SQL ServerCentral.com
redgate | Hub
Insights Learning Forums Ideas
Discover the new Redgate Hub
a community service from redgate
Try SQL Clone

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

Want more great articles like this? Sign up for fresh SQL Server knowledge delivered daily.
your@email.com
Sign up

Thank this author by sharing: G+ 1
Rate this | Join the discussion | Add to briefcase

Deeper into Nonclustered Indexes: Stairway to SQL Server Indexes Level 2

By David Durant, 2017/10/18 (first published: 2014/11/26)

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

Level 1 in the SQL Server Indexes Stairway introduced SQL Server indexes, in general, and nonclustered indexes specifically. As our first case study, we demonstrated the potential benefit of an index when retrieving a single row from a table. In this level, we continue our investigation of nonclustered indexes; examining their contribution to good query performance in cases that go beyond retrieving a single row from a table.

As will be the case in most of these our levels, we introduce a small amount of theory, examine some index internals to help explain the theory, and then execute some queries. These queries are executed with and without indexes, and with performance reporting statistics turned on, so that we can view the impact of the indexes.

We will be using the subset of tables from the AdventureWorks database that we used in Level 1, concentrating on the `Contact` table throughout this level. We will use just one index, the `FullName` index that we used in Level 1, to illustrate our points. To ensure that we control the indexing on the *Contact* table, we'll make two copies of the table in the *dbo* schema, and only build the *FullName* index on one of them. This will give us our controlled environment: two copies of the table, one with a single nonclustered index and one without any indexes.

Note:
All TSQL code shown in this Stairway level can be downloaded at the bottom of the article.

The code in Listing 1 makes the copies of the Person.Contact table, and we can rerun this batch anytime we wish to start with a 'clean slate'.

Related Articles

FORUM index entry
2008/10/24
index entry

ARTICLE Covering Index using Included Columns
2010/02/02
This article from Josef Richberg details the benefits of included columns for use in creating a cove...

ARTICLE C G e
2
T g e

BLOG C
2
M a e

ARTICLE I
2
A ir

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.

```
IF EXISTS (

    SELECT *

        FROM sys.tables 

        WHERE OBJECT_ID = OBJECT_ID('dbo.Co
DROP TABLE dbo.Contacts_index;
GO
IF EXISTS (

    SELECT *

        FROM sys.tables 

        WHERE OBJECT_ID = OBJECT_ID('dbo.Co

    DROP TABLE dbo.Contacts_noindex;
GO
SELECT * INTO dbo.Contacts_index

    FROM Person.Contact;
SELECT * INTO dbo.Contacts_noindex

    FROM Person.Contact;
```

Listing 2.1: Make copies of the Person.Contact table

A snippet of the Contacts table is shown here:

```
ContactID   FirstName  MiddleName LastName
EmailAddress
                                    .
                                    .
1288        Laura      F          Norman
laura1@adventure-works.com
651         Michael               Patten
   michael20@adventure-works.com
1652        Isabella   R          James
isabella6@adventure-works.com
1015        David      R          Campbell
david8@adventure-works.com
1379        Balagane              Swaminath
balaganesan0@adventure-works.c
742         Steve                 Schmidt
steve3@adventure-works.com
1743        Shannon    C          Guo
shannon16@adventure-works.com
1106        John       Y          Chen
john2@adventure-works.com
1470        Blaine                Dockter
blaine1@adventure-works.com
833         Clarence   R.         Tatman
clarence0@adventure-works.com
1834        Heather    M          Wu
heather6@adventure-works.com
1197        Denise     H          Smith
denise0@adventure-works.com
560         Jennifer   J.         Maxham
```

```
jennifer1@adventure-works.com
1561        Ido                    Ben-Sacha
ido1@adventure-works.com
924         Becky      R.          Waters
becky0@adventure-works.com
```

The following statement creates our FullName nonclustered index on the Contacts_index table.

```
CREATE INDEX FullName

            ON Contacts_index

    ( LastName, FirstName );
```

Listing 2.2 - Creating a nonclustered index

Remember that a nonclustered index stores the index keys in order, along with a bookmark that is used to access the actual data in the table itself. You can think of the bookmark as a kind of pointer. Future levels will describe the bookmark, its form and its use, in more detail.

