free space.

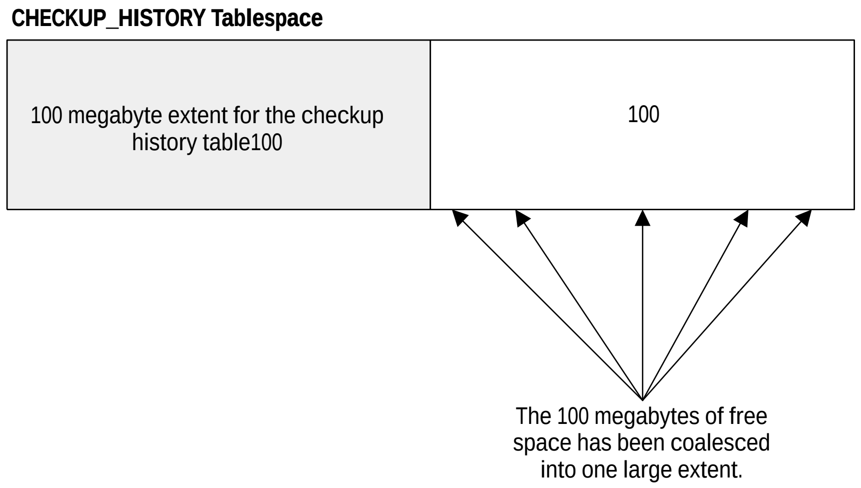
CHECKUP_HISTORY Tablespace
100 megabyte extent for the checkup history table100
100
The 100 megabytes of free space has been coalesced into one large extent.

Figure 6-8: The free space has been coalesced into one large extent.

Adding storage space when needed

Over time, the objects in your database will grow in size, and sooner or later, you will need to add more space to your database. As long as you have enough disk space, adding storage space is actually quite easy. The trick is in knowing when, and how much.

Knowing When You Need More Space

You know that you need more storage space when one of your users calls you about an error message like this:

```
ORA-01653: unable to extend table HR.EMPLOYEE by 2560 in tablespace HR_DATA
```

Of course, this isn’t the way that you want to find out! You won’t last long as a DBA if your space-management methodology is to wait until you run out. When you drive a car on a trip, you have some general idea of how far you can go on one tank of gas, and you monitor the gas gauge periodically so that you know ahead of time when you need to fill up. You need to do the same with your database. You need to monitor and to plan for the future. To do this, you need to be able to answer the following questions:

- ✦ How large are your objects (tables, indexes, and so on) now?
- ✦ How fast are your objects growing?
- ✦ How much free space do you have left now?

The next few sections will help you answer the first and last questions in this list. They are easy questions because they refer to the current state of the database. Predicting the future is a bit tougher, though.

The most reliable way to see how fast your database objects are growing is to check their size regularly. Once you've determined how much space is added to an object every day, every week, or during whatever interval you choose to measure, you can use that value to extrapolate into the future. For example, if a table is growing by 1MB per day, you can extrapolate that out to 30MB a month.

Tip

It can be cumbersome to regularly monitor the size of every object in a database. Often, only a few tables and indexes grow significantly on a daily basis, with the remaining tables being relatively static. Focus your efforts on the fast-growing objects, and monitor them often, perhaps daily. Choose a longer interval for the more static objects.

In addition to watching how fast your objects are growing, keep a close eye on the amount of free space remaining in each of your tablespaces. You should have some idea of an acceptable minimum for each tablespace. Think in terms of extents here. If the objects in a tablespace all have 50MB extent sizes, then you need some multiple of that amount free. When the amount of free space drops below this minimum, allocate more space.

Adding Storage Space to a Tablespace

You can take two approaches to add storage space to a tablespace. You can add a datafile to the tablespace, or you can attempt to extend one of the existing datafiles.

