14

CHAPTER

Managing Indexes

✦ ✦ ✦ ✦

In This Chapter

✦ ✦ ✦ ✦

Indexes are structures that allow you to quickly access rows in a table based on the value of one or more columns in that table. You use indexes for two primary purposes:

- **Faster queries.** Indexing columns frequently referenced in queries helps Oracle retrieve data at maximum speed.
- **Unique values.** Oracle automatically generates an index to enforce unique values in the primary key of a table. You can also use this feature for any other column (or set of columns) that you require to be unique.

Like anything else in life, using an index involves some tradeoffs. When you create an index, you are trading disk space and increased insert, update, and delete time in return for faster query response. Indexes require overhead during inserts and updates because the index needs to be updated in response to new rows, or to changes in indexed columns in existing rows. Even when deleting a row, you must take the time to remove that row's entry from the index.

Indexes also require disk space. The columns to be indexed must be reproduced as part of the index. If you have many indexes on a table, it's possible that the total space the indexes use will exceed the space the table itself uses.

In this chapter, you'll read about the different types of indexes that Oracle supports. You'll see how to create them, how to reorganize them, and how to drop them. You'll also learn about the data dictionary views that you can use to retrieve information about indexes.

Understanding Index Types

Oracle8i supports two fundamental types of indexes: B*Tree indexes and bitmapped indexes. B*Tree indexes have been around for years, and they are the most commonly used indexes. Bitmapped indexes are relatively new, and they were introduced as a data-warehousing feature.

Using B*Tree indexes

The most common type of index used is the B*Tree index, which uses an inverted tree structure to speed access to rows in a table. Figure 14-1 illustrates this structure.

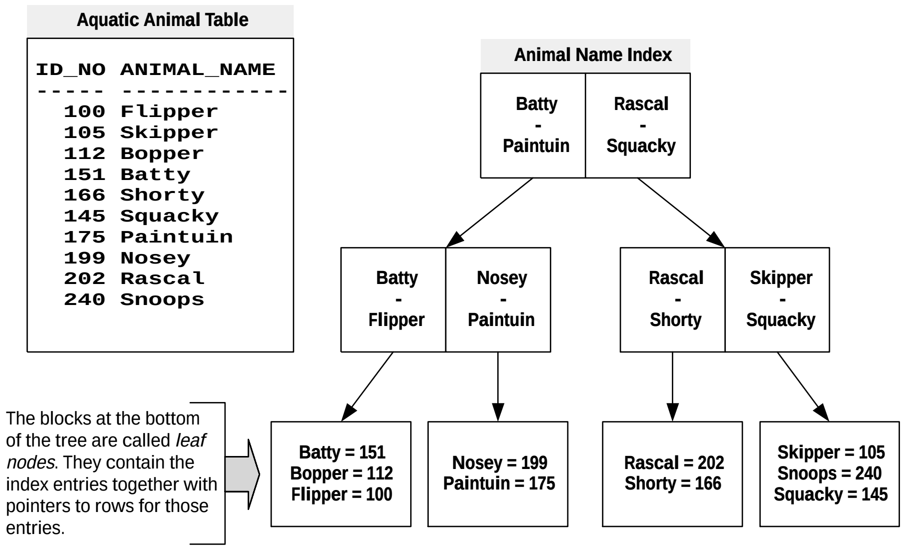

Aquatic Animal Table
ID_NO ANIMAL_NAME
----- -----------
100 Flipper
105 Skipper
112 Bopper
151 Batty
166 Shorty
145 Squacky
175 Paintuin
199 Nosey
202 Rascal
240 Snoops
Animal Name Index
Batty - Paintuin
Rascal - Squacky
Batty - Flipper
Nosey - Paintuin
Rascal - Shorty
Skipper - Squacky
Batty = 151 Bopper = 112 Flipper = 100
Nosey = 199 Paintuin = 175
Rascal = 202 Shorty = 166
Skipper = 105 Snoops = 240 Squacky = 145
The blocks at the bottom of the tree are called *leaf nodes*. They contain the index entries together with pointers to rows for those entries.

Figure 14-1: An example of the inverted tree structure used by B*Tree indexes

When you query for a particular value, Oracle navigates down through the tree to the leaf node containing the entry you are after. That entry contains a pointer to the row in the table. B*Tree indexes have several characteristics that make them a good choice for most uses:

- They maintain the sort order of the data, making it easy to look up a range of records. For example, you might want to find all animals whose names begin with "S."
- With multicolumn indexes, you can use the leading-edge columns to resolve a query, even if that query doesn't reference *all* columns in the index.

- They automatically stay balanced, at least they are supposed to, with all leaf nodes at the same depth, so the time you need to retrieve one entry is consistent for all entries in the index.
- Performance remains relatively constant, even as the size of the indexed table grows.