A snippet of the *FullName* index is shown here, consisting of the *LastName* and *FirstName* as Key columns, plus the bookmark:

```
:---      Search Key Columns   :
     Bookmark

                                        .

Russell          Zachary                  =>
Ruth             Andy                     =>
Ruth             Andy                     =>
Ryan             David                    =>
Ryan             Justin                   =>
Sabella          Deanna                   =>
Sackstede        Lane                     =>
Sackstede        Lane                     =>
Saddow           Peter                    =>
Sai              Cindy                    =>
Sai              Kaitlin                  =>
Sai              Manuel                   =>
Salah            Tamer                    =>
Salanki          Ajay                     =>
Salavaria        Sharon                   =>
```

Each entry contains the index key columns and the bookmark value. In addition, a SQL Server nonclustered index entry has some internal-use-only header information and may contain some optional data values. Both of these will be covered in later levels; neither is important at this time for a basic understanding of nonclustered indexes.

For now, all we need to know is that the key value enables SQL Server to find the appropriate index entry(s), and that the entry's bookmark value enables SQL Server to access the corresponding data row in the table.

An index's entries are sequenced by index key value(s), so SQL Server can rapidly traverse the entries sequentially in either direction. This scanning of sequenced entries can commence from the start of the index, the end of the index, or from any entry within the index.

Thus, if a request asks for all the contacts whose last name begins with the letter "S" (WHERE LastName LIKE 'S%'), SQL Server can quickly navigate to the first "S" entry ("Sabella, Deanna"), and then traverse the index, using the bookmarks to access the rows, until it reaches the first "T" entry; at which point it knows that it has retrieved all the "S" entries.

The above request would execute even faster if all the selected columns were in the index. Thus, if we issued:

```
SELECT FirstName, LastName

    FROM Contact

    WHERE LastName LIKE 'S%';
```

SQL Server can quickly navigate to the first "S" entry and then traverse through the index entries, ignoring the bookmarks and retrieving the data values directly from the index entries, until it reaches the first "T" entry. In relational database terminology, the index has "covered" the query.

Any SQL operator that benefits from sequenced data can benefit from an index. This includes ORDER BY, GROUP BY, DISTINCT, UNION (not UNION ALL), and JOIN…ON.

For instance, if a request asks for a count of contacts by last name, SQL Server can start counting at the first entry, and proceed down the index. Every time the value of the last name changes, SQL Server outputs the current count and starts a new count. As with the previous request, this is a covered query; SQL Server accesses just the index, ignoring the table completely.

Note the importance of the left-to-right sequence of the key columns. Our index is very helpful if a request asks for everyone whose last name is "Ashton", but it is of little or no help if the request is for everyone whose first name is "Ashton".

If you want to execute the test queries that follow, make sure you run the script to create both versions of the new *Contact* tables, *dbo.Contacts_index* and *dbo.Contacts_noindex*; and also run the script to create the *LastName, FirstName* index on *dbo.Contacts_index*.

To validate the assertions in the previous section, we turn on the same performance statistics that we used in level 1 and run same queries, with and without indexes.

```
SET STATISTICS io ON

SET STATISTICS time ON
```

Because the *Contacts* table from the *AdventureWorks* database has only 19972 rows in it, it will be difficult to get meaningful values for statistics time. Most of our queries will show a CPU time value of 0, so we are not showing the output from statistics time, only from statistics IO, which reflects the possible number of pages that will have to be read. These values will allow us to

compare queries in a relative sense, to determine which queries with which indexes perform better than others. If you want a bigger table for more realistic timing tests, a script to build a million row version of the Contact table is available with this article. All of the discussion that follows will assume you're using the standard 19972-row table.

Our first query is a query that will be covered by the index, one that retrieves a limited set of columns for all contacts whose last name begins with "S". Query execution information is given in Table 2.1.

SQL	SELECT FirstName, LastName FROM dbo.Contacts -- execute with both Contacts_noindex and -- Contacts_index WHERE LastName LIKE 'S%'
Without Index	(2130 row(s) affected) Table 'Contacts_noindex'. Scan count 1, logical reads 568
With Index	(2130 row(s) affected) Table 'Contacts_index'. Scan count 1, logical reads 14
Index Impact	IO reduced from 568 reads to 14 reads.
Comments	An index that covers the query is a good thing to have. Without an index, the entire table is scanned to find the rows. The "2130 rows" statistic indicates that "S" is a popular initial letter for last names, occurring in ten percent of all contacts.