You use the `ALTER TABLESPACE` command to add a datafile to a tablespace. The following example adds a 1000MB datafile to the `checkup_history` tablespace:

```
ALTER TABLESPACE checkup_history
    ADD     DATAFILE '/m02/oradata/BIBDB/checkup_history_2.dbf'
            SIZE 1000M;
```

You can add as many datafiles to a tablespace as you like, subject to the overall limit on the number of database files set by the `DB_FILES` initialization parameter.

Adding Space to a Datafile

Another way to add space to a tablespace is to resize one or more datafiles associated with the tablespace. You use the `ALTER DATABASE` command to do this. The following example resizes the checkup_history_2.dbf datafile to 1,500MB:

```
ALTER DATABASE
    DATAFILE '/m02/oradata/BIBDB/checkup_history_2.dbf'
    RESIZE 1500m;
```

Resizing a datafile works both ways: You can make a file larger, or you can attempt to shrink it smaller. You can resize a file downward as long as you are not throwing away blocks that are being used.

Manually Adding an Extent to a Table

You can arbitrarily allocate space to a table by using the `ALTER TABLE` command. The command in the following example causes Oracle to allocate a 5MB extent for the `checkup_history` table:

```
ALTER TABLE checkup_history
    ALLOCATE EXTENT (SIZE 5M);
```

Allocating space for a table involves some overhead. One advantage to manually allocating space like this is that you control when that overhead is incurred. The alternative is that some random user is going to absorb the overhead of extending the table.

Note

The `SIZE` parameter shown in the previous example is optional. You can issue the command `ALTER TABLE checkup_history ALLOCATE EXTENT`, and the size will default to the value for `NEXT`.

Another advantage to manually allocating storage space like this is that you can assure yourself that the table has enough space for some well-defined time period. For example, if you know the `checkup_history` table is growing by 5MB per month, you could manually allocate a 5MB extent at the beginning of each month. Then you can be pretty sure, barring some sudden spurt of growth, that you won't run out of space during the next month.

If you have a tablespace with several datafiles, you can force the extent to be allocated in a file of your choosing. For example:

```
ALTER TABLE checkup_history
    ALLOCATE EXTENT (
        SIZE 5M
        DATAFILE '/m02/oradata/BIBDB/checkup_history_2.dbf'
        );
```

This gives you the ability to manually strip your data across disks. If you have 20MB of data and four disks, you can create a datafile on each disk and manually allocate a 5MB extent in each of those datafiles. The result is that your 20MB of data is spread over four disks, which may reduce disk contention somewhat.

Note While you can specify the datafile to use when manually extending an existing object, you cannot specify the datafile to use for the initial extent when you first create the object.

Reporting an object's storage space usage

Two data dictionary views are key to finding out how a database is using storage space. These views are:

- DBA_EXTENTS
- DBA_FREE_SPACE

The DBA_EXTENTS view tells you about extents that have been allocated to objects such as tables, indexes, and so forth. The DBA_FREE_SPACE view returns information about chunks of free space within a tablespace.

While the DBA_EXTENTS view returns information about space that has been allocated to an object, it doesn't tell you how much of that space has been used. The ANALYZE command can help in this area. To find out how many blocks are actually being used to store information for a table, you can issue an ANALYZE TABLE command, followed by a query to DBA_TABLES.

The next few subsections show you how to determine the amount of space allocated to an object such as a table or a view; how to determine the free and used space within a tablespace; and how to use the ANALYZE command to find out how much data a table (or index) really contains.

Storage Space Allocated to an Object

To find out how much storage space has been allocated to an object, you need to query the DBA_EXTENTS view. The following query will tell you how much space the CHECKUP_HISTORY table has used:

```
SELECT COUNT(*) extents,
       SUM(bytes) bytes_used,
       SUM(blocks) blocks_used
FROM dba_extents
WHERE owner='SEAPARK'
  AND segment_name='CHECKUP_HISTORY'
  AND segment_type='TABLE';
```

The blocks column in DBA_EXTENTS tells you how many Oracle blocks make up the extent. The bytes column tells you how many bytes those blocks represent. It is the number of blocks multiplied by the database block size (from the DB_BLOCK_SIZE initialization parameter).

