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

Daily SQL Articles by email:
your@email.com
Sign up

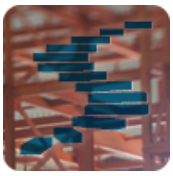
Write For Us
Find out more

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

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

Stairway to Columnstore Indexes Level 2: Columnstore Storage



By Hugo Kornelis, 2017/04/21 (first published: 2015/04/29)

The Series

This article is part of the Stairway Series: Stairway to Columnstore Indexes

SQL Server 2012 and later offer a very different type of index from the traditional b-tree, the in-memory columnstore index. These indexes use a column-based storage model, as well as a new 'batch mode' of query execution and can offer huge performance increases for certain workloads. But how are they built, how do they work, and why do they manage to have such a dramatic impact on performance? In this stairway, Hugo Kornelis explains all, with his usual mix of concise description and detailed demonstration.

The introduction of columnstore indexes in SQL Server 2012 was a revolutionary step in the world of SQL Server. A new index type was introduced, based on a completely new storage concept. This was considered so crucial that the very storage paradigm is encoded in the name, to remind us every time we use them that these indexes use a **columnar** oriented **storage** structure.

To fully appreciate just how different these indexes are, and why work so well in reporting and online analytical processing (OLAP) workloads, but not for online transaction processing (OLTP), we must first look at the traditional “rowstore” indexes.

Row-oriented storage

Traditionally, all indexes indexes in SQL Server share one feature: the basic layout of the pages in the database file. For each row, one single, consecutive block of bytes is used to store the values in all columns, or all columns included in the index. Even for LOB data and other varying length data that is too large to fit on the page, a pointer to the actual data is still included in the block of bytes that represents a row.

To understand how this works, let's look at a simple sample table. (See figure 2-1). Note that this table is for demonstration purposes only, and is not intended to be an example of good table design.

Related Articles

BLOG Clustered Columnstore Indexes – part 34 (“Deleted Segments Elimination”)
2014/07/20
Continuation from the previous 33 parts, starting from http://www.nikoport.com/2013/07/05/clustered

BLOG Clustered Columnstore Indexes – part 48 (“Improving Dictionary Pressure”)

BLOG

BLOG

BLOG Columnstore Indexes – part 84 (“Practical Dictionary Cases”)
2016/06/20
Continuation from the previous 83 parts, the whole series can be found at http://www.nikoport.com/co...

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

columnstore index
stairway series

Write For Us
Find out more

Copyright © 2002-2020 Redgate. All Rights Reserved. Privacy Policy. Terms of Use. Report Abuse.

Saledate	Product	Amt	GrossPrice	SalesTax	NetPrice	...
2012-03-08	Candy bar	50	75.00	14.25	89.25	...
2012-03-10	Smart phone	1	349.50	66.41	419.91	...
2012-03-11	Apple (bag)	7	31.57	1.89	33.46	...
2012-03-12	Smart phone	1	349.50	66.41	419.91	...
2012-03-19	Chair	1	599.50	113.91	713.41	...
2012-03-20	Toy car	3	29.97	5.69	35.66	...
2012-03-20	Chair	3	1,798.50	341.72	2,140.22	...
2012-03-20	Laptop	2	2,860.00	543.40	3,403.40	...
2012-03-21	Apple (bag)	14	63.14	3.79	66.93	...
2012-03-24	Pocket knife	1	12.95	2.46	15.41	...
2012-03-27	Apple (bag)	2	9.02	0.54	9.56	...
2012-03-28	Candy bar	5	7.50	1.43	8.93	...

Zoom in | Open in new window

Figure 2-1: Sample table

When SQL Server has to store this data, it will start by writing all columns of the first row to a page. Then if there is enough space, the second row will be added to the same page, and so on until the page is full. And then a new page is allocated and the process continues. In figure 2-2, you see a graphical representation of the end product, with different colors used to visualize the different pages. (In reality, a single page has 8,060 bytes available, so more than three rows will fit on a page; this is a simplification to visualize the structure – plus, I don't get paid by the word, so why bother creating a sample with hundreds of rows?)

