Columnar Storage in SQL Server 2012

Per-Ake Larson
palarson@microsoft.com

Eric N. Hanson
ehans@microsoft.com

Susan L. Price
susanp@microsoft.com

Abstract

SQL Server 2012 introduces a new index type called a column store index and new query operators that efficiently process batches of rows at a time. These two features together greatly improve the performance of typical data warehouse queries, in some cases by two orders of magnitude. This paper outlines the design of column store indexes and batch-mode processing and summarizes the key benefits this technology provides to customers. It also highlights some early customer experiences and feedback and briefly discusses future enhancements for column store indexes.

1 Introduction

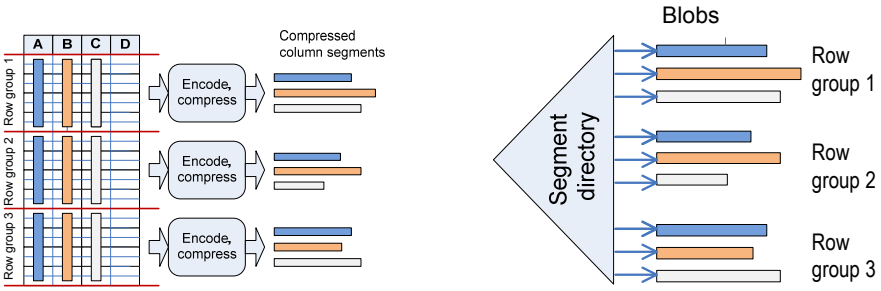
SQL Server is a general-purpose database system that traditionally stores data in row format. To improve performance on data warehousing queries, SQL Server 2012 adds columnar storage and efficient batch-at-a-time processing to the system. Columnar storage is exposed as a new index type: a column store index. In other words, in SQL Server 2012 an index can be stored either row-wise in a B-tree or column-wise in a column store index. SQL Server column store indexes are "pure" column stores, not a hybrid, because different columns are stored on entirely separate pages. This improves I/O performance and makes more efficient use of memory.

Column store indexes are fully integrated into the system. To improve performance of typical data warehousing queries, all a user needs to do is build a column store index on the fact tables in the data warehouse. It may also be beneficial to build column store indexes on extremely large dimension tables (say more than 10 million rows). After that, queries can be submitted unchanged and the optimizer automatically decides whether or not to use a column store index exactly as it does for other indexes. Some queries will see significant performance gains - even as much as 100X - while others will show smaller or no gains.

The idea of storing data column-wise goes back to the seventies. In 1975 Hoffer and Severance [3] investigated how to decompose records into smaller subrecords and storing them in separate files. A 1985 paper by Copeland and Khoshafian [2] proposed fully decomposed storage where each column is stored in a separate file. The development of MonetDB, a column store pioneer, began in the early nineties at CWI [4]. Sybase launched Sybase IQ, the first commercial columnar database system, in 1996. More recent entrants include Vertica, Exasol, Paraccel, InfoBright and SAND.

SQL Server is the first general-purpose database system to fully integrate column-wise storage and processing into the system. Actian Vectorwise Analytical Database (from Actian Corporation) is a pure column store and engine embedded within the Ingres DBMS but it does not appear to interoperate with the row-oriented Ingres engine, that is, a query cannot access data both in the Vectorwise column store and the standard Ingres row store

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering



(a) Convert rows to column segment

(b) Store segments in blobs

Figure 1: Creation of a column store index

[5]. Greenplum Database (from EMC Greenplum) and Aster Database (from Teradata's Aster Data) began as pure row stores but have now added column store capabilities. However, it is unclear how deeply column-wise processing has been integrated into the database engines. Teradata has announced support for columnar storage in Teradata 14 but, at the time of writing, this has not yet been released.

2 Technical Overview

SQL Server has long supported two storage organization: heaps (unordered) and B-trees (ordered), both row-oriented. A table or a materialized view always has a primary storage structure and may have additional secondary indexes. The primary structure can be either a heap or a B-tree; secondary indexes are always B-trees. SQL Server also supports filtered indexes, that is, an index that stores only rows that satisfy a selection predicate.