In this example, we are querying for the space allocated to a table, so we used a segment type of TABLE. If you were interested in the size of an index, you would use the index name and a segment type of INDEX. The valid segment types are CACHE, CLUSTER, DEFERRED ROLLBACK, INDEX, INDEX PARTITION, LOBINDEX, LOBSEGMENT, NESTED TABLE, ROLLBACK, TEMPORARY TABLE, and TABLE PARTITION.

You can get a report showing the space allocated to all objects in a tablespace by using the following query. Just replace *tablespace_name* with the name of the tablespace that you are interested in.

```
SELECT owner,
       segment_name,
       segment_type,
       count(*) extents,
       SUM(bytes) bytes_used,
       SUM(blocks) blocks_used
FROM dba_extents
WHERE tablespace_name='SYSTEM'
GROUP BY owner, segment_name, segment_type
ORDER BY owner, segment_name, segment_type;
```

The queries shown previously will work for partitioned and nonpartitioned objects that don’t contain large objects or nested tables stored in their own segments. For example, say that you created a table with a large object, and you stored that large object in its own tablespace. Your CREATE TABLE command might look like this:

```
CREATE TABLE items (
item_no        NUMBER,
item_photo     BLOB
) TABLESPACE item_data
  LOB (item_photo)
      STORE AS items_item_photo (TABLESPACE item_photos);
```

In this case, except for the item_photo column, the table data is stored in a segment named ITEMS, in the tablespace named item_data. The item_photo column, however, has been relegated to a separate tablespace named item_photos, where it will be stored in a segment named items_item_photo. To find the total space used by this table, you would need to issue the two queries shown in Listing 6-1.

The first query returns the amount of space allocated to the main table, while the second query returns the amount of space allocated to the item_photo column.

Listing 6-1: Finding total space in a table

```
SELECT COUNT(*) extents,
       SUM(bytes) bytes_used,
       SUM(blocks) blocks_used
FROM dba_extents
WHERE owner='SEAPARK'
  AND segment_name='ITEMS'
  AND segment_type='TABLE';

SELECT COUNT(*) extents,
       SUM(bytes) bytes_used,
       SUM(blocks) blocks_used
FROM dba_extents
WHERE owner='SEAPARK'
  AND segment_name='ITEMS_ITEM_PHOTO'
  AND segment_type='LOBSEGMENT';
```

Space Used in a Tablespace

The queries are quite simple to determine how much storage space has been used within a tablespace, versus how much is still free. The following query will tell you how much space each of your tablespaces has used:

```
SELECT ts.tablespace_name,
       NVL(COUNT(extent_id),0) extents,
       SUM(NVL(bytes,0)) bytes_used,
       SUM(NVL(blocks,0)) blocks_used
FROM dba_tablespaces ts, dba_extents ex
WHERE ts.tablespace_name = ex.tablespace_name(+)
GROUP BY ts.tablespace_name;
```

This query is written to return zeros for tablespaces where no extents have been allocated. If you didn't care to see that, you could eliminate the outer join, making the query a bit simpler.

Free Space in a Tablespace

The DBA_FREE_SPACE view returns information about the *unused* space within a tablespace. The following query returns the amount of unused space within each tablespace, as well as the size of the largest free extent in each tablespace:

```
SELECT ts.tablespace_name,
       NVL(COUNT(fs.block_id),0) free_extents,
       SUM(NVL(bytes,0)) bytes_free,
       SUM(NVL(blocks,0)) blocks_free,
       MAX(NVL(bytes,0)) largest_free_extent
FROM dba_tablespaces ts, dba_free_space fs
WHERE ts.tablespace_name = fs.tablespace_name(+)
GROUP BY ts.tablespace_name;
```

Knowing the largest free extent is helpful because when an object extends, it not only needs enough free space to extend, but it needs that space in one contiguous chunk.

