Keep up to date - daily newsletter:
your@email.com
Sign up

Write For Us
Find out more

SQL Prompt
Code analysis as you type
SQLQuery1.sql
Try it now
redgate

Improve your SQL Server knowledge daily with more articles by email.
your@email.com
Sign up

Thank this author by sharing: G+ 0
Rate this | Join the discussion | Add to briefcase

An Introduction to SQL Server Indexes: Stairway to SQL Server Indexes Level 1

By David Durant, 2014/11/05 (first published: 2011/02/17)

The Series

This article is part of the Stairway Series: Stairway to SQL Server Indexes

Indexes are fundamental to database design, and tell the developer using the database a great deal about the intentions of the designer. Unfortunately indexes are too often added as an afterthought when performance issues appear. Here at last is a simple series of articles that should bring any database professional rapidly 'up to speed' with them

This first level introduces SQL Server indexes: the database objects that enable SQL Server to find and/or modify the requested data in the minimum amount of time, using the fewest system resource to achieve maximum performance. Good indexes will also allow SQL Server to achieve maximum concurrency so that queries run by one user will have little effect on queries run by others. Finally, indexes provide an efficient way of enforcing data integrity, by guaranteeing uniqueness of key values when a unique index is created. This level is an introduction; it covers concepts and usage, but leaves the physical details to a later level.

A thorough understanding of indexes is important to the database developer for one reason above all others: when a request to SQL Server arrives from the client, SQL Server has only two possible ways to access the requested rows:

- It can scan every row in the table(s) containing the data, starting at the first row and continuing to the last, examining each row to see if it meets the request criteria.
- Or, if a beneficial index is available, it can use the index to locate the requested data.

The first option is always available to SQL Server. The second option is only available if you have instructed SQL Server to create a beneficial index, but it can result in significant performance improvement, as we will illustrate later in this level.

Because indexes have overhead associated with them (they take up space and they must be kept in sync with the tables), they are not required by SQL Server. It is possible to have a database with no indexes at all. It will probably perform very poorly and it

Related Articles

ARTICLE Stairway to ColumnStore Indexes Level 7: Optimizing Nonclustered Columnstore Indexes
2016/03/11
In this level, we will focus on optimization techniques to apply while building the nonclustered col...

ARTICLE Stairway to Columnstore Indexes Level 12: Clustered or N
2
T
d
o

ARTICLE D
I
I
2
B
th
d

FORUM i
2
ir

ARTICLE Clustered Indexes: Stairway to SQL Server Indexes Level 3
2017/06/23
Now that we've seen the basics of indexing, and taken a deeper dive into Nonclustered Indexes, this ...

Fresh articles daily:
Get the SQL Server Central newsletter and get a new SQL Server article each day. Get Database Weekly for a roundup of all the biggest SQL news from around the web.
your@email.com
Sign up
No thanks

Tags
indexing
stairway series

Write For Us
Find out more

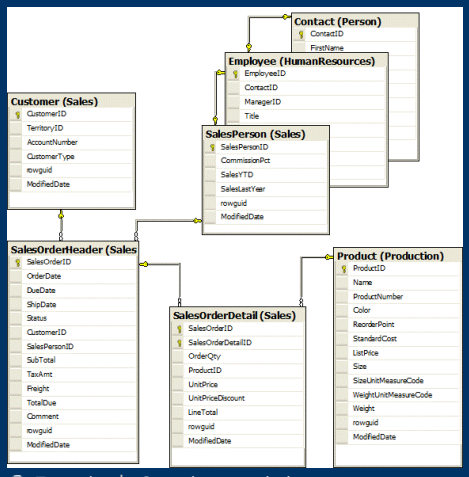
Copyright © 2002-2018 Redgate. All Rights Reserved. Privacy Policy. Terms of Use. Report Abuse.

will definitely have data integrity issues, but SQL Server will allow it.

However, it is not what we want. We all want a database that performs well, has data integrity, and, at the same time, keeps index overhead to a minimum. This level will start us toward that goal.

Throughout this StairWay we will use examples to illustrate the crucial concepts. These examples are based upon the Microsoft *AdventureWorks* sample database. We focus on Sales Order functionality. Five tables will give us a good mix of transactional and non-transactional data; *Customer*, *SalesPerson*, *Product*, *SalesOrderHeader*, and *SalesOrderDetail*. To keep things focused, we use a subset of the columns.

