SQL Server Index Basics

25 November 2008
by Robert Sheldon

Given the fundamental importance of indexes in databases, it always comes as a surprise how often the proper design of indexes is neglected. It often turns out that the programmer understands detail, but not the broad picture of what indexes do. Bob Sheldon comes to the rescue with a simple guide that serves either to remind or educate us all!

One of the most important routes to high performance in a SQL Server database is the index. Indexes speed up the querying process by providing swift access to rows in the data tables, similarly to the way a book's index helps you find information quickly within that book. In this article, I provide an overview of SQL Server indexes and explain how they're defined within a database and how they can make the querying process faster. Most of this information applies to indexes in both SQL Server 2005 and 2008; the basic structure has changed little from one version to the next. In fact, much of the information also applies to SQL Server 2000. This does not mean there haven't been changes. New functionality has been added with each successive version; however, the underlying structures have remained relatively the same. So for the sake of brevity, I stick with 2005 and 2008 and point out where there are differences in those two versions.

Index Structures

Indexes are created on columns in tables or views. The index provides a fast way to look up data based on the values within those columns. For example, if you create an index on the primary key and then search for a row of data based on one of the primary key values, SQL Server first finds that value in the index, and then uses the index to quickly locate the entire row of data. Without the index, a table scan would have to be performed in order to locate the row, which can have a significant effect on performance.

You can create indexes on most columns in a table or a view. The exceptions are primarily those columns configured with large object (LOB) data types, such as **image**, **text,** and **varchar(max)**. You can also create indexes on XML columns, but those indexes are slightly different from the basic index and are beyond the scope of this article. Instead, I'll focus on those indexes that are implemented most commonly in a SQL Server database.

An index is made up of a set of pages (index nodes) that are organized in a B-tree structure. This structure is hierarchical in nature, with the root node at the top of the hierarchy and the leaf nodes at the bottom, as shown in Figure 1.