Blocks Actually Used by an Object

The DBA_EXTENTS view tells you how much space has been allocated to an object, such as a table, but that's not always a good indicator of how large that object really is or how much space it is really using. Extents of any size can be allocated to empty tables, and it's common to allocate large extents in anticipation of future growth. How, then, do you find out how much space a table is really using versus what has been allocated to it?

One way to find out exactly how much storage space a table or an index is really using is to use the ANALYZE command. When you analyze a table, Oracle counts up the number of rows in the table, computes the average row size, and tallies the number of blocks actually being used to store data. Oracle does much the same thing when you analyze an index, except that it counts keys instead of rows, and it computes some other information that is specific to indexing.

The example in Listing 6-2 shows the ANALYZE command being used on the SEAPARK user's CHECKUP table:

Listing 6-2: **Using the ANALYZE command**

```
SQL> ANALYZE TABLE SEAPARK.CHECKUP COMPUTE STATISTICS;

Table analyzed.

SQL> SELECT blocks,
  2         num_rows,
  3         empty_blocks,
  4         avg_space,
  5         avg_row_len
  6  FROM dba_tables
  7  WHERE owner='SEAPARK'
  8    AND table_name='CHECKUP';

   BLOCKS  NUM_ROWS EMPTY_BLOCKS AVG_SPACE AVG_ROW_LEN
--------- --------- ------------ --------- -----------
        1         3            3      1904          16
```

As you can see, after the ANALYZE command has completed, the following columns were selected from DBA_TABLES:

BLOCKS	The BLOCKS column lists the number of blocks that have actually been used to store data.

NUM_ROWS	The NUM_ROWS column lists the number of rows in the table.
EMPTY_BLOCKS	The EMPTY_BLOCKS column lists the number of blocks allocated to the table that have never been used.
AVG_SPACE	The AVG_SPACE column lists the average amount of free space per block.
AVG_ROW_LEN	The AVG_ROW_LEN column lists the average row length.

Note

The value for EMPTY_BLOCKS doesn't include blocks that were once used but are now empty. It only includes those blocks that have never been used.

The CHECKUP table in the previous example has three rows with an average row length of 16 bytes. Multiplying those two numbers gives a total of 48 bytes of data. Only one block is being used to store that data. To properly interpret the AVG_SPACE figure, you need to know the block size. In this case, the block size is 2,048 bytes, so 2,048 – 1,904 = 144 bytes used per block, on average. Note that the bytes used per block includes overhead, so that figure won't necessarily match the result of multiplying NUM_ROWS by AVG_ROW_SIZE.

Estimating Space Requirements

When you create a new table or a new index, you need to estimate how much storage space will be required for the amount of data that you expect to have. There are three ways of going about this task:

- ✦ You can make an educated guess.
- ✦ You can extrapolate from a subset of data.
- ✦ You can calculate an estimate based on row size, block size, estimated free space within a block, and so forth.

Making an educated guess is by far the simplest approach, and in some cases it may be the most appropriate to use. Consider the SALAD_TYPE table owned by the user named AMY in the sample database for this book, and which is made up of the following columns:

```
SALAD_TYPE        VARCHAR2(10)
DESCRIPTION_TEXT  VARCHAR2(40)
ORGANIC_FLAG      VARCHAR2(3)
PRICE_PER_POUND   NUMBER(10,2)
LAST_CHANGE_DATE  DATE
```

How many types of salads could there possibly be? For a table this small, unless you just can't afford to make any mistake at all, you probably don't need to be too rigorous. If you allocated a 100KB extent and ended up wasting 80KB of space, so what? If you allocated a 10KB extent and ended up needing another 10KB extent, that also would be no big deal.

