http://www.sqlservercentral.com/articles/ColumnStore+Index/110762/
Printed 2014/08/07 06:29AM

Introducing the Columnstore Index

By Gayathri Jayaraman, 2014/05/16

Introduction

The Columnstore index was first introduced in SQL Server 2012 and despite its restrictive nature, has been widely acclaimed for its bold attempt towards near-real time execution of some data warehousing queries. This article aims to demystify the Columnstore index, how it has evolved in SQL Server 2014, and its typical usage scenarios.

What is Columnstore?

Indices in SQL Server have traditionally been "row stores". In a row store, all field values for a row are stored together, in a single row. And, multiple such rows of the table are stored in the same page.

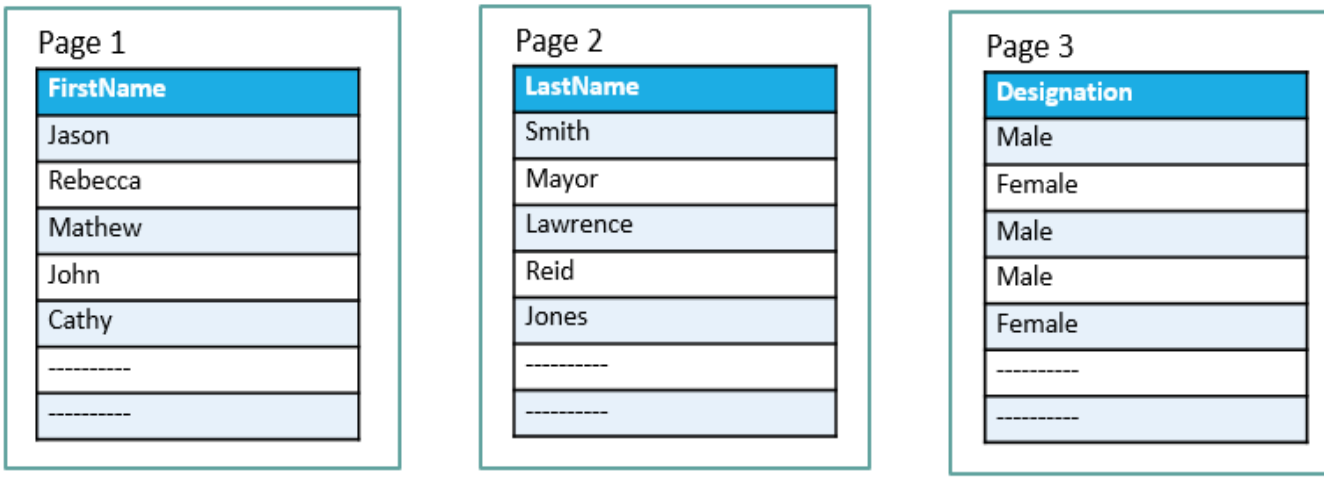
A typical row store page would look like the image below. All the column values of a table, for multiple rows, would be stored in the same page:

Page 1

FirstName	LastName	Gender	Designation
Jason	Smith	Male	Developer
Rebecca	Mayor	Female	Architect
Mathew	Lawrence	Male	Architect
John	Reid	Male	Developer
Cathy	Jones	Female	Developer
----------	----------	----------	----------
----------	----------	----------	----------

Starting SQL Server 2012, Microsoft has introduced a new type of index called the *Columnstore index*. As the name suggests, a Columnstore index stores and retrieves data in a columnar format. Data for "each" column of a Columnstore index is stored in a separate page.

A typical column store page would look like this – values for a single column stored as column vectors in the same page:



At the first thought, it makes one question the usefulness of such an index. In our day-to-day querying scenarios, doesn't a Columnstore seem like an overkill? Reading page after page of data to get all the fields for just one record doesn't sound right. But then, a Columnstore index is not intended for routine queries. In fact, it may perform poorly if we try to fit it in our OLTP scenarios that often require a record lookup.

The real benefits of the Columnstore index come to light in data warehousing scenarios that generally deal with large number of rows but, more often than not, only a fewer number of columns.

There are some restrictions to using the Columnstore index too. To begin with, only one of it can be created on a table. Secondly, not all data types can be used in a Columnstore index - Varchar(max), nVarchar(max), text, ntext, binary, varbinary and XML are some that are not supported.

As there is no concept of key column in a Columnstore index, index key-related limitations are not really applicable. Also, all columns in a Columnstore index are INCLUDED columns.