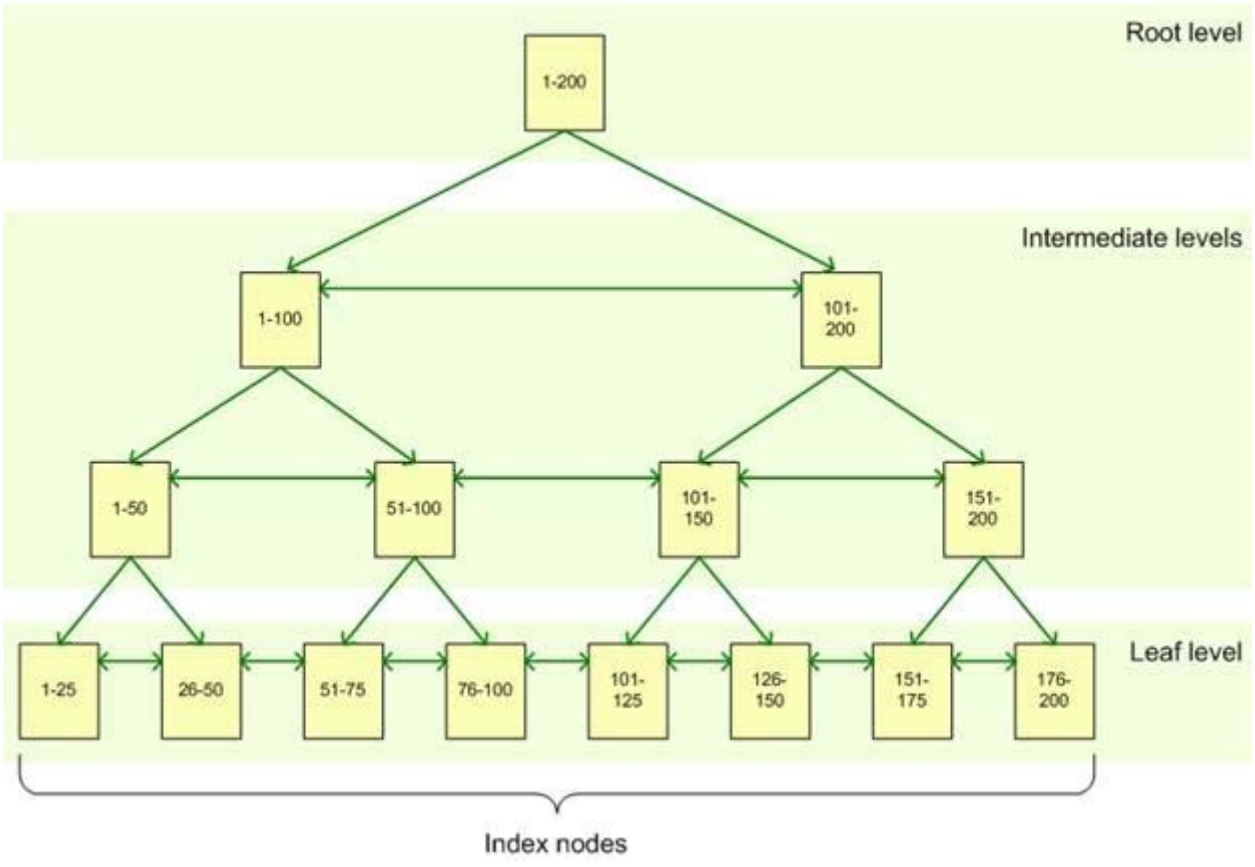

Figure 1: B-tree structure of a SQL Server index

When a query is issued against an indexed column, the query engine starts at the root node and navigates down through the intermediate nodes, with each layer of the intermediate level more granular than the one above. The query engine continues down through the index nodes until it reaches the leaf node. For example, if you're searching for the value 123 in an indexed column, the query engine would first look in the root level to determine which page to reference in the top intermediate level. In this example, the first page points the values 1-100, and the second page, the values 101-200, so the query engine would go to the second page on that level. The query engine would then determine that it must go to the third page at the next intermediate level. From there, the query engine would navigate to the leaf node for value 123. The leaf node will contain either the entire row of data or a pointer to that row, depending on whether the index is clustered or nonclustered.

Clustered Indexes

A clustered index stores the actual data rows at the leaf level of the index. Returning to the example above, that would mean that the entire row of data associated with the primary key value of 123 would be stored in that leaf node. An important characteristic of the clustered index is that the indexed values are sorted in either ascending or descending order. As a result, there can be only one clustered index on a table or view. In addition, data in a table is sorted only if a clustered index has been defined on a table.

> **Note:** A table that has a clustered index is referred to as a *clustered table*. A table that has no clustered index is referred to as a *heap*.

Nonclustered Indexes

Unlike a clustered indexed, the leaf nodes of a nonclustered index contain only the values from the indexed columns and row locators that point to the actual data rows, rather than contain the data rows themselves. This means that the query engine must take an additional step in order to locate the actual data.

A row locator's structure depends on whether it points to a clustered table or to a heap. If referencing a clustered table, the row locator points to the clustered index, using the value from the clustered index to navigate to the correct data row. If referencing a heap, the row locator points to the actual data row.

Nonclustered indexes cannot be sorted like clustered indexes; however, you can create more than one nonclustered index per table or view. SQL Server 2005 supports up to 249 nonclustered indexes, and SQL Server 2008 support up to 999. This certainly doesn't mean you should create that many indexes. Indexes can both help and hinder performance, as I explain later in the article.

In addition to being able to create multiple nonclustered indexes on a table or view, you can also add *included columns* to your index. This means that you can store at the leaf level not only the values from the indexed column, but also the values from non-indexed columns. This strategy allows you to get around some of the limitations on indexes. For example, you can include non-indexed columns in order to exceed the size limit of indexed columns (900 bytes in most cases).