Saledate	Product	Amt	GrossPrice	SalesTax	NetPrice	...
2012-03-08	Candy bar	50	75.00	14.25	89.25	...
2012-03-10	Smart phone	1	349.50	66.41	419.91	...
2012-03-11	Apple (bag)	7	31.57	1.89	33.46	...
2012-03-12	Smart phone	1	349.50	66.41	419.91	...
2012-03-19	Chair	1	599.50	113.91	713.41	...
2012-03-20	Toy car	3	29.97	5.69	35.66	...
2012-03-20	Chair	3	1,798.50	341.72	2,140.22	...
2012-03-20	Laptop	2	2,860.00	543.40	3,403.40	...
2012-03-21	Apple (bag)	14	63.14	3.79	66.93	...
2012-03-24	Pocket knife	1	12.95	2.46	15.41	...
2012-03-27	Apple (bag)	2	9.02	0.54	9.56	...
2012-03-28	Candy bar	5	7.50	1.43	8.93	...

Zoom in | Open in new window

Figure 2-2: Row-oriented storage

This storage structure is ideal for OLTP workloads. If you need to insert a row, delete a row, or update a row, you make all changes on just a single page in the data file. And queries that are selective enough to return only a few rows usually also touch only a single page, or two pages in the worst case. In the example in Figure 2-2, if you select all sales for a single day, you need just a single page for most dates, and still only two for March 20, 2012.

But in reporting and analytical processing, the typical query looks much different. In a data warehouse (DW), most tables have dozens of columns (at least), and most queries select only a few of those columns – forcing SQL Server to process the pages that contain not only the three columns you need, but also all of the other columns, even though they are not needed. A second difference is that typical DW queries are not very restrictive; they process large amounts of data, using a WHERE clause that selects millions of rows or even no WHERE clause at all, and then use aggregates to trim the result set to something a human can peruse.

So when an information worker is trying to identify a trend in amounts and net prices over time, they would probably use a query that selects the Saledate, *Amt*, and NetPrice column, with no WHERE clause. In figure 2-3, I have highlighted the required data in bold and red, and shaded the data actually read from disk or cache. All the non-bold and italic data is read as well, even though it is not used in the query. Reading this additional data causes significant overhead.

Saledate	Product	Amt	GrossPrice	SalesTax	NetPrice	...
2012-03-08	*Candy bar*	**50**	*75.00*	*14.25*	**89.25**	...
2012-03-10	*Smart phone*	**1**	*349.50*	*66.41*	**419.91**	...
2012-03-11	*Apple (bag)*	**7**	*31.57*	*1.89*	**33.46**	...
2012-03-12	*Smart phone*	**1**	*349.50*	*66.41*	**419.91**	...
2012-03-19	*Chair*	**1**	*599.50*	*113.91*	**713.41**	...
2012-03-20	*Toy car*	**3**	*29.97*	*5.69*	**35.66**	...
2012-03-20	*Chair*	**3**	*1,798.50*	*341.72*	**2,140.22**	...
2012-03-20	*Laptop*	**2**	*2,860.00*	*543.40*	**3,403.40**	...
2012-03-21	*Apple (bag)*	**14**	*63.14*	*3.79*	**66.93**	...
2012-03-24	*Pocket knife*	**1**	*12.95*	*2.46*	**15.41**	...
2012-03-27	*Apple (bag)*	**2**	*9.02*	*0.54*	**9.56**	...
2012-03-28	*Candy bar*	**5**	*7.50*	*1.43*	**8.93**	...

Zoom in | Open in new window

Figure 2-3: For DW queries, much more data is read than is needed

One possible way to alleviate this overhead is to use nonclustered covering indexes with included columns. A covering index on only the columns Saledate, *Amt*, and NetPrice would make it possible to execute this same data without the additional overhead. But the next query fired by the same information worker will probably select a different subset of columns, and it is simply not feasible to have a nonclustered index on every perceivable combination of columns.

