B

Indexes

This appendix is a discussion of the basics of index and table structures. Indexes are some of the most important objects involved in physical modeling. You use them to implement primary keys and alternate keys (unique constraints), and you can use them to improve performance.

Understanding how indexes work will give you a base level from which to determine what kinds of indexes to use. I'll only briefly cover how SQL Server implements indexes, as this is a complex topic outside the scope of this book.

Basic Index Structure

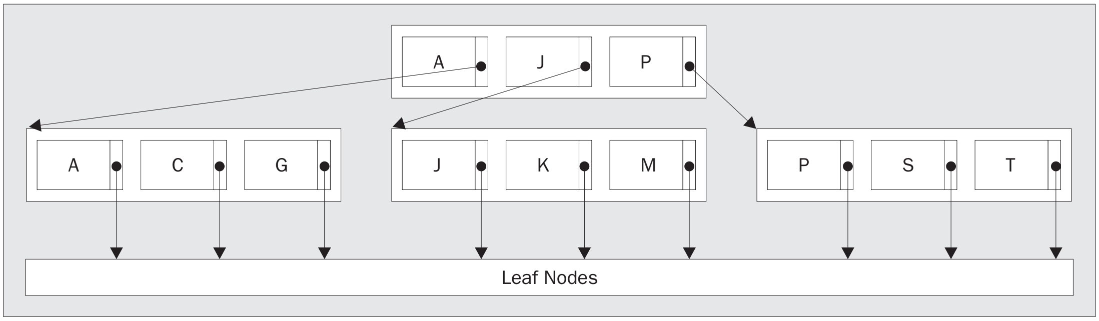
Indexes speed up retrieval of the rows in a table, and are generally, though not always, separate structures associated with the table. They are built by taking one or more of the table's columns and building a special structure on the table, based on those keys. There are two distinct classes of indexes:

- **Clustered:** Orders the physical table in the order of the index
- **Nonclustered:** Completely separate structures that simply speed access

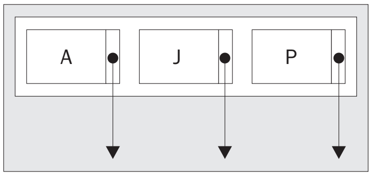
You can have unique type indexes of either class. Unique indexes are used not only to speed access, but also to enforce uniqueness over a column or group of columns. The reason that they are used for this purpose is that, to determine if a value is unique, you have to look it up in the table, and because the index is used to speed access to the data, you have the perfect match.

There are other settings, but they are beyond the scope of this book (consult the *Create Index* topic in SQL Server 2000 Books Online for a full explanation of creating indexes). Here, I'll only present the basic options that are germane to designing the physical structures. The additional settings affect performance as well as a few other features that are handy but beyond this book's scope.

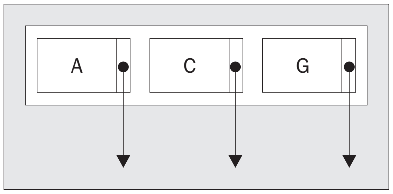
Indexes are implemented using a balanced tree structure, also known as a **B-tree**. In the following diagram, you see the basic structure of the type of B-tree that SQL Server uses for indexes.



Each of the outer rectangles is an 8KB index page, which SQL Server uses for all data storage. The first page is shown at the top of the diagram:



You have three values that are the index keys. To decide which path to follow to the lower level of the index, you have to decide if the value is between two of the keys: A to I, J to P, or greater than P. If the value is between A and I, you follow the first route down to the next level:



You continue this down to the leaf nodes of the index. You'll look at leaf nodes when I come to discuss the details of clustered and nonclustered indexes. The leaf node structure is how all SQL Server indexes are implemented. However, one important detail should be noted. I've stated that each of these pages is 8KB in size. Depending on the size of a key (determined by summing the data lengths of the columns in the key, up to a maximum of 900 bytes), it is possible to have anywhere from eight entries to over one thousand on a single page. Hence, these trees don't generally grow all that large, if designed properly. This example uses a simplistic set of index values to show how the index tree fans out.

In the next example, you see how to build a basic table and basic indexes. I'll use the same table in the rest of the examples in this section.

```
CREATE TABLE testIndex
(
   id int IDENTITY,
   firstName varchar(20),
   middleName varchar(20),
   lastName varchar(30)
)
CREATE INDEX XtestIndex ON testIndex(id)
CREATE INDEX XtestIndex1 ON testIndex(firstName)
CREATE INDEX XtestIndex2 ON testIndex(middleName)
CREATE INDEX XtestIndex3 ON testIndex(lastName)
CREATE INDEX XtestIndex4 ON testIndex(firstName,middleName,lastName)
```

If all is well, there will be no output from this statement. To drop an index, use the following statement:

```
DROP INDEX testIndex.Xtestindex
```

Clustered Indexes

The leaf nodes of the clustered index are the actual data pages. Hence, there can only be one clustered index per table. The exact implementation of the data pages isn't a particularly important detail; rather, it's sufficient to understand that the data in the table is physically ordered in the order of the index.

