SQL ServerCentral.com
Proactively monitor your growing SQL Server environment.
Try free SQL Monitor
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

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

Clustered Indexes: Stairway to SQL Server Indexes Level 3

By David Durant, 2013/01/25 (first published: 2011/06/22)

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

The preceding levels in this Stairway provided an overview of indexes in general and of nonclustered indexes specifically. It concluded with the following key concept regarding SQL Server indexes. When a request arrives at your database, be it a SELECT statement or an INSERT, UPDATE or DELETE statement, SQL Server has only three possible ways to access the data of a table referenced in the statement:

- Access just the nonclustered index and avoid accessing the table. This in only possible if the index contains all the data for this table that is being requested by the query
- Use the search key(s) to access the index, and then use the selected bookmark(s) to access individual rows of the table.
- Ignore the index and search the table for the requested rows.

This level begins by focusing on the third choice in the above list; searching the table. This in turn, will lead us to a discussion of clustered indexes; a subject that was mentioned, but not covered, in Level 2.

The primary *AdventureWorks* database table that we will be using during this level is the *SalesOrderDetail* table. At 121,317 rows, it is large enough to illustrate some benefits of a having clustered index on a table. And, having two foreign keys, it is complex enough to illustrate some the design decisions you must make about your clustered indexes.

Although we already discussed the sample database in Level 1, it bears repeating at this time. Throughout this Stairway we will use examples to illustrate concepts. These examples are based upon

Related Articles

FORUM Insert on a clustered index - is column ordering (ASC or DESC) important ?
2011/03/10
insert clustered index column order asc desc

FORUM clustered index
2012/04/10
clustered index

BLOG
FORUM
BLOG

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

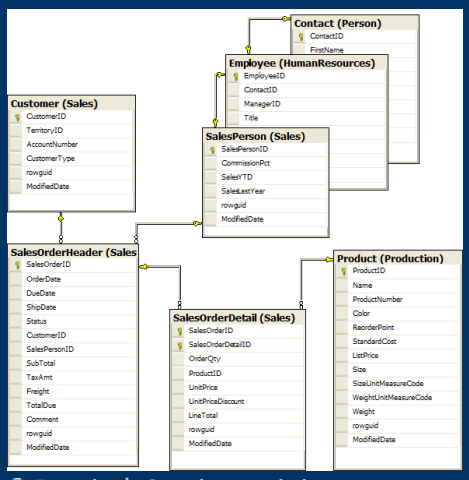
Write For Us
Find out more

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

the Microsoft *AdventureWorks* sample database. We focus on sales orders. 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. Because *AdventureWorks* is well normalized, sales person information is factored into three tables: *SalesPerson*, *Employee* and *Contact*.

Throughout this Stairway we use the following two terms that refer to a single line on an order interchangeably: "line item" and "order detail". The former is the more common business term; the latter appears within the name of an *AdventureWorks* table.

The complete set of tables, and the relationships between them, is shown in Figure 1.



Zoom in | Open in new window

Figure 1: The tables used in the examples in this Stairway

> **Note:**
> All TSQL code shown in this Stairway level can be downloaded along with article.

We start by asking the following question: how much work is required to find a row(s) in a table if a nonclustered index is not used? Does searching the table for the requested rows mean scanning every row in an unordered table? Or could SQL Server permanently sequence the rows of the table so that it could quickly access them by search key, just as it quickly accesses the entries of a nonclustered index by search key? The answer depends upon whether you instructed SQL Server to create a clustered index on the table or not.

Unlike nonclustered indexes, which are a separate object and occupy their own space, the clustered index and the table are one and the same. By creating a clustered index, you instruct SQL Server to sort the rows of the table into index key sequence and to maintain that sequence during future data modifications. Upcoming levels will look at the internal data structures that are generated to accomplish this. But for now, think of a clustered index as a sorted table. Given the index key value for a row, SQL Server can quickly access that row, and can proceed sequentially through the table from that row.