Within the B*Tree structure, you have some flexibility in how the index operates. When you create a B*Tree index, you can choose from among these two options:

`REVERSE`	Reverses the bytes in each index entry. The name Justin, for example, can be indexed as nitsuJ. If Oracle Parallel Server is being used, and primary keys are being generated from a numeric sequence, reversing the index can help prevent multiple instances from competing for the same index blocks.
`UNIQUE`	Requires that each index entry be unique. This also inherently prevents two rows in the underlying table from having the same value for the indexed columns. Oracle normally uses unique indexes to enforce primary key constraints.

The `REVERSE` option is compatible with `UNIQUE`. The default, if you choose neither option, is for the index to be stored unreversed, and for it to allow multiple entries for the same value.

Using bitmapped indexes

Bitmapped indexes were introduced in Oracle8. You can use them to index columns that contain a relatively small number of distinct values. Bitmapped indexes always contain one entry for each row in the table. The size of the entry depends on the number of distinct values in the column you are indexing. Figure 14-2 shows a conceptual view of two bitmapped indexes: In one, you see Yes and No columns, and in another, you see state abbreviation columns.

Bitmapped indexes are very compact because they are composed of long strings of bits. The bit-string for one value in a million-row table would take up only around 122KB. Oracle can quickly scan that bit-string for rows that match the criteria specified in a query. Bitmapped indexes really shine when you can use several together to resolve a query. Consider the following `SELECT` statement:

```
SELECT *
FROM employee
WHERE retired = 'Y'
AND state = 'MI';
```

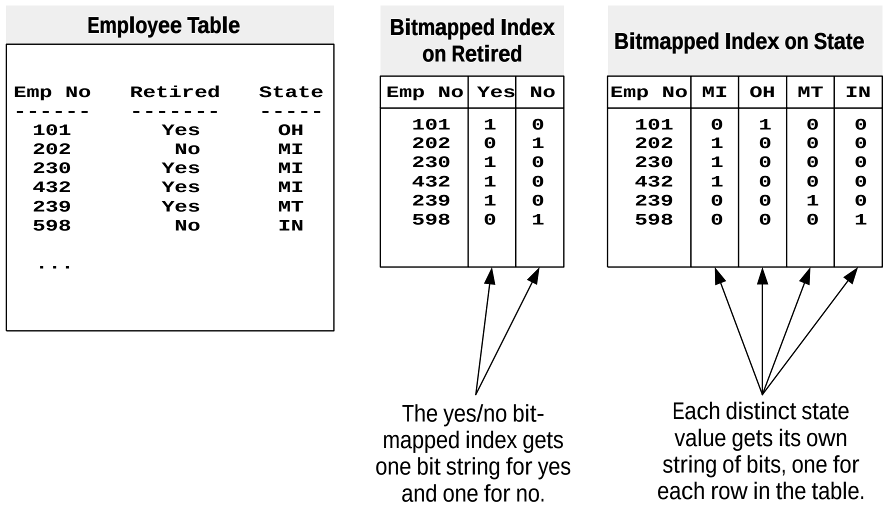


Figure14-2: A conceptual view of a bitmapped index

With bitmapped indexes on both the `RETIRED` column and the `STATE` column, as shown in Figure 14-2, Oracle can easily retrieve the bitmaps for Y and for MI, and then *and* them together. Oracle *ands* two bitmaps by merging them into one bitmap that flags rows that meet the condition represented by both of the original bitmaps. Table 14-1 provides an example.

Table 14-1
Anding Two Bitmaps Together

RETIRED = Y	*STATE = MI*	*Anded Result*
1	0	0
0	1	0
1	1	1
1	1	1
1	0	0
0	0	0

Anding two bit streams is a logic operation that most computers can do extremely quickly. Oracle can then use the resulting bitmap to identify those rows that match all the criteria.

As great as bitmapped indexes are, they do have some disadvantages. They were designed for query-intensive databases, so you shouldn't use them in online transaction processing (OLTP) systems. They aren't good for range queries, and you shouldn't use them for columns that contain more than just a few distinct values. Creating a bitmapped index on a name column would be a poor choice. The greater the number of distinct values in a column, the less efficient bitmapped indexing becomes.

Note Bitmapped indexes are available only in Oracle Enterprise Edition.

Creating an Index

Indexes are created using the `CREATE INDEX` statement. When you create an index, you need to specify the table on which the index is to be built. After identifying the table, you should also consider the following questions:

- Which columns should be part of the index?
- Should the index be B*Tree or bitmapped?
- Should the index be unique?
- Should the `REVERSE` option be used?
- In what tablespace should the index be stored?

Once you have answered these questions, you are ready to create the index using Schema Manager, or by executing a `CREATE INDEX` command from SQL*Plus.