Index Types

In addition to an index being clustered or nonclustered, it can be configured in other ways:

- **Composite index:** An index that contains more than one column. In both SQL Server 2005 and 2008, you can include up to 16 columns in an index, as long as the index doesn't exceed the 900-byte limit. Both clustered and nonclustered indexes can be composite indexes.
- **Unique Index:** An index that ensures the uniqueness of each value in the indexed column. If the index is a composite, the uniqueness is enforced across the columns as a whole, not on the individual columns. For example, if you were to create an index on the FirstName and LastName columns in a table, the names together must be unique, but the individual names can be duplicated.

A unique index is automatically created when you define a primary key or unique constraint:

  - **Primary key:** When you define a primary key constraint on one or more columns, SQL Server automatically creates a unique, clustered index if a clustered index does not already exist on the table or view. However, you can override the default behavior and define a unique, nonclustered index on the primary key.
  - **Unique:** When you define a unique constraint, SQL Server automatically creates a unique, nonclustered index. You can specify that a unique clustered index be created if a clustered index does not already exist on the table.
- **Covering index:** A type of index that includes all the columns that are needed to process a particular query. For example, your query might retrieve the FirstName and LastName columns from a table, based on a value in the ContactID column. You can create a covering index that includes all three columns.

Index Design

As beneficial as indexes can be, they must be designed carefully. Because they can take up significant disk space, you don't want to implement more indexes than necessary. In addition, indexes are automatically updated when the data rows themselves are updated, which can lead to additional overhead and can affect performance. As a result, index design should take into account a number of considerations.

Database

As mentioned above, indexes can enhance performance because they can provide a quick way for the query engine to find data. However, you must also take into account whether and how much you're going to be inserting, updating, and deleting data. When you modify data, the indexes must also be modified to reflect the changed data, which can significantly affect performance. You should consider the following guidelines when planning your indexing strategy:

- For tables that are heavily updated, use as few columns as possible in the index, and don't over-index the tables.
- If a table contains a lot of data but data modifications are low, use as many indexes as necessary to improve query performance. However, use indexes judiciously on small tables because the query engine might take longer to navigate the index than to perform a table scan.
- For clustered indexes, try to keep the length of the indexed columns as short as possible. Ideally, try to implement your clustered indexes on unique columns that do not permit null values. This is why the primary key is often used for the table's clustered index, although query considerations should also be taken into account when determining which columns should participate in the clustered index.
- The uniqueness of values in a column affects index performance. In general, the more duplicate values you have in a column, the more poorly the index performs. On the other hand, the more unique each value, the better the performance. When possible, implement unique indexes.
- For composite indexes, take into consideration the order of the columns in the index definition. Columns that will be used in comparison expressions in the WHERE clause (such as WHERE FirstName = 'Charlie') should be listed first. Subsequent columns should be listed based on the uniqueness of their values, with the most unique listed first.
- You can also index computed columns if they meet certain requirements. For example, the expression used to generate the values must be deterministic (which means it always returns the same result for a specified set of inputs). For more details about indexing computed columns, see the topic "Creating Indexes on Computed Columns" in SQL Server Books Online.

Queries

Another consideration when setting up indexes is how the database will be queried. As mentioned above, you must take into account the frequency of data modifications. In addition, you should consider the following guidelines:

- Try to insert or modify as many rows as possible in a single statement, rather than using multiple queries.
- Create nonclustered indexes on columns used frequently in your statement's predicates and join conditions.
- Consider indexing columns used in exact-match queries.

Index Basics

In this article, I've tried to give you a basic overview of indexing in SQL Server and provide some of the guidelines that should be considered when implementing indexes. This by no means is a complete picture of SQL Server indexing. The design and implementation of indexes are an important component of any SQL Server database design, not only in terms of what should be indexed, but where those indexes should be stored, how they should be partitioned, how data will be queried, and other important considerations. In addition, there are index types that I have not discussed, such as XML indexes as well as the filtered and spatial indexes supported in SQL Server 2008. This article, then, should be seen as a starting point, a way to familiarize yourself with the fundamental concepts of indexing. In the meantime, be sure to check out SQL Server Books Online for more information about the indexes described here as well as the other types of indexes.