Data warehouses grow bigger by the day. At the same time, management has an increased demand to get actionable results from that huge pile of data – fast! With the unavoidable overhead introduced by the row-oriented storage structure, this was getting harder and harder. The only way to deal with this problem was to make the overhead go away. That is where the new column-oriented storage structure comes in.

In a columnstore index, data is still stored on the same 8K pages that SQL Server has been using since SQL Server 7.0. But instead of filling these pages with complete rows, a columnstore index isolates all values for a single column and stores them in one or more pages. The data of the other columns then go into different pages. Obviously, for columns with a small data type, more values fit on a single page than for columns with longer data types. See figure 2-4 for a simplified representation of data stored in a columnstore index.

Saledate	*Product*	*Amt*	*GrossPrice*	*SalesTax*	*NetPrice*	...
2012-03-08	Candy bar	50	75.00	14.25	89.25	...
2012-03-10	Smart phone	1	349.50	66.41	419.91	...
2012-03-11	Apple (bag)	7	31.57	1.89	33.46	...
2012-03-12	Smart phone	1	349.50	66.41	419.91	...
2012-03-19	Chair	1	599.50	113.91	713.41	...
2012-03-20	Toy car	3	29.97	5.69	35.66	...
2012-03-20	Chair	3	1,798.50	341.72	2,140.22	...
2012-03-20	Laptop	2	2,860.00	543.40	3,403.40	...
2012-03-21	Apple (bag)	14	63.14	3.79	66.93	...
2012-03-24	Pocket knife	1	12.95	2.46	15.41	...
2012-03-27	Apple (bag)	2	9.02	0.54	9.56	...
2012-03-28	Candy bar	5	7.50	1.43	8.93	...

Zoom in | Open in new window

Figure 2-4: Column-oriented storage

For OLTP workloads, this storage paradigm spells disaster. To reconstruct just a single row of data, one full page has to be read for each column in the table. And then there is also some overhead required to reconstruct the original row – the only way of knowing that the sale date March 19, the product Chair, and the net price of 713.41 belong together is because each of them is the fifth element in the value list of that column.

But for DW workloads, the reverse holds. If you compare figure 2-3 and figure 2-4, you will see that the column-oriented storage paradigm allows SQL Server to read the required data for this query without reading a single byte of data it does not need.

To see this in action, we can run the code in listing 2-1 in the database we downloaded in the first level and then created a nonclustered columnstore index for. On my system, the first query requires only 92 logical reads, whereas the second query takes 184 logical reads, which is twice as many. Since the data is stored by column, reading all the data in two columns is twice as much work as reading all the data in a single column.

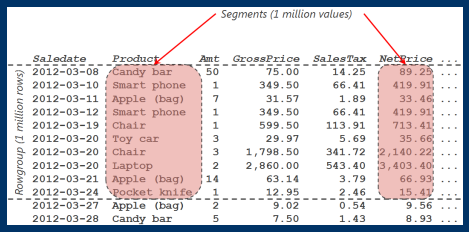
```
USE ContosoRetailDW;
GO

SET STATISTICS IO ON;

DECLARE @Dummy int;

SELECT @Dummy = SUM(SalesQuantity)
FROM   dbo.FactOnlineSales;

SELECT @Dummy = SUM(SalesQuantity - ReturnQu
FROM   dbo.FactOnlineSales;
```

Listing 2-1: Reading less columns requires less I/O

There are a few additional bonus features to the columnstore index as well, but to understand them, we have to drill a bit deeper.

Columnstore indexes are intended to be used for very large tables with at least tens of millions of rows. Those tables are often partitioned, and SQL Server's columnstore indexes automatically align with the table's partition scheme. But even a single partition will contain several million or even billions of rows. For a columnstore index, each partition (or unpartitioned table) is further divided into units called rowgroups, each containing up to about a million rows. (The actual maximum is 1,048,576 rows – 220 for the binary minded).

A rowgroup is sometimes also called a segment – but that is technically incorrect. The term *segment* is actually used for a single column of a single rowgroup. See figure 2-5 for an illustration of these concepts.