Perhaps the most reliable method of estimating the space required for a table or an index is to extrapolate based on a small amount of sample data. Let's go back to the SALAD_TYPE example. Assume for a moment, absurd as it may seem, that you are expecting 10,000 rows to be in that table. Maybe there really are 10,000 types of salad in the world. One way of estimating the amount of space required for those rows would be to populate the table with 100 representative rows, find out how much space is required for those rows, and multiply that value by 100. You can use the ANALYZE command to determine the space actually used in a table. The following example shows ANALYZE being used on a SALAD_TYPE table that contains 100 rows:

```
SQL> ANALYZE TABLE salad_type COMPUTE STATISTICS;

Table analyzed.

SQL> SELECT blocks, avg_row_len
  2  FROM user_tables
  3  WHERE table_name='SALAD_TYPE';

   BLOCKS AVG_ROW_LEN
--------- -----------
        4          54

1 row selected.
```

In this example, the average length of a row in the SALAD_TYPE table is 54 bytes, and it took four blocks to hold 100 records. Extrapolating from this, you can be pretty safe in assuming that 400 blocks would hold 10,000 records.

Note With such a small initial sample, you may end up allocating more space than is absolutely necessary for a table. Ninety-nine rows may fit in three blocks, and the 100th row might have been barely enough to spill over into the fourth.

Remember, unused space is also being multiplied by 100. You can refine the estimate by increasing the sample size. This example predicts 40 blocks for 1,000 rows, but when you increase the sample to 1,000 rows, you'll find that only 34 blocks are used. Extrapolating from that, you would expect 340 blocks for 10,000 rows. In fact, only 325 blocks were necessary. In this case, the estimate based on 1,000 rows was closer to the final result than the estimate based on 100 rows.

Estimating future storage space requirements based on existing data works only when you have some representative data to look at. If you are developing or implementing a new system, you may be forced, because of the schedule, to create a database well before you have any data to put in it. If that's the case, and you still need a rigorous size estimate, you can still use some calculations to get that. The next few sections describe the information that you need and show you the formulas to use.

Sizing Small Code Tables

How would you really size a small table like SALAD_TYPE? First of all, look at other code tables in the system. If the INITIAL and NEXT extent sizes are consistent for the other small tables, you would probably follow that pattern for SALAD_TYPE. Often, a database will have a tablespace set aside specifically for small code tables, with reasonable default storage parameters already set. If no pattern exists to follow, you might choose to make a rough estimate of the space required.

The sum of the VARCHAR2 column lengths in the table is 53. The DATE columns are 7 bytes each, and the NUMBER column in this case will use about 5 bytes. That works out to 65 bytes per row. Allowing for 25 salad types, you would have 25 rows × 65 bytes = 1,625 bytes total. If you were doing this calculation in your head, you might round up the row size to 100, to make the math easier. Whatever value you came up with for a total size, you would then round up to the nearest block size increment and use that value for your initial and next extent sizes.

You'll recognize that this calculation yields only a crude estimate. Often, though, that's enough to get you in the ballpark. See the section, "Estimating the Space for a Table," for a more rigorous set of calculations.

Collecting prerequisite information

The formulas described in this chapter come from the *Oracle8 Administrator's Guide*. To use them, you need to query your database to find out your database block size. You also need to retrieve the sizes for the following types: KCBH, UB1, UB2, UB4, KTBBH, KTBIT, KDBH, and KDBT. You can retrieve the database block size from the V$PARAMETER view. The type sizes come from the V$TYPE_SIZE view. Listing 6-3 provides an example that shows how you can query for these values.