Column store capability is exposed as a new index type: a column store index. A column store index stores its data column-wise in compressed form and is designed for fast scans. The initial implementation has restrictions but, in principle, any index can be stored as a column store index, be it primary or secondary, filtered or non-filtered, on a base table or on a view. A column store index can, in principle, support all the same index operations (scans, lookups, updates, and so on) that heaps and B-tree indexes support. All index types can be made functionally equivalent but they do differ in how efficiently various operations can be performed.

2.1 Column-wise Index Storage

Figure 1 illustrates how a column store index is created and stored. The first step is to convert a set of rows to column segments. The rows are first divided into row groups of about one million rows each. Each row group is then encoded and compressed independently, producing one compressed column segment for each column included in the index. Figure 1(a) shows a table divided into three row groups where three of the four columns are included in the index. This yields nine compressed column segments, three segments for each of columns A, B, and C. Further details about encoding and compression can be found in reference [6].

The column segments are then stored using existing SQL Server storage mechanisms as shown in Figure 1(b). Each column segment is stored as a separate blob (LOB). Segment blobs may span multiple disk pages but this is automatically handled by the blob storage mechanisms. A segment directory keeps track of the location of segments so all segments comprising a column can be easily located. The directory contains additional metadata about each segment such as number of rows, size, how data is encoded, and min and max values. Dictionary compression is used for (large) string columns and the resulting dictionaries are stored in separate blobs.

Storing the index in this way has several important benefits. It leverages the existing blob storage and catalog implementation - no new storage mechanisms - and many features are automatically available. Locking, logging, recovery, partitioning, mirroring, replication and other features immediately work for the new index type.

2.2 I/O and Caching

Column segments and dictionaries are brought into memory as needed. They are stored not in the page-oriented buffer pool but in a new cache designed for handling large objects (columns segments, dictionaries). Each object in the cache is stored contiguously and not scattered across discrete pages. This simplifies and speeds up scanning of a column because there are no "page breaks" to worry about.

A segment storing a blob may span multiple disk pages. To improve I/O performance, read-ahead is applied aggressively both within and among segments. In other words, when reading a blob storing a column segment, read-ahead is applied at the page level. A column may consist of multiple segments so read-ahead is also applied at the segment level. Finally, read-ahead is also applied when loading data dictionaries.

2.3 Batch Mode Processing

Standard query processing in SQL Server is based on a row-at-a-time iterator model, that is, a query operator processes one row at a time. To reduce CPU time a new set of query operators were introduced that instead process a batch of rows at a time. They greatly reduce CPU time and cache misses on modern processors when processing a large number of rows.

A batch typically consists of around a thousand rows. As illustrated in Figure 2, each column is stored as a contiguous vector of fixed-sized elements. The "qualifying rows" vector is used to indicate whether a row has been logically deleted from the batch. Row batches can be processed very efficiently. For example, to evaluate a simple filter like $Col1 < 5$, all that's needed is to scan the Col1 vector and, for each element, perform the comparison and set/reset a bit in the "qualifying rows" vector. As convincingly shown by the MonetDB/X100 project [1], this type of simple vector processing is very efficient on modern hardware; it enables loop unrolling and memory prefetching and minimizes cache misses, TLB misses, and branch mispredictions.

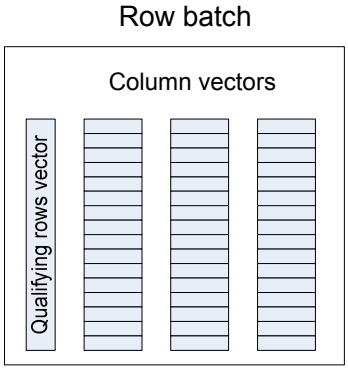


Figure 2: A row batch object

In SQL Server 2012 only a subset of the query operators are supported in batch mode: scan, filter, project, hash (inner) join and (local) hash aggregation. The hash join implementation consists of two operators: a (hash table) build operator and an actual join operator. In the build phase of the join, multiple threads build a shared in-memory hash table in parallel, each thread processing a subset of the build input. Once the table has been built, multiple threads probe the table in parallel, each one processing part of the probe input.

Note that the join inputs are not pre-partitioned among threads and, consequently, there is no risk that data skew may overburden some thread. Any thread can process the next available batch so all threads stay busy until the job has been completed. In fact, data skew actually speeds up the probing phase because it leads to higher cache hit rates.