Choosing the columns for an index

Most of the time, you choose the columns for an index because you are using them in the `WHERE` clause of queries that you write. However, there are times when you might want to add columns to an index that aren't used in a `WHERE` clause, or when you want to exclude some that are. Suppose that you have the following table, which contains both a first and last name column, in your database:

```
CREATE TABLE employee (
   last_name       VARCHAR2(30),
   first_name      VARCHAR2(30),
   department_no   NUMBER
);
```

Further suppose that you are running an application that frequently queries for employees based on their first and last names. Perhaps the application uses a query like this:

```
SELECT last_name, first_name, department_no
FROM employee
WHERE last_name = 'Burgers'
AND first_name = 'Jenny';
```

At first glance, you might think to create an index on both the first and last name columns. However, if you think more about it, you may have only a few employees with any given last name. By creating the index on last name only, you cut the space required by the index in half. The tradeoff is increased I/O. If three people have the same last name, Oracle will need to follow each of the three index entries to the table, retrieve the row, and then check the first name. Sometimes this tradeoff is worth making, and sometimes it isn't. You may need to experiment a bit to be sure.

Rather than cutting fields from an index, another strategy is to add more fields than are required. Let's assume that your application allows queries based only on the last name, and that it always issues SQL statements like this:

```
SELECT last_name, first_name, department_no
FROM employee
WHERE last_name LIKE 'Nue%';
```

Further, assume that once this query is executed, the name and department number of each matching employee will be listed on a screen from which you can choose the specific employee that you want. Which index would you create to support this query? The obvious answer is to create an index on last name. However, you might gain some efficiency in terms of response time if you create the following index instead:

```
CREATE INDEX emp_name_dept ON employee (
   last_name,
   first_name,
   department_no
);
```

Why place all three columns in the index when the query bases its search on only one? Doing so allows Oracle to retrieve from the index all the data that the query requires. Instead of reading each index entry and following it back to the table to get the first name and department number, Oracle can just get all three values from the index. The tradeoff here is more disk space for the index in return for the potentially faster response time.

Tip No hard and fast rules exist for making these types of choices. The best strategy is to develop some relevant metrics, measure the performance that you get with different types of indexes, and compare that to the disk space that they use. Base your decision on those numbers.

Note

You cannot index `LONG` or `LONG RAW` columns.

Choosing the index type

You've already read about the differences between bitmapped and B*Tree indexes. You've also learned about reversed indexes and unique indexes. Follow these guidelines to make the appropriate choice from among those options:

- ✦ Use unique indexes only when you need to enforce a business rule requiring unique values in a column.
- ✦ Consider the `REVERSE` option when you have large numbers of index entries all starting with the same characters and when reversing those characters would eliminate that clustering. The classic example of when to use a reversed index is when you are indexing a numeric column that is sequentially incremented each time a row is added to the database.
- ✦ Do not use the `REVERSE` option if you are querying for ranges of data. The reversed index entries randomize the location of entries. To find a specific range of values requires the entire index to be scanned.
- ✦ Consider bitmapped indexes for columns containing a low number of distinct values — a Yes or No type of column, for example — and where the primary use of the table is for query purposes (that is, few updates).

If you are uncertain about which choice to make, stick with the plain-vanilla, nonunique, nonreversed B*Tree index.

Using SQL to create an index

The SQL statement for creating an index is `CREATE TABLE`. You can issue it from SQL*Plus, Server Manager, or any other suitable tool. The following example shows a standard B*Tree index being created on the `CHECKUP_HISTORY` table. The `UNIQUE` keyword is used because two checkups on the same date should really be treated as one.

```
CREATE UNIQUE INDEX animal_checkup_history
ON checkup_history (
   id_no,
   checkup_date
) TABLESPACE indx;
```

If you were creating this index, and you didn't want it to be unique, you would just leave the keyword `UNIQUE` out of the command. To make it a reverse index, you can add the keyword `REVERSE` immediately following the tablespace name.

The CHECKUP_TYPE field in the CHECKUP_HISTORY table is a good candidate for a bitmapped index. This field is a foreign key to the CHECKUP table, and contains only the three distinct values ANNUAL, MONTHLY, and DAILY. You can create a bitmapped index on that column using this command:

```
CREATE BITMAP INDEX checkup_history_type
ON checkup_history (
   checkup_type
) TABLESPACE indx;
```

Bitmapped indexes may not be unique, so you can't combine the two types.

The NOLOGGING Option

The NOLOGGING option specifies to create the index without logging the index creation work to the database redo log files. The NOLOGGING option goes into the CREATE INDEX command, as shown in this example:

```
CREATE UNIQUE INDEX animal_checkup_history
ON checkup_history (
   id_no,
   checkup_date
) TABLESPACE indx
  NOLOGGING;
```

The advantage of using the NOLOGGING option is greater speed, which results from not having to write the index data out to the redo log. The risk you are taking is that if you lose some of your database files because of a disk failure and you have to recover those files, the index won't be recovered. That's a pretty small risk, because you can always create the index again. You just have to remember to do that.