Table 2.1: Execution results when running a covered query

Next, we modify our query to request the same rows as before, but include columns that are not in the index. Query execution information is given in Table 2.2.

SQL	SELECT * FROM dbo.Contacts -- execute with both Contacts_noindex and -- Contacts_index WHERE LastName LIKE 'S%'
Without Index	Same as previous query. (Because it is a table scan).
With Index	(2130 row(s) affected) Table 'Contact_index'. Scan count 1, logical reads 568.
Index Impact	No impact at all.
Comments	The index was never used during the execution of the query! SQL Server decided that jumping from an index entry to the corresponding row in the table 2130 times (once for each row) was more work than scanning the entire table of one million rows to find the 2130 rows that it needed.

Table 2.2: Execution results when running a non-covered query

This time, we make our query more selective; that is, we narrow down the number of rows being requested. This increases the probability that the index will be beneficial to that query. Query execution information is given in Table 2.3.

SQL	SELECT * FROM dbo.Contacts -- execute with both Contacts_noindex and -- Contacts_index WHERE LastName LIKE 'Ste%'
Without Index	Same as previous query. (Because it is a table scan)
With index	(107 row(s) affected) Table 'Contact_index'. Scan count 1, logical reads 111.
Index Impact	IO reduced from 568 reads to 111 reads..
Comments	SQL Server accessed the 107 "Ste%" entries, all of which are located consecutively within the index. Each entry's bookmark was then used to retrieve to corresponding row. The rows are not located consecutively within the table. The index benefitted this query; but not as much as it benefitted the first query, the "covered" query, especially in terms of number of IOs required to retrieve each row. You might expect that reading 107 index entries plus 107 rows would require 107 + 107 reads. The reason why only 111 reads were required will be covered at a higher level. For now, we will say that very few of the reads were used to access the index entries; most were used to access the rows. Since the previous query, which requested 2130 rows, did not benefit from the index, and this query, which requested 107 rows, did benefit from the index - you might also wonder "where does the tipping point lie?" The calculations behind SQL Server's decision also will be covered in a future level.

Table 2.3: Execution results when running a more selective non-covered query

Our last sample query will be an aggregate query; that is a query that involves counting, totaling, averaging, et cetera. In this case, it is a query that tells us the extent of name duplication within the Contact table.

The results, in part, look like this:

```
Steel          Merrill          1
Steele         Joan             1
Steele         Laura            2
Steelman       Shanay           1
```

```
Steen          Heidi          2
Stefani        Stefano        1
Steiner        Alan           1
```

Query execution information can be seen in Table 2.4.

SQL	SELECT LastName, FirstName, COUNT(*) as 'Contacts' FROM dbo.Contacts -- execute with both Contacts_noindex and -- Contacts_index WHERE LastName LIKE 'Ste%' GROUP BY LastName, FirstName
Without Index	Same as previous query. (Because it is a table scan)
With Index	(104 row(s) affected) Table 'Contacts_index'. Scan count 1, logical reads 4
Index Impact	IO reduced from 568 reads to 4 reads.
Comments	All the information needed by the query is in the index, and it is in the index in the ideal sequence for calculating the counts. All the "last name begins with 'Ste'" entries are consecutive within the index, and within that group, all the entries for a single FirstName / LastName value are grouped together. No accessing of the table was required, nor was any sorting of intermediate results needed. Again, an index that covers the query is a good thing to have.

Table 2.4: Execution results when running a covered aggregate query

If we change the query to include columns that are not in the index, we get the performance results that we see in Table 2.5.

SQL	SELECT LastName, FirstName, MiddleName, COUNT(*) as 'Contacts' FROM dbo.Contacts -- execute with both Contacts_noindex and -- Contacts_index WHERE LastName LIKE 'Ste%' GROUP BY LastName, FirstName, MiddleName
Without Index	Same as previous query. (Because it is a table scan)
With Index	(105 row(s) affected) Table 'ContactLarge'. Scan count 1, logical reads 111.
Index Impact	IO reduced from 568 reads to 111 reads, same as the previous non-covered query
Comments	Intermediate work done while processing the query does not always appear in the statistics. Techniques that use memory or tempdb to sort and merge data are examples of this. In reality, the benefit of an index may be greater than that shown by the statistics.

Table 2.5: Execution results when running a noncovered aggregation query