When choosing the fields to use as the basis for the clustered index, it's best to consider it as the humanly readable index. In other words, if the table has an inherent ordering that will make sense for a human to scan, this is a good candidate for the clustered index. Take, for example, a dictionary. Each word in the dictionary can be thought of as a row in a database. It is then clustered on the word. The most important reason for this is that you frequently need to scan a group of words and their meanings to see if you can find the one you need. Because all of the data is sorted by word, you can scan the words on the pages without any trouble. You use much the same logic to decide whether or not to use a clustered index.

Clustered indexes should be used for the following:

- **Key sets that contain a limited number of distinct values:** Because the data is sorted by the keys, when you do a lookup on the values in the index, all of the values you need to scan are visible straight away.
- **Range queries:** Having all of the data in order usually makes sense when there is data that you often need to get a range, like from A to F.
- **Data that is accessed sequentially:** Obviously, if the data needs to be accessed in a given order, having the data already sorted in that order will significantly improve performance.

- ❑ **Queries that return large result sets:** This point will make more sense once I cover the nonclustered index, but for now note that having the data on the leaf index node saves overhead.
- ❑ **Key sets that are frequently accessed by queries involving `JOIN` or `GROUP BY` clauses:** Frequently these may be foreign key columns, but not necessarily. `GROUP BY` clauses always require data to be sorted, so it follows that having data sorted anyway will improve performance.

Because the clustered index physically orders the table, inserting new values will usually force the new data into the middle of the table–the common exception to this is a clustered index with auto incrementing (identity) type fields. If the new row won't fit on the page, SQL Server has to do a **page split** (which is when a page gets full, so half of the data is split off to a new page–hence the name). Page splits are fairly costly operations and hurt performance because data won't be located on successive pages, eliminating the possibility of synchronous reads by the disk subsystem.

If you don't have a set of keys that meet one of the criteria for a clustered index, it's generally best to build the index on an autogenerated key, primarily to make the table clustered, rather than a heap (a concept I'll discuss late in this appendix). It's also important not to use a clustered index on frequently changing columns. As the table is ordered by the values in the column, SQL Server may have to rearrange the rows on their pages–a costly operation.

Another important point to note is that you need to keep the clustering key as small as possible, as it will be used in every nonclustered index (as you'll see). So, if the clustered index key is modified, additional work is required to maintain all nonclustered indexes as well.

For a clustered table, the row locator is always the clustered index key. If the clustered key is nonunique, SQL Server adds a random value to make it unique.

Nonclustered Indexes

The nonclustered index is analogous to the index in a textbook, with the book indexed on page number. Much like the index of a book, the nonclustered index is completely separate from the rows of data.

A leaf node of a nonclustered index contains only a pointer to the data row containing the key value. The pointer from the nonclustered index to the data row is known as a **row locator**. The structure of the row locator depends on whether the table has a clustered index or not. The upper levels of the index are the same as for the clustered index. You'll look at diagrams showing the different row locators after a bit more discussion on the basics of nonclustered indexes.

I've discussed tables with clustered indexes–these are referred to as **clustered tables**. If you don't have a clustered index, the table is referred to as a **heap**. My dictionary's definition of a heap is, "A group of things placed or thrown one on top of the other." This is a great way to explain what happens in a table when you have no clustered index:. SQL Server simply puts every new row on the end of the table. Note also that if you do a delete, SQL Server will simply pull out the row, not reusing the space. However, for updates, there are two possibilities:

- ❑ **Update in place:** The values are simply changed in the row.
- ❑ **Delete and insert:** The entire row is deleted from the table, and then reinserted.

There are several criteria that SQL Server uses to determine whether or not to perform an update on a row. Two of the more common ones include:

- ❑ **Having an after update trigger on the table:** When there is a trigger on the table that fires after the row is updated, the row is deleted first and then inserted.
- ❑ **Trying to insert more data than the page will handle:** When there is variable length data in the table, and then you increase the size of the data stored, SQL Server will have to delete the row and reinsert it somewhere else, perhaps on a different page. This procedure differs from when there is a clustered index, because there is now no reason to do a page split in the middle of the structure; it's faster and easier to move the data to the end of the heap.