The NOSORT Option

If you've just loaded a large table, say using SQL*Loader, and you know for a fact that all the rows were presorted on the same columns that you are indexing, you can create the index using the NOSORT option. Consider this example:

```
CREATE UNIQUE INDEX animal_checkup_history
ON checkup_history (
   id_no,
   checkup_date
) TABLESPACE indx
  NOSORT;
```

The NOSORT option does exactly what its name implies. Normally, when you create an index, Oracle sorts the table based on the index columns. When you use NOSORT, Oracle skips the sort step and assumes that the data in the table is ordered correctly to start with. Needless to say, skipping the sort can save you significant time. If, while creating the index, Oracle discovers that the rows in the table are in fact not presorted, the operation will abort, and you will have to create the index in the usual manner.

The ONLINE Option

Normally, when you create an index, Oracle places a lock on the table being indexed to prevent users from changing any of the data within that table. As you might imagine, if the indexing operation takes any amount of time, the users of that table will be inconvenienced. Oracle8i implements a new feature allowing you to create indexes in a way that leaves the table available for use. To take advantage of this feature, add the ONLINE keyword to your CREATE command, as shown here:

```
CREATE UNIQUE INDEX animal_checkup_history
ON checkup_history (
   id_no,
   checkup_date
) TABLESPACE indx
  ONLINE;
```

You can't create all indexes while the table remains online. Whether you can use ONLINE depends on the exact mix of options that you are using when you create the index.

The COMPUTE STATISTICS Option

If you are using the cost-based query optimizer, it's important to maintain current statistics on database objects such as tables and indexes. Index statistics provide Oracle with information on the number of distinct index entries, the total number of leaf blocks, the average number of data blocks per key, and so forth. This information is critical to the optimizer, relative to determining the most efficient way to execute a query.

Cross-Reference You'll learn more about how the optimizer works in Chapter 19, "Tuning SQL Statements."

Index statistics aren't maintained automatically. You have to periodically regenerate them using ANALYZE TABLE commands.

Cross-Reference Chapter 18, "Optimizers and Statistics," talks more about the importance of doing this.

Generating statistics via the ANALYZE command uses a significant amount of CPU and I/O resources. You end up reading through either the entire index or at least a significant part of it. When you create a new index, however, you have a perfect opportunity to sidestep this overhead by combining the generation of the statistics with the creation of the index. To do that, use the COMPUTE STATISTICS keyword in your CREATE command, as shown here:

```
CREATE UNIQUE INDEX animal_checkup_history
ON checkup_history (
   id_no,
   checkup_date
) TABLESPACE indx
  COMPUTE STATISTICS;
```

There is one disadvantage to the `COMPUTE STATISTICS` option, and that is that you can't mix it with the `ONLINE` option.

Using function-based indexes

Function-based indexes are an exciting new feature in Oracle8i. Rather than indexing based just on the column contents, you can create an index on the result of an expression applied to the column. One use for this is to create indexes to support case-insensitive queries. The following command creates a function-based index on the `ANIMAL_NAME` column of the `AQUATIC_ANIMAL` table:

```
CREATE INDEX animal_name_upper
ON aquatic_animal (
   UPPER(animal_name)
) TABLESPACE indx
  NOLOGGING;
```

Note

To create function-based indexes, you must have the `QUERY REWRITE` system privilege.

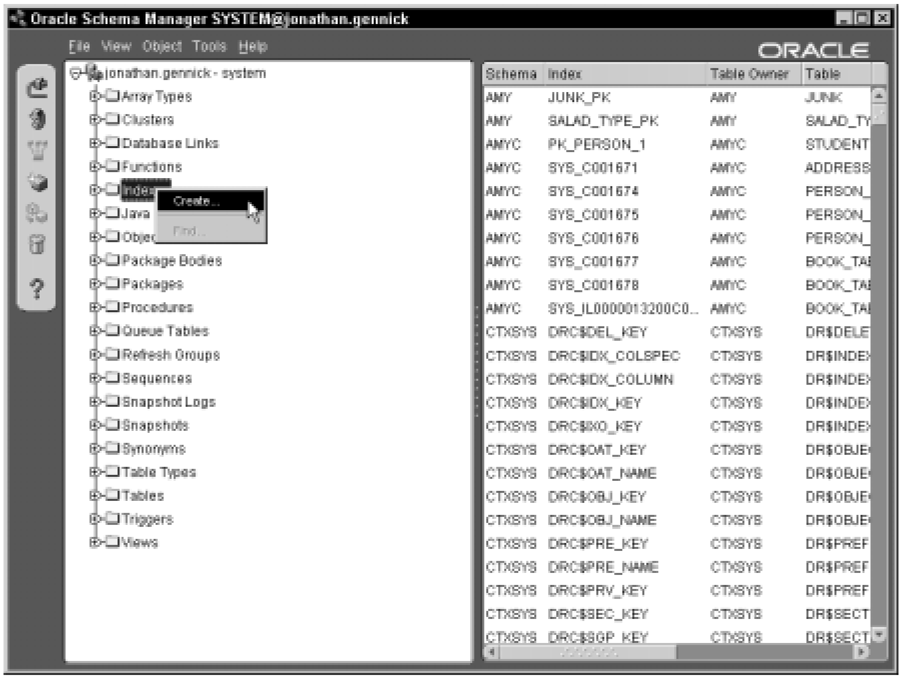
Once you've created an index such as the one shown previously, you can place the function call `UPPER(animal_name)` in the `WHERE` clause of a `SELECT` statement, and Oracle will be able to use the function-based index `ANIMAL_NAME_UPPER` to resolve the query. Without a function-based index, wrapping a function around a column name in the `WHERE` clause of a query would prevent any index on that column from ever being used.