Listing 6-3: **Retrieving the database block size value**

```
SQL> SELECT value
  2  FROM v$parameter
  3  WHERE name='db_block_size';

VALUE
--------------------------------------------------------------
2048

1 row selected.

SQL>
SQL> SELECT component, type, description, type_size
```

Continued

Listing 6-3: *(continued)*

```
FROM v$type_size
WHERE type IN ('KCBH','UB1','UB2','UB4',
               'KTBBH','KTBIT','KDBH','KDBT',
               'SB2');

COMPONENT TYPE     DESCRIPTION                      TYPE_SIZE
--------- -------- -------------------------------- ---------
S         UB1      UNSIGNED BYTE 1                          1
S         UB2      UNSIGNED BYTE 2                          2
S         UB4      UNSIGNED BYTE 4                          4
S         SB2      SIGNED BYTE 2                            2
KCB       KCBH     BLOCK COMMON HEADER                     20
KTB       KTBIT    TRANSACTION VARIABLE HEADER             24
KTB       KTBBH    TRANSACTION FIXED HEADER                48
KDB       KDBH     DATA HEADER                             14
KDB       KDBT     TABLE DIRECTORY ENTRY                    4

9 rows selected.
```

In addition to the type sizes and the database block size, you need to know the values that you are going to use for the `INITRANS` and `PCTFREE` parameters when you create the table. If you aren't going to specify those parameters, then use the defaults for the calculations. The default value for `INITRANS` is 1, and the default for `PCTFREE` is 20.

Estimating the space for a table

To estimate the size of a table, you need to do the following:

1. Figure out how much space in each database block is available for data.
2. Determine an average row size on which to base your estimate.
3. Compute the number of blocks required based on the previous two values.

The following sections show you some formulas that you can use for each of these steps. After that, there is an example based on the `SALAD_TYPE` table scenario discussed earlier.

Estimating the Amount of Available Space per Block

The amount of storage space available in a block depends primarily on the block size, the amount of overhead, and the `PCTFREE` value. Oracle provides the following formula:

```
SPACE_FOR_DATA = CEIL((DB_BLOCK_SIZE - KCBH - UB4 -KTBBH
- ((INITTRANS - 1) * KTBIT) - KDBH)
* (1-PCTFREE/100)) - KDBT
```

The `CEIL` function is an Oracle function that rounds fractional values up to the nearest integer. For example, `CEIL (5.3)` results in a value of 6.

Estimating the Row Size

Estimating the row size is perhaps the trickiest part of this entire process. It's fairly easy to come up with a maximum row size. It's tougher to settle on a good average to use for estimation purposes.

The first step in calculating the row size is to sum up the lengths of the individual columns. Table 6-1 will help you do this. As you work with each column, give some thought about your expectations in terms of the average amount of data the column will hold. A description field, for example, may be defined as a `VARCHAR(40)`, but how often are you likely to save a description that uses all 40 characters? You may want to compute column sizes based on what you expect a reasonable average to be.

Table 6-1
Column Sizes

Datatype	*Size*
`CHAR(x)`	The column size will be x bytes.
`NUMBER(x,y)`	Number columns have a maximum size of 21 bytes. The amount used depends on the value being stored. Figure two digits per byte: one digit for the exponent, and one more for the negative sign (only when the number is negative). A value of 123.45 would take 3 + 1 + 0, or 4 bytes. A value of –123.45 would take 3 + 1 + 1, or 5 bytes.
`VARCHAR2(x)` `VARCHAR(x)`	If the length is 250 or less, the column size will be x+1; otherwise, it will be x+3.
`DATE`	Date columns always consume 7 bytes.
`LONG` `LONGRAW` `RAW`	Same as for the `VARCHAR2` datatype.
`NVARCHAR2(x)`	If x is in bytes, the calculation is the same as for the `VARCHAR2` datatype. If x represents some number of multibyte characters, then you have to multiply by the number of bytes per character.
`NCHAR(x)`	If x is in bytes, the calculation is the same as for the `CHAR` datatype. If x represents some number of multibyte characters, then you have to multiply by the number of bytes per character.

Once you have the sum of the column sizes, you can estimate the space required for each row using this formula:

```
ROWSIZE = MAX(UB1 * 3 + UB4 + SB2, (3 * UB1)
          + SUM_OF_COLUMN_SIZES)
```

The MAX function is an Oracle function that returns the larger of two values. For example, MAX(4, 5) returns a value of 5.