Columnstore index is of two types:

i. Non-clustered Columnstore
ii. Clustered Columnstore

The following table lists some information and key differences between the two types:

Non-clustered Columnstore	**Clustered Columnstore**
Introduced in SQL 2012, was the only type available in SQL 2012	Introduced in SQL 2014
Once created, the underlying table becomes read-only	Table is open for DML operations
Could be used only as the secondary index. So, data is necessarily duplicated	Can be used as the primary (clustered) index. Hence, must include all columns of the table (field data is not sorted while storing, though)

Can be a subset of another index; Table can have other index too	Is the only index in table. Cannot be clubbed with any other index

Note that if the base table is a clustered index, all columns MUST be present in the Columnstore index. Any column ignored in the CREATE INDEX statement gets added automatically.

Usage Example

We will now try out a simple use case that highlights the behavior of a Columnstore index. For our example scenario, we will use SalesOrderDetail from the popular AdventureWorks sample database (available for free download). This table has around 120K records – not a lot but enough to prove the point.

1. From this location - http://msftdbprodsamples.codeplex.com/releases/view/55330, download the backup files for AdventureWorks2012
2. Restore the database
3. Run the attached 'Table creation scripts' in the newly created database.

   This script contains table and index creation statements for the following tables:

   i. SalesOrderDetail – with no index (this table already exists in AdventureWorks2012 and we will use it as-is)
   ii. SalesOrderDetail_Clustered – create with a clustered index on id field
   iii. SalesOrderDetail_Nonclustered – create with a non-clustered index on LineTotal,ProductId fields
   iv. SalesOrderDetail_ColumnStore – create ColumnStore index on LineTotal,ProductId fields

   This piece of code from the script shows how we can create a Columnstore index:

```
CREATE COLUMNSTORE INDEX Idx_SalesOrderDetail_ColumnStore
  ON [dbo].[SalesOrderDetail_Columnstore](LineTotal, ProductId);
```

   Note: This creates a NON CLUSTERED index.

4. Run the attached ‘Select_queries’ script with ‘Include Actual Execution Plan’ option selected in the editor. This runs the same query on all the four tables, in a single batch.

This is a query run on one of the tables – pulled from the script for illustration:

```
SELECT
   ProductId
 , SUM(LineTotal) 'Total Sales Amount'
 , MIN(LineTotal) 'Min Order Amount'
 , AVG(LineTotal) 'Average Sales'
 FROM [dbo].[SalesOrderDetail_Columnstore]
 GROUP BY ProductId ORDER BY ProductId;
```

5. Observe the results tabs

The result for each table can be seen as below – Note the logical reads & relative query cost in each case:

SalesOrderDetail:

```
(266 row(s) affected)
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical r
Table 'SalesOrderDetail'. Scan count 1, logical reads 1494, physical reads 0, read-ahead reads 0, lob logical reads 0, lob
```

Query cost relative to the batch: ***33***

SalesOrderDetail_ClusteredIndex:

```
(266 row(s) affected)
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical rea
Table 'SalesOrderDetail_ClusteredIndex'. Scan count 1, logical reads 1515, physical reads 0, read-ahead reads 0, lob logical

(1 row(s) affected)
```

Query cost relative to the batch: ***33***

SalesOrderDetail_NonClusteredIndex:

```
(266 row(s) affected)
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physic
Table 'SalesOrderDetail_NonClusteredIndex'. Scan count 1, logical reads 533, physical reads 0, read-ahead reads 0, lob

(1 row(s) affected)
```

Query cost relative to the batch: ***20***

SalesOrderDetail_Columnstore:

```
(266 row(s) affected)
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical r
Table 'SalesOrderDetail_Columnstore'. Scan count 1, logical reads 63, physical reads 0, read-ahead reads 0, lob logical rea

(1 row(s) affected)
```

Query cost relative to the batch: ***13***

As is evident from the results, our Columnstore query performs more optimally compared to the other scenarios. In fact, it is more efficient than the corresponding non-clustered index created on the same fields.

The logical IO for the query using Columnstore is significantly less because the query has to now read fewer number of pages to get all data needed for its execution and this translates to faster queries.

Benefits

A Columnstore index works well in scenarios where there is regular and frequent access of a smaller percentage of the overall columns in a table. Fewer columns needs fewer pages to be read and results in reduced IO & improved query execution times.

A Columnstore is the ideal fit for storing fact tables in a data warehouse. For summarizing/aggregating scenarios in a Data warehouse, pre-computed summary tables are created – sometimes one for each problem. These are inflexible and difficult to maintain – at times, even minor modifications in the query may require a re-creation of summary tables. A Columnstore works best in such scenarios – it can accommodate query modifications too, without much ado.

A Columnstore can benefit from the potential application of compression techniques due to the higher likelihood of values repeating within the same column. This in turn results in lesser number of pages needed to fit column data.

Summary

The Columnstore index is a definitive boon to the data warehousing community that had, until recently, resigned themselves to the world of slower queries and longer wait times. With Columnstore indexes, near-instantaneous response time for queries on huge tables is no longer unimaginable.