Using Enterprise Manager to create an index

If you have Enterprise Manager installed, you can use Schema Manager to create your indexes. Schema Manager provides a convenient graphical user interface. To create an index using Schema Manager, follow these steps:

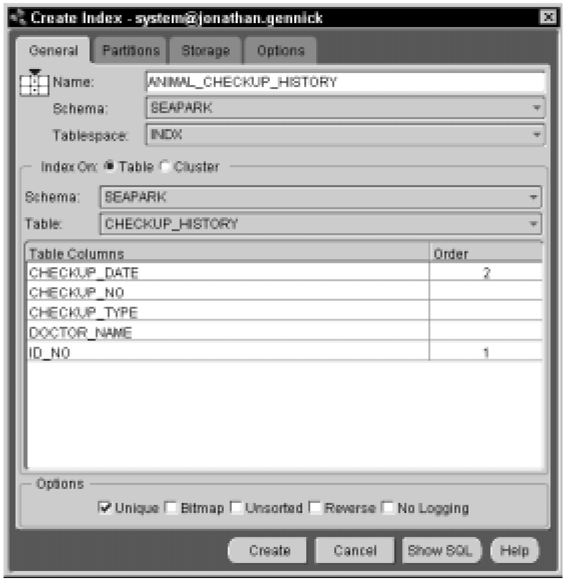
1. Start Schema Manager and log on to the database.
2. Right-click the Indexes folder and select Create from the pop-up menu. See Figure 14-3.
3. Fill in the Create Index form that appears.

   Figure 14-4 shows you a completed General tab in the Create Index dialog box to create the `ANIMAL_CHECKUP_HISTORY` index shown in the previous section.
4. Click the Create button.



Schema	Index	Table Owner	Table
AMY	JUNK_PK	AMY	JUNK
AMY	SALAD_TYPE_PK	AMY	SALAD_TY
AMYC	PK_PERSON_1	AMYC	STUDENT
AMYC	SYS_C001671	AMYC	ADDRESS
AMYC	SYS_C001674	AMYC	PERSON_
AMYC	SYS_C001675	AMYC	PERSON_
AMYC	SYS_C001676	AMYC	PERSON_
AMYC	SYS_C001677	AMYC	BOOK_TA
AMYC	SYS_C001678	AMYC	BOOK_TA
AMYC	SYS_IL0000013200C0...	AMYC	BOOK_TA
CTXSYS	DRC$DEL_KEY	CTXSYS	DR$DELE
CTXSYS	DRC$IDX_COLSPEC	CTXSYS	DR$INDEX
CTXSYS	DRC$IDX_COLUMN	CTXSYS	DR$INDEX
CTXSYS	DRC$IDX_KEY	CTXSYS	DR$INDEX
CTXSYS	DRC$IXO_KEY	CTXSYS	DR$INDEX
CTXSYS	DRC$OAT_KEY	CTXSYS	DR$OBJE
CTXSYS	DRC$OAT_NAME	CTXSYS	DR$OBJE
CTXSYS	DRC$OBJ_KEY	CTXSYS	DR$OBJE
CTXSYS	DRC$OBJ_NAME	CTXSYS	DR$OBJE
CTXSYS	DRC$PRE_KEY	CTXSYS	DR$PREF
CTXSYS	DRC$PRE_NAME	CTXSYS	DR$PREF
CTXSYS	DRC$PRV_KEY	CTXSYS	DR$PREF
CTXSYS	DRC$SEC_KEY	CTXSYS	DR$SECT
CTXSYS	DRC$SGP_KEY	CTXSYS	DR$SECT

Figure 14-3: Selecting the Create option to create a new index



Name: ANIMAL_CHECKUP_HISTORY
Schema: SEAPARK
Tablespace: INDX
Index On: [✓ Table] Cluster
Schema: SEAPARK
Table: CHECKUP_HISTORY

Table Columns	Order
CHECKUP_DATE	2
CHECKUP_NO	
CHECKUP_TYPE	
DOCTOR_NAME	
ID_NO	1