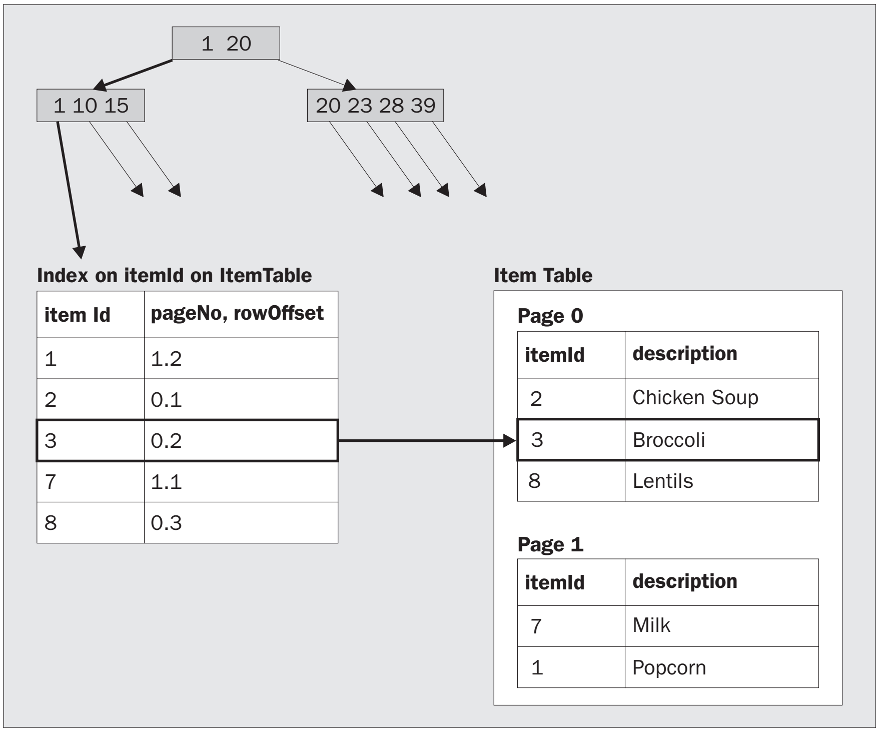
Next I'll show you the two distinct types of tables and how their nonclustered row locators are implemented.

Heap

For the "heap" table, with no clustered index, the row locator is a pointer to the page that contains the row.

Suppose that you have a table called `ItemTable`, with a clustered index on the `description` column, and a nonclustered index on the `itemId` column.

In the following diagram, you see that, to find `itemId 3` in the table, you take the left branch of the tree because 3 is between 1 and 20, then follow that down the path between 1 and 10. Once you get to the page, you get a pointer to the page that has the value. Assume this pointer consists of the page number and offset on the page (the pages are numbered from 0 and the offset is numbered from 1). The most important fact about this pointer is that it points directly to the row on the page that has the values that you are looking for. This is not how it will work for the clustered table, and it is an important distinction to understand.