Estimating the Number of Blocks Required

The last step, once you've computed the available space per block and the row size, is to compute the number of blocks required for the number of rows that you expect to store. The calculation for this is:

```
BLOCKS_REQUIRED = CEIL(ROWS_TO_STORE
                  / FLOOR(SPACE_FOR_DATA / ROWSIZE)
```

The FLOOR function is similar to CEIL, but it rounds any fractional value down to an integer value. For example, FLOOR(9.9) returns a value of 9.

Estimating the SALAD_TYPE Table's Row Space Requirement

Returning to the SALAD_TYPE table scenario described earlier, let's work through the process of estimating the space required by 10,000 rows. We'll use the type sizes listed earlier, in the section "Collecting prerequisite information." The columns in the SALAD_TYPE table are defined like this:

```
SALAD_TYPE        VARCHAR2(10)
DESCRIPTION_TEXT  VARCHAR2(40)
ORGANIC_FLAG      VARCHAR2(3)
PRICE_PER_POUND   NUMBER(10,2)
LAST_CHANGE_DATE  DATE
```

Assuming a 2KB block size, and assuming that default values are used for PCTFREE and INITRANS, the calculation to find the space in each block that is available for data is as follows:

```
SPACE_FOR_DATA = CEIL((2048 - 20 - 4 -48
- ((1 - 1) * 24) - 14)
* (1-20/100)) - 4
```

This works out to 1,566 bytes per block. The next step is to compute an average row size. Let's assume that the SALAD_TYPE and DESCRIPTION_TEXT fields will, on average, be 50 percent full. For the remaining fields, we will use the maximum size as the basis for our estimate. The following calculations demonstrate how the column lengths are computed:

SALAD_TYPE	Average of 5 bytes used, plus one more for the length, yields a total of 6 bytes.

`DESCRIPTION_TEXT`	Average of 20 bytes used, plus one more for the length, yields a total of 21 bytes.
`ORGANIC_FLAG`	Three bytes, plus one for the length, yields a total of 4 bytes.
`PRICE_PER_POUND`	Ten digits ÷ 2 per byte = 5 bytes. Add one for the exponent to get a total of 6 bytes.
`LAST_CHANGE_DATE`	Dates are always 7 bytes.

The sum of the column sizes is 44 bytes. Plug that into the `rowsize` formula, and we get:

```
ROWSIZE = MAX(UB1 * 3 + UB4 + SB2, (3 * UB1)
          + 44)
```

The average rowsize works out to 47 bytes. The calculation to compute the space required for 10,000 rows then becomes:

```
BLOCKS_REQUIRED = CEIL(10000
                  / FLOOR(1566 / 47)
```

Work out the math, and you'll find that the estimated number of blocks required to hold 10,000 rows of data is 304. That's fairly close to, but a bit lower than, the 325 blocks that were actually used for 10,000 rows earlier in the chapter when the extrapolation method was used. If you look back, though, you'll find that those rows had an average length of 54 bytes. Plug that value into the previous formula, instead of 47, and the estimate works out to be 345 blocks. That almost exactly matches the earlier extrapolation estimate based on 1,000 representative rows.

Remember that an estimate is just that, an estimate. Don't expect reality to exactly match your estimates. The best that you can hope for is that your estimated space requirements will be just a bit higher than what is actually required. That way, you don't waste too much space, and you don't have to create more extents than originally planned.

Estimating the storage space for an index

The process for estimating the space requirements for an index is similar to that for a table. The calculations are different because the structure of an index differs from that of a table, but you still follow the same basic steps:

1. Figure out how much space in each database block is available for index data.
2. Determine an average index entry size on which to base your estimate.
3. Compute the number of blocks required based on the previous two values.

You can use the calculations shown in the following sections to estimate the space required for the traditional B*Tree index that is most often used in an Oracle database.