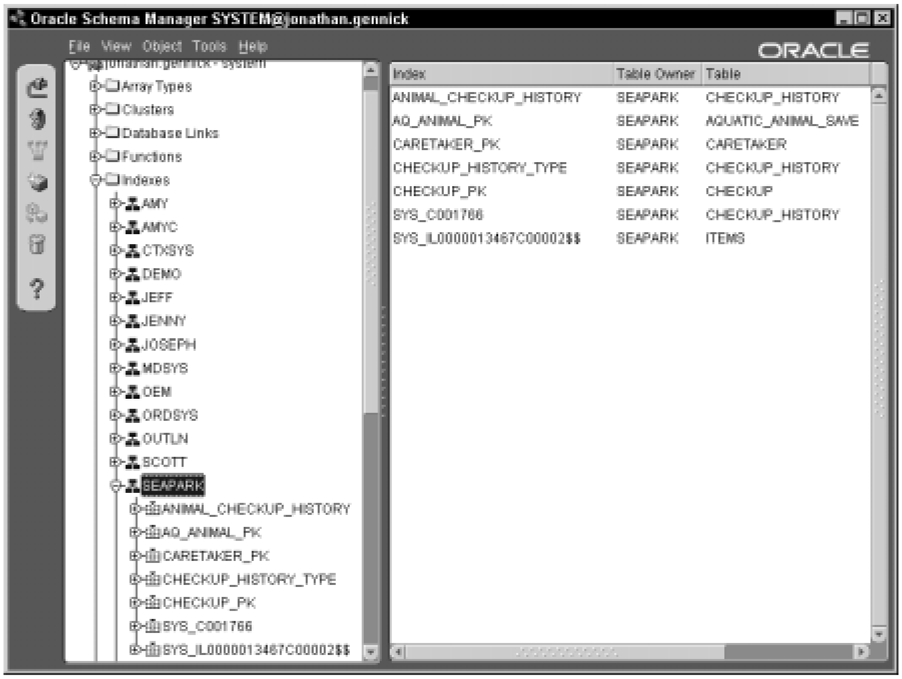
Options: [✓ Unique] Bitmap Unsorted Reverse No Logging

Figure 14-4: Filling in the information for a new index

When you're selecting the columns to be included in an index, click those columns in the order in which you want them to appear. If you click the wrong column, or if you click a column out of order, just click the column again to deselect it. As you click each column to be included in the index, Schema Manager displays the order on the right-hand side of the Create Index dialog box.

Browsing indexes

You can easily browse indexes using Schema Manager and quickly get a look at the definitions for any that interest you. Do this by expanding the Indexes folder and drilling down to the schema of interest. Figure 14-5 illustrates using Schema Manager to list the indexes in the SEAPARK schema.



Index	Table Owner	Table
ANIMAL_CHECKUP_HISTORY	SEAPARK	CHECKUP_HISTORY
AQ_ANIMAL_PK	SEAPARK	AQUATIC_ANIMAL_SAVE
CARETAKER_PK	SEAPARK	CARETAKER
CHECKUP_HISTORY_TYPE	SEAPARK	CHECKUP_HISTORY
CHECKUP_PK	SEAPARK	CHECKUP
SYS_C001766	SEAPARK	CHECKUP_HISTORY
SYS_IL0000013467C00002$$	SEAPARK	ITEMS

Figure 14-5: Listing the SEAPARK indexes

If a particular index interests you, click that index, and Schema Manager displays the details in the right pane of the window.

Altering an Index

You can change an index after you create it, but for the most part, you are limited to changing only its physical characteristics. You can't add columns to an index. To do that, you would have to drop the index and re-create it. However, Oracle does allow you to make the following changes:

- Move an index to another tablespace
- Rebuild an index using different storage parameters
- Rename an index
- De-allocate unused space

Many of the changes that you can make, such as relocating an index to another tablespace, actually require the complete re-creation of the index. However, Oracle automates the process and can often keep the index online and usable while the rebuild proceeds.

Note

When Oracle rebuilds an index, it does so using only the information in the index. If the index is corrupt (missing a value or containing a pointer to a nonexistent row), the corruption will still exist in the rebuilt index.

Using SQL to alter an index

You can use the `ALTER INDEX` command to change an index. To rename an index, for example, issue a command like the following:

```
ALTER INDEX animal_checkup_history RENAME TO an_chk_hist;
```

You can make many of the physical changes by using the `REBUILD` clause. When you rebuild an index, Oracle re-creates the index from scratch by using a new set of storage parameters. The following example rebuilds the `animal_checkup_history` index by changing a number of parameters and settings:

```
ALTER INDEX animal_checkup_history REBUILD
   TABLESPACE users
   ONLINE
   NOLOGGING
   STORAGE (INITIAL 5K NEXT 5K);
```

Let's go through these clauses one by one:

- `ALTER INDEX animal_checkup_history REBUILD`—Specifies that you want to rebuild the index named `an_chk_hist`.
- `TABLESPACE users`—Creates the new version of the index in the tablespace named `USERS`.
- `ONLINE`—Specifies that you want users to be able to access the table while the rebuild is in progress.