Zoom in | Open in new window

Figure 2-5: Rowgroups and segments

With proper forethought, this subdivision into rowgroups and segments can really work to your advantage. That is because SQL Server, in the metadata for each individual segment, keeps track of the minimum and maximum of all the values stored in that segment. For the sample data depicted above, the metadata of the *Saledate* data for the depicted segment would show a minimum value of 2012-03-08 and the maximum would be 2012-03-24. Now when a query is executed that needs *Product*, *GrossPrice*, and *SalesTax* data for sales made in April 2012, a simple comparison of the values in the WHERE clause with the metadata for the segment will allow SQL Server to immediately conclude that there can never be any matching rows in the first rowgroup – so all the segments of this rowgroup (not just *Saledate*, but *Product*, *GrossPrice*, and *SalesTax* as well) are

skipped. SQL Server will only process rowgroups where the *Saledate* segment metadata shows that the interval between minimum and maximum value of *Saledate* in that column overlaps with the interval specified in the WHERE clause. This process is called "segment elimination" – even though the process actually eliminates entire rowgroups, not just segments.

So far, the process is remarkably similar to the partition elimination you would get if you had partitioned the table on the *Saledate* column. But segment elimination truly takes this to the next level. Partition elimination works only on the partitioning column, but segment elimination works on almost every column. For instance, with the data of figure 2-5, a filter on *NetPrice* equal to 3,500 would also immediately rule out the depicted rowgroup, because the maximum value is 3,403.40. However, a filter on *NetPrice* equal to 3,300 would still have to process this rowgroup to determine that there are no matches, because this cannot be predicted from the available metadata (which is limited to just the minimum and maximum).

The only limitation of segment elimination is that it does not work on string columns. A filter on *Product* equal to "Xylophone" will not eliminate the rowgroup, even though we can see that the maximum value is "Toy car". This should not be a major limitation for most use cases; in a properly designed star schema the big fact tables should have very few string columns, and none that are filtered on.

The effect of segment elimination can be observed by running the code in listing 2-2. The results from the query show that there are more rows with PromotionKey 28 than there are with PromotionKey 18, yet the first query requires more logical I/O. On my system, the first query requires 102 logical reads, but the second uses only 82. That is because there are a few segments where the highest PromotionKey is less than 28, but none with maximum PromotionKey less than 18. As a result, some segments are eliminated for the second query, but not for the first. Note that the exact results on your system may vary; this is related to details of how columnstore indexes are built, as will be explained in a later level.

```
USE ContosoRetailDW;
GO

SET STATISTICS IO ON;

SELECT COUNT(*)
FROM   dbo.FactOnlineSales
WHERE  PromotionKey = 18;

SELECT COUNT(*)
FROM   dbo.FactOnlineSales
WHERE  PromotionKey = 28;
```

Listing 2-2: Segment elimination in action

In a later level, we will cover how you can inspect the metadata, and we will also give some guidance on how to ensure that you get optimal benefit from this "segment elimination" process.

Data stored in a columnstore index is always compressed. And the effectiveness of compression achieved for columnstore indexes is extremely high. The reason for this is that the column-oriented storage structure causes all data stored on a single page to be of the same type. That kind of uniform data compresses much better than the more disparate data found in the data pages of row-oriented storage structures. In a test I did with realistic test data, row compression reduced the on-disk data size to 44% of the original size, and page compression got it down to 25% – but columnstore compression reduced that to just 9.9% of the original size!

This is not just a savings on disk. When the data has to be read from disk, far fewer bytes have to be read from storage. This can result in significant performance benefits. And once it has been read from disk, it will be stored, still in compressed form, in main memory for later reuse. Since your servers has a finite amount of memory, that is used for all the data needed by the SQL Server instance, compressing the data in a frequently-used large table can also benefit other queries by increasing the cache hit ratio and page life expectancy, which in turn will reduce physical I/O and increase the overall performance of the database server.