For demonstration purposes we create two copies of our example table, *SalesOrderDetail*: one with no indexes and one with a clustered index. Regarding the index's key columns, we make the same choice that designers of the *AdventureWorks*database

made: SalesOrderID / SalesOrderDetailID. The code in Listing 1 makes the copies of the SalesOrderDetail table. We can rerun this code anytime we wish to start with a 'clean slate'.

```
IF EXISTS (SELECT * FROM sys.tables 
WHERE OBJECT_ID = OBJECT_ID('dbo.SalesOrder
DROP TABLE dbo.SalesOrderDetail_index;
GO
IF EXISTS (SELECT * FROM sys.tables 
WHERE OBJECT_ID = OBJECT_ID('dbo.SalesOrder
DROP TABLE dbo.SalesOrderDetail_noindex;
GO


SELECT * INTO dbo.SalesOrderDetail_index FR
SELECT * INTO dbo.SalesOrderDetail_noindex
GO


CREATE CLUSTERED INDEX IX_SalesOrderDetail
ON dbo.SalesOrderDetail_index (SalesOrderID
GO
```

Listing 1: Create copies of the SalesOrderDetail table

So, assume the SalesOrderDetail table looks like this before a clustered index is created:

```
SalesOrderID SalesOrderDetailID ProductID
OrderQty UnitPrice
69389        102201             864
3        38.10
56658        59519              711
1        34.99
59044        70000              956
        2        1430.442
48299        22652              853
4        44.994
50218        31427              854
8        44.994
53713        50716              711
1        34.99
50299        32777              739
1        744.2727
45321        6303               775
6        2024.994
72644        115325             873
1        2.29
48306        22705              824
4        141.615
69134        101554             876
1        120.00
48361        23556              760
3        469.794
53605        50098              888
1        602.346
48317        22901              722
1        183.9382
66430        93291              872
```

```
1        8.99
65281        90265              889
2        602.346
52248        43812              871
1        9.99
47978        20189              794
2        1308.9375
```

After creating the clustered index shown above, the resulting table / clustered index would look like this:

```
SalesOrderID SalesOrderDetailID ProductID
OrderQty UnitPrice
43668        106                722
3          178.58
43668        107                708
1           20.19
43668        108                733         3
        356.90
43668        109                763
3          419.46
43669        110                747
1          714.70
43670        111                710
1            5.70
43670        112                709
2            5.70
43670        113                773
2        2,039.99
43670        114                776
1        2,024.99
43671        115                753
1        2,146.96
43671        116                714
2           28.84
43671        117                756
1          874.79
43671        118                768
2          419.46
43671        119                732
2          356.90
43671        120                763
2          419.46
43671        121                755
2          874.79
43671        122                764
2          419.46
43671        123                716
1           28.84
43671        124                711
1           20.19
43671        125                708
1           20.19
43672        126                709
6            5.70
43672        127                776
2        2,024.99
43672        128                774
1        2,039.99
43673        129                754
```

```
1           874.79
43673         130                 715
3            28.84
43673         131                 729
1           183.94
```

As you look at the sample data shown above, you may notice that each *SalesOrderDetailID* value is unique. Do not be confused; *SalesOrderDetailID* is not the primary key of the table. The combination of *SalesOrderID* / *SalesOrderDetailID* is the primary key of the table, as well as the index key for the clustered index.

The clustered index key can be comprised of any columns you chose; it does not have to be based on the primary key. In our example here, what is most important is that the left most column of the key is a foreign key, the *SalesOrderID* value. Thus, all line items for a sales order appear consecutively within the *SalesOrderDetail* table.

Keep in mind these additional points about SQL Server clustered indexes:

- Because the entries of the clustered index are the rows of the table, there is no bookmark value in a clustered index entry. When SQL Server is already at a row, it does not need a piece of information that tells it where to find that row.
- A clustered index always covers the query. Since the index and the table are one and the same, every column of the table is in the index.
- Having a clustered index on a table does not impact your options for creating nonclustered indexes on that table.

There can be, at most, one clustered index per table. The rows of a table can be in only one sequence. You need to decide what sequence, if any, would be best for each table; and, if possible, create the clustered index before the table becomes filled with data. When making this decision, keep in mind that sequencing not only means ordering, it also means grouping; as in grouping line items by sales order.

This is why the designers of the *AdventureWorks*database chose *SalesOrderDetailID* within *SalesOrderID* as the sequence for the *SalesOrderDetail* table; it is the natural sequence for line items. For instance, if a user requests a line item of an order, they will usually request all the line items for that order. One look at a typical sales order form tells us that a printed copy of the order always includes all the line items. It is the nature of the sales order business to cluster line items by sales order. There may be an occasional request from the warehouse wanting to view line items by product rather than by sales order; but the majority of the requests, such as those from sales people, or customers, or the program that prints invoices, or a query that calculates the total value of each order, will need all the line items for any given sales order.