Note

You can't rebuild an index online if you are also using the `COMPUTE STATISTICS` or `REVERSE` clause. Bitmapped indexes cannot be rebuilt online either.

✦ `NOLOGGING` — Prevents Oracle from recording the rebuild in the redo log. This will speed the rebuild process (less I/O), but if you ever need to recover the database from a backup and roll forward through the changes, the index rebuild will not be included. Instead, you may need to manually rebuild the index again.

✦ `STORAGE (INITIAL 5K NEXT 5K)` — Specifies to allocate an initial extent of 5KB for the index. Subsequent extents will also be 5KB each. These values override the default settings for the tablespace in which the index is being created.

Certain types of changes are not allowed when the `ONLINE` option is requested. For example, you can't compute statistics during an online rebuild. If you try to do a rebuild online that Oracle doesn't support, you will get the following error:

```
ORA-08108: may not build or rebuild this type of index online
```

If you get an ORA-08108 error, you will have to rebuild the index offline. This means that no one will be able to insert, update, or delete from the table until the rebuild is finished.

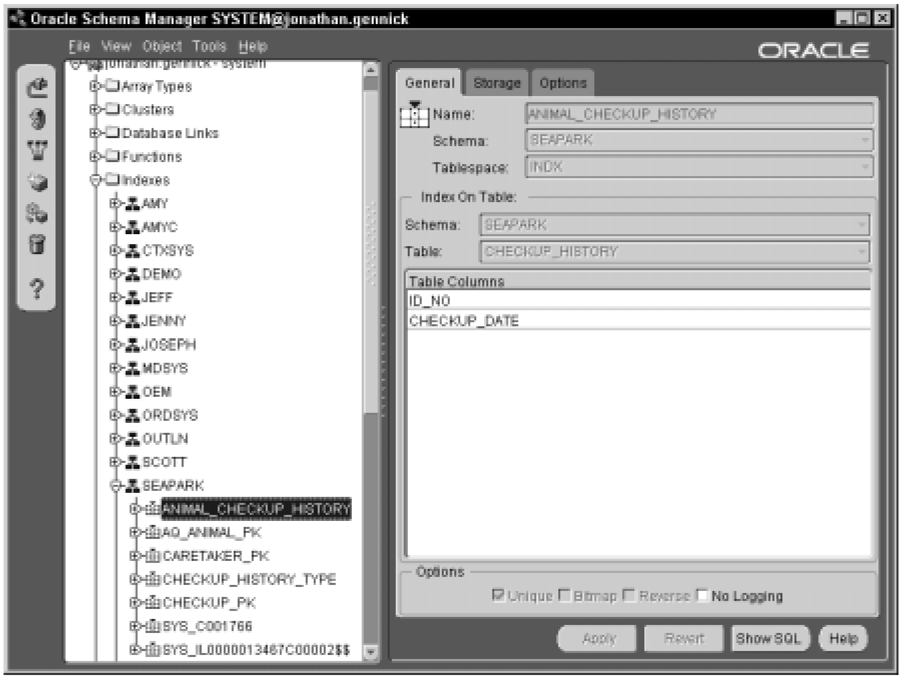
Using Enterprise Manager to alter an index

You can use Enterprise Manager's Schema Manager to modify an index using a GUI interface. Schema Manager builds the necessary `ALTER INDEX` statement based on information that you provide and then executes it for you. To alter an index using Schema Manager, follow these steps:

1. Start Schema Manager and log on to the database.
2. Open the Indexes folder and navigate to the index that you want to change. Figure 14-6 shows the Indexes folder open to the `ANIMAL_CHECKUP_HISTORY` index.
3. Right-click the index and select Edit from the pop-up menu.

   You will see the General tab in an Edit Indexes dialog box, which is similar to the Create Index dialog box shown in Figure 14-6.
4. Fill in the General tab's information.

   Figure 14-6 shows how you can fill in the information on this tab to create the `ANIMAL_CHECKUP_HISTORY` index shown in the previous section.
5. Go through the tabs and modify fields that you want to change. Dimmed fields indicate items that you can't change. Most of what you can change is on the Storage tab shown in Figure 14-7.
6. Click the Create button.