Estimating the Amount of Available Space per Block

Oracle provides the following formula for use in estimating the available space in an index block:

```
AVAILABLE_SPACE = ((DB_BLOCK_SIZE - (113 + 24 * INITRANS))
- ((DB_BLOCK_SIZE - (113 + 24 * INITRANS))
* (PCTFREE / 100))
```

The default values for PCTFREE and INITRANS are the same for indexes as for tables. Use 20 for PCTFREE and 1 for INITRANS, if you don't plan to specify different values when creating the index.

Estimating the Entry Size

With a table, you need to calculate an average row size. For an index, you need to calculate the average size of an index entry. The first step in this process is to compute the sum of the column lengths for all columns involved in the index. Table 6-1 can help with this.

Once you have summed the column lengths, you can use the following formula to estimate the size of an index entry:

```
ENTRY_SIZE = 2 + 10 + SUM_OF_COLUMN_SIZES
```

When you are computing the column sizes for columns in the index, take into account the average amount of space that you expect to be used. If you are indexing a VARCHAR(40) field, but you expect the average entry to be only 20 characters long, you will get a more accurate estimate using 20 for the size.

Estimating the Number of Blocks Required

After you've estimated the available space and the average entry size, compute the number of blocks required using the following formula:

```
BLOCKS_REQUIRED = CEIL(1.05 * (NUM_ROWS
                  / FLOOR (AVAILABLE_SPACE / ENTRY_SIZE)))
```

In this formula, NUM_ROWS is the number of rows that you expect to be indexed. If you are indexing columns that are allowed to be null, allow for that when estimating the number of rows. For example, if your table will contain 10,000 rows, but you expect 2,000 rows to have null values for the indexed fields, then use a value of 8,000 for NUM_ROWS.

Estimating the SALAD_TYPE_PK Index's Block Size Requirement

Return one more time to the SALAD_TYPE scenario used earlier. This time, we'll estimate the number of blocks required for an index on the SALAD_TYPE column. The SALAD_TYPE column is defined as follows:

```
SALAD_TYPE         VARCHAR2(10)
```

Assuming a 2KB block size, and assuming the default values of 20 for PCTFREE and 1 for INITRANS, the following calculation returns the available space in each index block:

```
AVAILABLE_SPACE = ((2048 - (113 + 24 * 1))
- ((2048 - (113 + 24 * 1))
* (20 / 100))
```

In this case, the available space works out to 1,529 bytes per block. In computing the average entry size, assume that the SALAD_TYPE FIELD will be 75 percent full — say seven characters on average. Add one byte for the length, and you have 8 bytes for the column length. Plug that value into the entry size formula, and you get the following:

```
ENTRY_SIZE = 2 + 10 + 8
```

This works out to an average entry size of 20 bytes. You can now obtain an estimate of the number of blocks required by using this formula:

```
BLOCKS_REQUIRED = CEIL(1.05 * (10000
                  / FLOOR (1528 / 20)))
```

This finally produces an estimate of 139 blocks for an index on the SALAD_TYPE field.

Summary

In this chapter, you learned:

- Oracle manages space in terms of blocks, extents, segments, and tablespaces.
- An Oracle block is the smallest unit of space managed by Oracle.
- When Oracle allocates space to an object, it allocates a group of contiguous blocks. Such a group is referred to as an extent.
- Tablespaces are the logical containers for database objects. Each tablespace uses one or more physical files to store its data.

- You can use the `DEFAULT STORAGE` clause when creating a tablespace to specify default storage attributes for objects stored within that tablespace. When creating an object, you can use the `STORAGE` clause to override the default storage attributes.
- The `DBA_EXTENTS` and `DBA_FREE_SPACE` views are the keys to understanding how the space within a database has been allocated.
- There are three basic methods for estimating the size of a database object: You can make an educated guess; you can extrapolate from a small sample; or you can calculate an estimate using a set of formulas.

✥ ✥ ✥