User requirements alone, however, do not determine what would be the best clustered index. Future levels in this series will cover

the internals of indexes, because certain internal aspects of indexing will also influence your choice of clustered index columns.

If there is no clustered index on a table, the table is called a heap. Every table is either a heap or a clustered index. So, although we often state that every index falls into one of two types, clustered or nonclustered, it is equally important to note that every table falls into one of two types: it is a clustered index or it is a heap. Developers often say that a table "has" or "does not have" a clustered index, but it is more meaningful to say that the table "is" or "is not" a clustered index.

There is only one way for SQL Server to search a heap when looking for rows (excluding the use of nonclustered indexes), and this is to start at the very first row in the table and proceed through the table until all the rows have been read. Without a sequence, there is no search key and no way to quickly navigate to specific rows.

To evaluate the performance of a clustered index versus a heap, listing 1 makes two copies of the *SalesOrderDetail*table. One copy is the heap version. On the other, we create the same clustered index that is on the original table (*SalesOrderID*, *SalesOrderDetailID*). Neither table has any nonclustered indexes.

We will run the same three queries against each version of the table; one that retrieves a single row, one that retrieves all rows for a single order, and one that retrieves all rows for a single product. We present the SQL and the results of each execution in the tables shown below.

Our first query retrieves a single row and the execution details are shown in Table 1.

SQL	SELECT * FROM SalesOrderDetail WHERE SalesOrderID = 43671 AND SalesOrderDetailID = 120
Heap	(1 row(s) affected) Table 'SalesOrderDetail_heindex'. Scan count 1, logical reads 1495
Clustered Index	(1 row(s) affected) Table 'SalesOrderDetail_heindex'. Scan count 1, logical reads 3
Impact of having the Clustered Index	IO reduced from 1495 reads to 3 reads.
Comments	No surprise. Table scanning 121,317 rows to find just one is not very efficient.

Table 1: Retrieving a single row

Our second query retrieves all rows for a single Sales Order, and you can see the execution details in Table 2.

SQL	SELECT * FROM SalesOrderDetail WHERE SalesOrderID = 43671
Heap	(11 row(s) affected)

	Table 'SalesOrderDetail_noindex'. Scan count 1, logical reads 1495.
Clustered Index	(11 row(s) affected) Table 'SalesOrderDetail_noindex'. Scan count 1, logical reads 3.
Impact of having the Clustered Index	IO reduced from 1495 reads to 3 reads.
Comments	Same statistics as the previous query. The heap still required a table scan, while the clustered index grouped the 11 detail rows of the requested order sufficiently close together so that the IO required to retrieve 11 rows was the same as the IO required to retrieve one row. An upcoming Level will explain in detail why no additional reads were required to retrieve the additional 10 rows.

Table 2: Retrieving all rows for a single SalesOrder

And our third query retrieves all rows for a single Product, with the execution results as shown in Table 3.

SQL	SELECT * FROM SalesOrderDetail WHERE ProductID = 755
Heap	(228 row(s) affected) Table 'SalesOrderDetail_noindex'. Scan count 1, logical reads 1495.
Clustered Index	(228 row(s) affected) Table 'SalesOrderDetail_index'. Scan count 1, logical reads 1513.
Impact of having the Clustered Index	IO slightly greater for the clustered index version; 1513 reads versus 1495 reads.
Comments	Without a nonclustered index on the ProductID column to help find the rows for a single Product, both versions had to be scanned. Because of the overhead of having a clustered index, the clustered index version is the slightly larger table; therefore scanning it required a few more reads than scanning the heap.

Table 3: Retrieving all rows for a single Product

Our first two queries greatly benefitted from the presence of the clustered index; the third was approximately equal. Are there times when a clustered index is a detriment? The answer is yes, and it is mostly related to inserting, updating and deleting rows. Like so many other aspects of indexing encountered in these early Levels, it too is a subject that will be covered in more detail in a higher Level.

In general, the retrieval benefits outweigh the maintenance detriments, making a clustered index preferable to a heap. If you are creating tables in an Azure database, you have no choice; every table must be a clustered index.

A clustered index is a sorted table whose sequence is specified by you when the index is created, and maintained by SQL