The reduction in CPU time for hash join is very significant. One test showed that regular row-mode hash join consumed about 600 instructions per row while the batch-mode hash join needed about 85 instructions per row and in the best case (small, dense join domain) was a low as 16 instructions per row. However, the initial version of batch-mode hash join has limitations: the hash table must fit entirely in memory and it supports only inner join. These limitations will be addressed in future releases.

The scan operator scans the required set of columns from a segment and outputs batches of rows. Certain filter predicates and bitmap filters are pushed down into scan operators. (Bitmap filters are created during the build phase of a hash join and propagated down on the probe side.) The scan operator evaluates the predicates directly on the uncompressed data, which can be significantly cheaper and reduces the output from the scan.

The query optimizer decides whether to use batch-mode or row-mode operators. Batch-mode operators are typically used for the data intensive part of the computation, performing initial filtering, projection, joins and aggregation of the inputs. Row-mode operators are typically used on smaller inputs, higher up in the tree to finish the computation, or for operations not yet supported by batch-mode operators.

For queries that process large numbers of rows, the net result of the improvements in query processing is an order of magnitude better performance on large OLAP queries.

3 Customer Experiences

A customer using SQL Server 2012 on a star schema database with a two billion row fact table achieved a remarkable speedup of over 200 times. Their original strategy with a prior version of SQL Server was to run reports at night and cache the results for users to retrieve during the business day. The nightly report generation process took 18 hours, running on a four processor, 16 core machine with 512GB RAM and a good I/O system. After upgrading to SQL Server 2012 and creating a column store index on the fact table, they were able to generate the nightly reports in 5 minutes, on the same hardware. Individual queries that scan the fact table now run in about three seconds each, versus up to 17 minutes each before using column store indexes. They are thus going to give their users the ability to do interactive reporting rather than merely allowing them to retrieve pre-generated reports, clearly adding business value.

A Microsoft IT group that manages a data warehouse containing financial information ran their standard test set of 133 real end-user queries before and after building column store indexes on 23 fact tables. Two-thirds of the queries ran faster - three queries by as much as 50X. The number of queries running longer than ten minutes decreased by 90% (from 33 to 3). The queries that benefited from column store indexes were the queries that are most painful for end users - the longest-running queries. All but one of the queries that did not improve were queries that ran in 40 seconds or less. Only one of the queries with a fast baseline regressed to longer than 40 seconds (to about 42 seconds).

The average compression ratio (size of base table/size of column store index on all columns) was 3.7. Ten of the 23 tables were already using SQL Server ROW compression. Compared to the base table plus existing non-clustered B-tree indexes, the column store index was 11.2X smaller (i.e. total size of row-based structures/size of column store index = 11.2).

Another Microsoft IT group that runs a data warehouse containing current and historical employee data, created column store indexes on their fact tables and larger dimension tables. Average query response time dropped from 220 seconds to 66 seconds. In addition, the team was able to eliminate most of the row-based indexes on the tables with column store indexes. Another early user reported that creating a column store index sped up an ETL query to extract data from a billion-row source table from 15 minutes to 3 minutes.

We have begun formulating best practices based on our early customers' experiences.

1. Use a star schema when possible. We have optimized query execution for star-join style queries.
2. Include all the columns in the column store index. Although it is possible to look up data from missing columns in the base table, such usage is not very efficient and query performance will suffer. The cost of including all columns is very small since only columns touched by the query are retrieved from disk.
3. Ensure that enough memory is available to build the column store index. Creating the index is memory-intensive, especially for wide tables.
4. Create the column store index from a clustered index to get segment elimination for queries with predicates on the clustering key. Although the column store index is not ordered, the clustered index is scanned in index order, naturally resulting in data locality when rows are assigned to row groups.

5. Check query plans for use of batch mode processing and consider tuning queries to get increased benefit of batch mode processing. Because not all operators can be executed in batch mode yet, query rewriting can sometimes yield large performance gains.

Customer feedback has identified several improvements that would be highly beneficial such as extending the repertoire of batch-mode operators, especially outer join, union all, scalar aggregates, and global aggregation. SQL Server includes many performance optimizations for row-mode queries. To provide exceptional performance across a wide range of scenarios, batch-mode processing needs to match many of those optimizations. For example, we currently push some filters into the column store scan; customers would like us to match or exceed the range of filter types that are pushed in row-mode queries. Also, B-tree indexes can be rebuilt online so this option should be provided for column store indexes too.