AdventureWorks is well normalized, so sales person information is factored into three tables; *SalesPerson*, *Employee* and *Contact*. For some examples, we will treat them as a single table. The complete set of tables we will be using, and the relationships between them, is shown in Figure 1.1.



Zoom in | Open in new window

Figure 1.1: The AdventureWorks tables that will be used in this Stairway

> **Note:**
> All TSQL code shown in this Stairway Level can be downloaded along with article (see the link at the bottom of this article)

We begin our study of indexes with a brief story; one that uses an old, but proven technology, and one that we will refer to throughout this article as we introduce the basic concepts of indexes.

> You leave your house to run a few errands. When you return, you find a message from your daughter's softball coach waiting for you. Three of the girls, Tracy, Rebecca, and Amy have lost their team caps. Could you please swing by the Athletic Products Store and buy caps for the girls. Their parents will reimburse you at the next game.
>
> You know the girls and you know their parents. But you do not know their hat sizes. Somewhere in

your town are three residences, each containing a piece of information that you need. No problem, you'll just call the parents and get the hat sizes. You reach for your phone, and you reach for an index -- the white pages of your telephone directory.

The first residence that you need to reach is that of Helen Meyer. Estimating that "Meyer" will be located near the middle of the population, you jump to the middle of the white pages, only to discover that you are at the page whose heading says "Kline-Koerber". You make a smaller jump forward and reach the "Nagle-Nyeong" page. One even smaller jump backwards puts you at the "Maldonado-Nagle" page. Realizing that you are now at the correct page, you scan down the page till you reach the "Meyer, Helen" line and obtain the telephone number. Using the phone number, you reach the Meyer residence and obtain the information you need.

You repeat the process two more times, reach two other residences, and obtain two more hat sizes.

You have just used an index, and you have used it in much the same way that SQL Server uses an index, for there are great similarities, and some differences, between the white pages and a SQL Server index.
Actually, the index you just used represents one type of SQL Server index of the two that SQL Server supports: clustered and nonclustered. The white pages best represents the concept of **a nonclustered index**. Thus, in this level we introduce nonclustered indexes. Subsequent levels will introduce clustered indexes and drill ever deeper into both types.

The white pages are analogous to a nonclustered index in that they are not an organization of the data itself, but rather, a mechanism, or map, to help you to access that data. The data itself is the actual people we need to contact. The phone company does not arrange the town's residences into a meaningful sequence, moving houses from one location to another so that all girls on the same softball team live next door to each other and the houses are not organized by residents' last name. Instead, it gives you a book containing one entry for each residence. These entries are sequenced by the white pages' search key, last name, first name, middle initial and street address. Each entry contains the search key and the piece of data that enables you to access the residence, the phone number.

Like an entry the white pages, each entry in a SQL Server nonclustered index consists of two parts:

- **The search key**, such as last name -- first name -- middle initial. In SQL Server terminology, this is the *index key*.
- **The bookmark**, which serves the same purpose as the phone number does, allowing SQL Server to navigate directly to the row in the table that corresponds to this index entry.

In addition, a SQL Server nonclustered index entry has some internal-use-only header information and may contain some optional information. Both of these will be covered in later levels.

neither is important at the time for an understanding of nonclustered indexes.

Like the white pages, a SQL Server index is maintained in search key sequence so that any specific entry can be accessed in set of small "jumps". Given the search key, SQL Server can quickly get to the index entry for that key. Unlike the white pages, a SQL Server index is dynamic. That is, SQL Server updates the index every time a row is added, removed, or has a search key column value modified.

Just as the sequence of entries in the white pages in not the same as the geographic sequence of residences within the town, the sequence of entries in the nonclustered index is not the same as the sequence of rows in the table. The first entry in the index might be that of the last row in the table, and the second entry in the index might be that of the first row in the table. If fact, unlike an index, whose entries are always in a meaningful sequence, a table's rows can be completely unsequenced.

When you create an index, SQL Server generates and maintains exactly one entry in the index for each row in the underlying table (an exception to this general rule will be encountered in a later level when we cover *filtered indexes*). You can create more than one nonclustered index on a table, but you cannot have an index that contains data from more than one table.