The compression algorithms used for columnstore indexes are all designed to ensure that the extra CPU required for decompressing the data does not exceed the savings achieved by the reduced I/O. This is of course very important for data that is frequently used. But for older data that is only accessed infrequently, you may decide that a significant saving on the storage cost is more important than the best possible query performance. This is now made possible in SQL Server 2014 by the introduction of the new "Archival Columnstore" compression mode for columnstore indexes. When this is used, the most aggressive compression algorithms are used, shaving the total storage size further down to just 4.5% of the original size: a 55% reduction when compared to default columnstore mechanism. But this is not a free lunch – you pay for the reduced disk footprint by increased CPU use when decompressing the data, so much that the query performance may deteriorate.

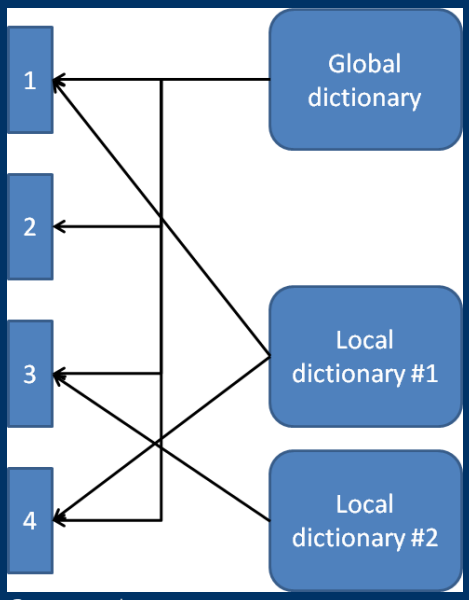
All data types that store character data (n)char and (n)varchar), as well as binary and varbinary, when used in a columnstore index, use dictionaries for their internal storage. For other columns, dictionaries may also be used – especially if they use only a small set of distinct values. Using a dictionary means that SQL Server creates a list of all values that are used, and assigns each value a unique number. For a column with colors, the dictionary would, for instance, contain a list of the value pairs (Black, 1); (Red, 2); (Green, 3); (Mauve, 4); etc. This dictionary is stored as a separate entity, related to but not part of the storage area used for the table data.

The actual table data is then stored by replacing the colors with their respective numbers. So a row for a Black T-Ford would not store the color as "black", but as "1".

The actual implementation of dictionaries in SQL Server's columnstore indexes is a bit more complex than this. For every column that uses a dictionary, a single "Global Dictionary" is used by all segments. But individual segments may also use one extra dictionary, the "Local Dictionary". These local dictionaries can be shared by multiple segments – see figure 2.6 for an illustration.

All four segments use the global dictionary, and there are two local dictionaries. One of those local dictionaries is shared between segments 1 and 4, the other is exclusive to segment 3; segment 2 uses the global dictionary only.

Note that in SQL Server 2012, different terminology was used: "Primary" and "Secondary" dictionaries rather than "Global" and "Local". I will stick to the new terminology, except when discussing issues related to SQL Server 2012 only.



Zoom in | Open in new window

Figure 2.6: Global and local dictionaries

Though the reasons for this architecture are not documented, they are not hard to guess. With the size of tables where columnstore indexes are recommended, a single dictionary for the entire column would quickly grow to unmanageable sizes. But completely separate dictionaries for every segment would include a lot of overhead -- if the data in the column "color" is stored in 2000 segments, you do not want 2000 different dictionaries all defining values for the colors black, red, green, and blue.

The more complex architecture with one global dictionary and multiple local dictionaries allows SQL Server to store values that are used in all segments just once for all the segments, and store values used in a specific segment only in its reserved secondary dictionary without cluttering the dictionary for the other segments, without exceeding the size limit of 16 MB for the global dictionary and another 16 MB for each local dictionary.

The architecture of a columnstore index allows far better compression than the traditional indexes, because data that is stored together is from a single column and hence tends to be more similar. That saves on storage size, and by extension also on I/O needed for executing a query, which translates into better performance.

Additionally, more I/O is saved because only data is read for columns that are used in the query, and segment elimination provides a third mechanism of I/O reduction.

This article is part of the Stairway to Columnstore Indexes Stairway