4 Customer Benefits

The dramatically improved query performance enabled by column store indexes provides significant benefits to customers. Most importantly, it allows a much more interactive and deeper exploration of data which ultimately leads to better insight and more timely and informed decisions.

It has been common practice to use summary aggregates, whether in the form of materialized views or user-defined summary tables, to speed up query response time. The improved performance using column store indexes also means that the number of summary aggregates can be greatly reduced or eliminated completely. Furthermore, OLAP cubes, if they had been used strictly to improve query performance due to their internal summary aggregate maintenance and aggregate navigators, also may be eliminated in favor of SQL reporting directly on the DBMS.

Users typically don't have the patience to wait for more than half a minute for a query result. Hence, reporting applications need to pre-prepare reports if they run more slowly than this. In addition, users will modify their information requests to accommodate the system almost subconsciously to get faster response. For example, they will ask for a summary of activity on one day a month for the last six months instead of a summary of activity every day for the last six months, if that lets the query run in seconds instead of minutes. The column store index mechanism can allow them to get the interactivity they want for the question they really want to ask. In summary, the key customer benefits of column store indexes and more efficient query execution are as follows.

1. Faster data exploration leading to better business decisions.
2. Lower skill and time requirements; designing indexes, materialized views and summary tables requires time and a high level of database expertise.
3. Reduced need to maintain a separate copy of the data on an OLAP server.
4. Faster data ingestion due to reduced index and aggregate maintenance.
5. Reduced need to move to a scale-out solution.
6. Lower disk space, CPU, and power requirements.
7. Overall lower costs.

The use of an OLAP (BI) mechanism still will be warranted if it makes the end-user reporting environment richer, and the business value of that outweighs IT cost savings. We expect the ability of the relational DBMS to run data warehouse queries with interactive response time will drive additional effort into relational OLAP (ROLAP) systems. Such systems can provide rich interactive data exploration environments on top of a single copy of the data warehouse or mart maintained by the relational DBMS.

5 Future Enhancements

For reasons of scope and schedule, the implementation of column store indexes had to be scoped down in the initial release, leaving important features unsupported. These limitations will be addressed in future releases though we cannot at this stage disclose on what schedule.

Direct update and load of a table with a column store index is not supported in the initial release. Even so, data can be added to such a table in a number of ways. If the table is not too large, one can drop its column store index, perform updates, and then rebuild the index. Column store indexes fully support range partitioning. So for large tables, the recommended way is to use partitioning to load a staging table, index it with a column store index, and switch it in as the newest partition. SQL Server 2012 allows up to 15,000 partitions per table so this approach can handle many loads per day, allowing data to be kept current.

In SQL Server, the primary organization of a table can be either a heap or a B-tree. However, a column store index cannot be used as the primary organization in this release; they can only be used for secondary indexes. This restriction may in some cases result in wasted disk space so it will be lifted.

Batch-mode processing is crucial to realize the full performance gains but the initial repertoire of batch-mode operators is limited. Batch-mode hash join will be extend to support all join types (inner, outer, semijoin, anti-semijoin) and additional operators will be implemented in batch-mode.

6 Acknowledgements

Many people have contributed to the success of this project. We thank Ted Kummert and the senior leadership team in SQL Server for their ongoing support and sponsorship of the project. Amir Netz and Cristian Petulescu generously shared their ideas and insights from PowerPivot. Hanuma Kodavalla initiated the project and urged us to show that "elephants can dance". Development was lead by Srikumar Rangarajan with Aleksandras Surna, Artem Oks, and Cipri Clinciu contributing major pieces and Qingqing Zhou monitoring and tracking performance.

References

[1] P. Boncz, M. Zukowski, and N. Nes. MonetDB/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[2] G. P. Copeland and S. Khoshafian. A decomposition storage model. In *SIGMOD Conference*, pages 268–279, 1985.

[3] J. A. Hoffer and D. G. Severance. The use of cluster analysis in physical data base design. In *VLDB*, pages 69–86, 1975.

[4] M. Holsheimer and M. L. Kersten. Architectural support for data mining. In *KDD Workshop*, pages 217–228, 1994.

[5] D. Inkster, M. Zukowski, and P. Boncz. Integration of vectorwise with Ingres. *SIGMOD Record*, 40(3):45–53, 2011.

[6] P.-Å. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL server column store indexes. In *SIGMOD Conference*, pages 1177–1184, 2011.