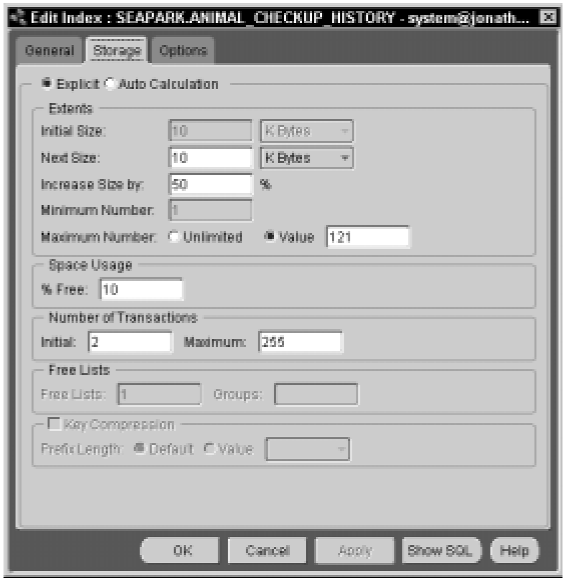
Name: ANIMAL_CHECKUP_HISTORY
Schema: SEAPARK
Tablespace: INDX

Index On Table:
Schema: SEAPARK
Table: CHECKUP_HISTORY

Table Columns
ID_NO
CHECKUP_DATE

Options
[✓ Unique] Bitmap Reverse No Logging

Figure 14-6: Selecting the ANIMAL_CHECKUP_HISTORY index


Edit Index : SEAPARK.ANIMAL_CHECKUP_HISTORY - system@jonath...
General Storage Options
OK Cancel Apply Show SQL Help

[✓ Explicit] Auto Calculation

Extents
Initial Size: 10 K Bytes
Next Size: 10 K Bytes
Increase Size by: 50 %
Minimum Number: 1
Maximum Number: Unlimited [✓ Value] 121

Space Usage
% Free: 10

Number of Transactions
Initial: 2 Maximum: 255

Free Lists
Free Lists: 1 Groups:

Key Compression
Prefix Length: [✓ Default] Value

Figure 14-7: Using Schema Manager to change an Index’s storage properties

One task that Schema Manager can't perform is to rebuild an index. To do that, you have to write the `ALTER INDEX REBUILD` statement yourself and submit it using SQL*Plus.

Deleting an Index

You can use the `DROP INDEX` command to delete an index from a database. Consider this example:

```
DROP INDEX animal_checkup_history;
```

In Schema Manager, you can drop an index by right-clicking on the index name and selecting Remove from the pop-up menu.

Listing Indexes

If you need to find out information about indexes, you can query Oracle's data dictionary. The data dictionary will tell you the names of the indexes in your database, the names of the columns that make up those indexes, and which options you chose to use when creating them. The following two data dictionary views are relevant to indexes:

`DBA_INDEXES`	Returns one row for each index in the database
`DBA_IND_COLUMNS`	Returns one row for each indexed column

If you don't have access to the DBA views, remember that you do have access to the `ALL` and `USER` views. The `ALL_INDEXES` view lists indexes on all tables to which you have access, and `USER_INDEXES` lists indexes that you own.

Listing indexes on a table

You can get a list of all the indexes defined for a table by querying `DBA_INDEXES`. The following query returns the index name and type and a flag indicating whether it is a unique index:

```
SELECT index_name, index_type, uniqueness
FROM DBA_INDEXES
WHERE table_owner = 'schema_name'
AND table_name = 'table_name';
```

If you don't have access to the DBA_INDEXES view, use ALL_INDEXES instead. The DBA_INDEXES view contains many more columns than those listed here. The contents of most columns are self-evident from the column names.

Listing the columns in an index

The DBA_IND_COLUMNS view, or ALL_IND_COLUMNS if you aren't the DBA, returns information about the columns that make up an index. The following query returns a list of columns for a given index:

```
SELECT column_name, column_position, descend
FROM dba_ind_columns
WHERE index_owner = 'schema_name'
AND index_name = 'index_name'
ORDER BY column_position;
```

You must order the results by COLUMN_POSITION to get them to list in the correct order. The DESCEND column tells you whether the index sorts a particular column in ascending order or descending order. Most indexes are sorted in ascending order.

Note The COLUMN_NAME field in DBA_IND_COLUMNS is defined as a VARCHAR2(4000). That's rather wide. If you are executing a query from SQL*Plus that includes that column, you may want to enter a command like COLUMN COLUMN_NAME FORMAT A30 to set the display width to a more reasonable value, which in this case would be 30 characters.

Summary

In this chapter, you learned:

- ✦ Indexes on a database function like an index in a book: They speed your access to the data in a table.
- ✦ Oracle supports two types of indexes: B*Tree and bitmapped. B*Tree indexes are structured in an inverse tree and are good all-around choices. Bitmapped indexes excel when you are querying columns containing a small number of distinct values (a Yes or No column, for example).
- ✦ Function-based indexes, a new feature in Oracle8i, allow you to create indexes based on the result of an expression applied to a column. Queries that use the same expression in their WHERE clause can take advantage of these indexes.
- ✦ The DBA_INDEXES view returns information about the indexes in a database. DBA_IND_COLUMNS tells you which columns make up an index.

✦ ✦ ✦