Index on itemId on ItemTable

item Id	pageNo, rowOffset
1	1.2
2	0.1
3	0.2
7	1.1
8	0.3

Item Table

Page 0

itemId	description
2	Chicken Soup
3	Broccoli
8	Lentils

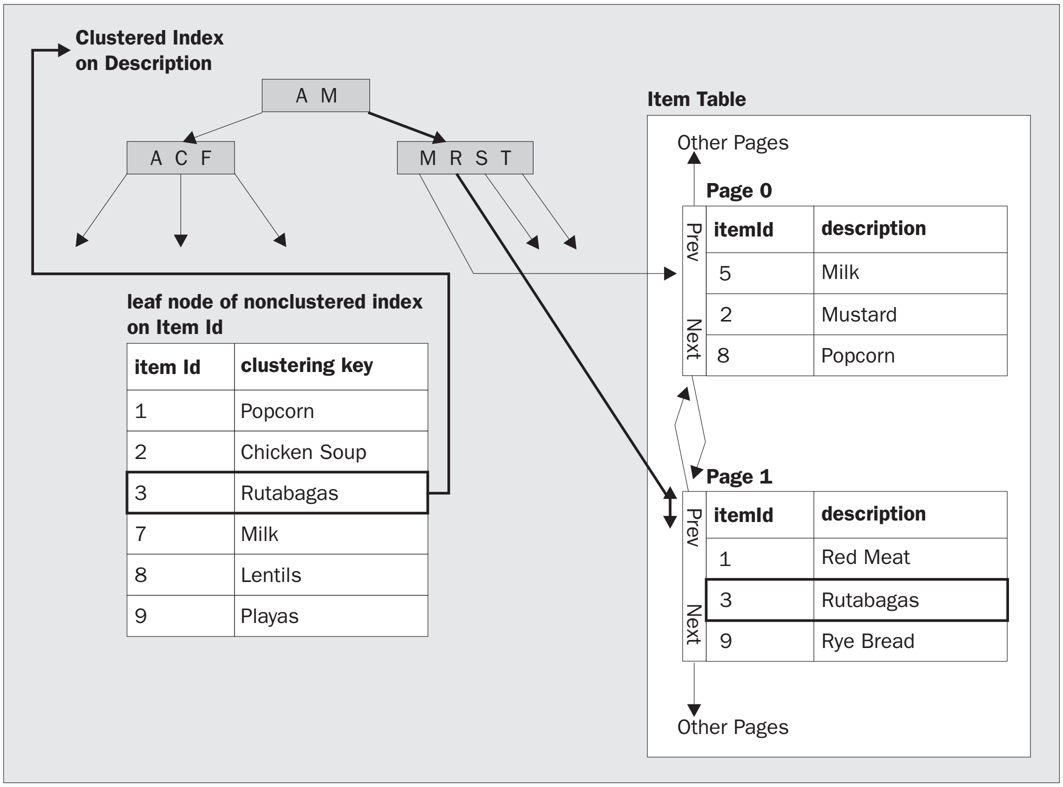
Page 1

itemId	description
7	Milk
1	Popcorn

Nonclustered Indexes on a Clustered Table

In this situation, when the *nonclustered index* has been traversed and arrived at its leaf node, instead of finding a pointer that points directly to the data, you must traverse the clustered index to retrieve the data.

In the next graphic, the lower-left box is the nonclustered index, as in the previous example, except that the pointer has changed from a specific location to the value of the clustering key. You then use the clustering key to traverse the clustered index to reach the data.



The overhead of this operation is minimal and it's "theoretically" better than having direct pointers to the table, because only minimal reorganization is required for any modification of the clustered index or the clustering key. When modifications are made, the pointers aren't moved. Instead, a pointer is left in the table that points to the new page where the data now is. All existing indexes that have the old pointer simply go to the old page, then follow the new pointer on that page to the new location of the data.

Unique Indexes

The unique index is the second most important thing that you set when implementing indexes. You can create unique indexes for clustered and nonclustered indexes alike. It ensures that, among the index's keys, there can't be any duplicate values. For example, if you created the following indexes on the test table:

```
CREATE UNIQUE INDEX XtestIndex1 ON testIndex(lastName)
CREATE UNIQUE INDEX XtestIndex2 ON testIndex(firstName,middleName,lastName)
```

you wouldn't be able to insert duplicate values in the `lastName` column, nor would you be able to create entries that have the same `firstName`, `middleName`, and `lastName` values.

You typically don't use unique indexes to enforce uniqueness of alternate keys. SQL Server has a mechanism known as a `UNIQUE` constraint, as well as a `PRIMARY KEY` constraint that you'll employ. Use unique indexes when you need to build an index to enhance performance.

It should also be noted that it's very important for the performance of your systems that you use unique indexes whenever possible, as it enhances the SQL Server optimizer's chances of predicting how many rows will be returned from a query that uses the index.

In this example, if you tried to run the following query:

```
INSERT  INTO testIndex ('firstName', 'middleName', 'lastname')
VALUES ('Louis','Banks','Davidson')
INSERT INTO testIndex ('firstName', 'middleName', 'lastname')
VALUES ('James','Banks','Davidson')
```

you get the following error:

```
Server: Msg 2601, Level 14, State 3, Line 1
Cannot insert duplicate key row in object testIndex with unique index
XtestIndex1. The statement has been terminated.
```

While the error message could be much improved upon, it clearly prevents you from inserting the duplicate row with the same last name.

IGNORE_DUP_KEY

A frequently useful setting of unique indexes is `IGNORE_DUP_KEY`. By using this, you can tell SQL Server to ignore any rows with duplicate keys, if you so desire.

If you change your index in the previous example:

```
DROP INDEX testIndex.XtestIndex4
CREATE UNIQUE INDEX XtestIndex4
   ON testIndex(firstName,middleName,lastName)
   WITH ignore_dup_key
```

Then, when you execute the same code as before:

```
INSERT   INTO testIndex ('firstName', 'middleName', 'lastname')
VALUES   ('Louis','Banks','Davidson')
INSERT   INTO testIndex ('firstName', 'middleName', 'lastname')
VALUES   ('Louis','Banks','Davidson')
```

you get the following result:

```
Server: Msg 3604, Level 16, State 1, Line 3
Duplicate key was ignored.
```

This setting is probably not what is desired most of the time, but it can come in handy when building a replica of some data. Then data can be inserted without much worry as to whether it has been inserted before, as the index will toss out duplicate values. Note that this doesn't work on updates, only on inserts.

Summary

Indexes are used to speed access to the data in a table, which is why they are also used when you build constraints that enforce uniqueness. This performance comes at a price, and that price is maintenance. There is a balance in every system to which the cost of the building of one or more indices repays itself. You see how to use indexes in Chapter 10 to set up `PRIMARY KEY` and `UNIQUE` constraints, and finally in Chapter 15 when I discuss building nonvolatile read-only databases.

For more information on how indexes work, see the *CREATE INDEX* topic in SQL Server Books Online.