And the biggest difference of all: SQL Server cannot use the telephone. It must use the information in the bookmark portion of the index entry to navigate to the corresponding row of the table. This will be necessary whenever SQL Server needs any information that is in the data row but not in corresponding index entry, such as Tracy Meyer's softball cap size. So, for a better analogy, a white pages' entry contains a set of GPS coordinates instead of a phone number. You then use the GPS coordinates to navigate to the residence represented by the white page entry.

We end this level by twice querying our sample database. Make sure you are using the version of *AdventureWorks* intended for SQL Server 2005, which can be used by SQL Server 2008. The *AdventureWorks2008* database has a different table structure and the queries below will fail. We will run the same query each time; but the first execution will occur before we create an index on the table, the second execution will be after we have created an index. Each time, SQL Server will tell us how much work was done in retrieving the requested information. We'll be looking for the "Helen Meyer" row in our *Contact* table (her row is located near the middle of the table). Initially, the table will not have an index on either the *FirstName* column or the *LastName* column. To ensure you can run the example multiple times, make sure that the index we will be building in the third batch does not exist, by running the following code:

```
IF EXISTS (SELECT * FROM sys.indexes
WHERE OBJECT_ID = OBJECT_ID('Person.Contact
AND name = 'FullName')
DROP INDEX Person.Contact.FullName;
```

Listing 1.1 - Ensuring the index does not exist

Our task will require four SQL command batches.

The first command batch:

```
SET STATISTICS io ON
SET STATISTICS time ON
GO
```

Listing 1.2 - Turning on statistics

The above batch informs SQL Server that we want our queries to return performance information as part of the output.

The second command batch:

```
SELECT *
    FROM Person.Contact
    WHERE FirstName = 'Helen'
        AND LastName = 'Meyer';
GO
```

Listing 1.3 - Retrieving some data

This second batch retrieves the "Helen Meyer" row:

```
584 Helen Meyer helen2@adventure-works.com   0-
519-555-0112
```

Plus the following performance information:

```
Table 'Contact'. Scan count 1, logical reads
569.
SQL Server Execution Times:   CPU time = 3 ms.
```

This output informs us that our request performed 569 logical IOs, and required approximately 3 milliseconds of processor time to do so. Your values for processor time may be different.

The third command batch:

```
CREATE NONCLUSTERED INDEX FullName
            ON Person.Contact
    ( LastName, FirstName );
GO
```

Listing 1.4 - Creating a non-clustered index

This batch creates a nonclustered composite index on the Contact table's first and last name columns. A composite index is an index with more than one column determining the index row sequence.

The fourth command batch:

```
SELECT *
    FROM Person.Contact
    WHERE FirstName = 'Helen'
        AND LastName = 'Meyer';
GO
```

Listing 1.3 (again)

This final batch is a re-execution of our original `SELECT` statement. We get the same row returned as before, but this time the performance statistics are different:

```
Table 'Contact'. Scan count 1, logical reads 4.
SQL Server Execution Times:   CPU time = 0 ms.
```

This output informs us that our request needed only 4 logical IOs, and required an immeasurably small amount of processor time to retrieve the "Helen Meyer" row.

Creation of well-chosen indexes can greatly improve database performance. In the next level we'll begin to look at the physical structure of indexes. We'll examine why this nonclustered index was so beneficial to this query, any why that might not always be the case. Future levels will cover other types of indexes, additional benefits of indexes, the costs associated with indexes, monitoring and maintaining your indexes, and best practices, all with the goal of providing you with the knowledge necessary to create the best possible indexing scheme for your own tables in your own databases.

- Level 1 - IntroToIndexes_Durant_Code.sql
- Level 1 - MillionRowContactTable.sql

Level 1 - IntroToIndexes_Durant_Code.sql | Level 1 - MillionRowContactTable.sql

This article is part of the Stairway to SQL Server Indexes Stairway

Sign up to our RSS feed and get notified as soon as we publish a new level in the Stairway!

Improve your SQL Server knowledge daily with more articles by email.

your@email.com

Sign up

Thank this author by sharing: 0

Rate this | Join the discussion | Add to briefcase

Total article views: 57933 | Views in the last 30 days: 59