Enhancements to SQL Server Column Stores

Per-Åke Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, Mayukh Saubhasik
Microsoft

{palarson, ciprianc, cfraser,ehans, mostafm, michalno, vasilp, srikumar, remusr, maysau }@microsoft.com, prices08@gmail.com

ABSTRACT
SQL Server 2012 introduced two innovations targeted for data warehousing workloads: column store indexes and batch (vectorized) processing mode. Together they greatly improve performance of typical data warehouse queries, routinely by 10X and in some cases by a 100X or more. The main limitations of the initial version are addressed in the upcoming release. Column store indexes are updatable and can be used as the base storage for a table. The repertoire of batch mode operators has been expanded, existing operators have been improved, and query optimization has been enhanced. This paper gives an overview of SQL Server's column stores and batch processing, in particular the enhancements introduced in the upcoming release.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *relational databases, Microsoft SQL Server*

General Terms
Algorithms, Performance, Design

Keywords
Columnar storage, column store, index, OLAP, data warehousing.

1. INTRODUCTION
SQL Server has long supported two storage organizations: heaps and B-trees, both row-oriented. SQL Server 2012 introduced a new index type, column store indexes, where data is stored column-wise in compressed form. Column store indexes are intended for data-warehousing workloads where queries typically process large numbers of rows but only a few columns. To further speed up such queries, SQL Server 2012 also introduced a new query processing mode, batch processing, where operators process a batch of rows (in columnar format) at a time instead of a row at a time.

Customers have reported major performance improvements when using column store indexes. One customer achieved a speedup of over 200X on a star schema database with a fact table containing two billion rows. They ran a nightly report generation process that took 18 hours. After upgrading to SQL Server 2012 and creating a column store index on the fact table, they were able to generate the reports in 5 minutes on the same hardware. Some queries that scan the fact table now run in about three seconds each compared with up to 17 minutes previously. References to this and other case studies can be found in a column store index FAQ [11].

The initial implementation in SQL Server 2012 had several limitations that are remedied in the upcoming release: column store indexes are updatable, they can be used as the primary storage of a table, they can be further compressed to save disk space, and batch-mode processing has been significantly extended and enhanced.

This paper describes the main enhancements of column stores indexes and batch processing in the upcoming release of SQL Server. For completeness, we begin with an overview of the basics of column store indexes and batch processing in section 2. Section 3 outlines the extensions needed to allow a column store index to be used as the primary or base storage for a table. Section 4 covers how inserts, deletes and updates of a column store index are handled. Enhancements to query optimization and query processing are discussed in section 5. Archival compression is covered in section 6. Some performance results are presented in section 7.

2. BACKGROUND
This section describes the basic structure of column store indexes and how they are constructed and stored. It also outlines how batch processing works. More detailed information can be found in the paper on the initial implementation [9], product documentation [10], and the column store index FAQ [11].

2.1 Index Storage
Figure 1 illustrates how a column store index is created and stored. The set of rows is first divided into row groups of about one million rows each. Each row group is then encoded and compressed independently and in parallel, producing one compressed column segment for each column included in the index. For columns that use dictionary encoding the conversion may also produce a number of dictionaries. Note that data in a column store index is not sorted, not even within a column segment.

Figure 1 shows a table with three columns divided into three row groups. The conversion produces nine compressed column segments, three segments for each of columns A, B, and C. Column A used dictionary encoding so the output also includes three dictionaries, one for each segment of column A. More details about encoding and compression can be found in our earlier paper [9].

The column segments and dictionaries are then stored using existing SQL Server storage mechanisms as illustrated on the right side of Figure 1. Each column segment and dictionary is stored as a separate blob (LOB). A blob may span multiple disk pages but this is automatically handled by the blob storage mechanisms.

SIGMOD'13, June 22-27, 2013, New York, New York, USA.

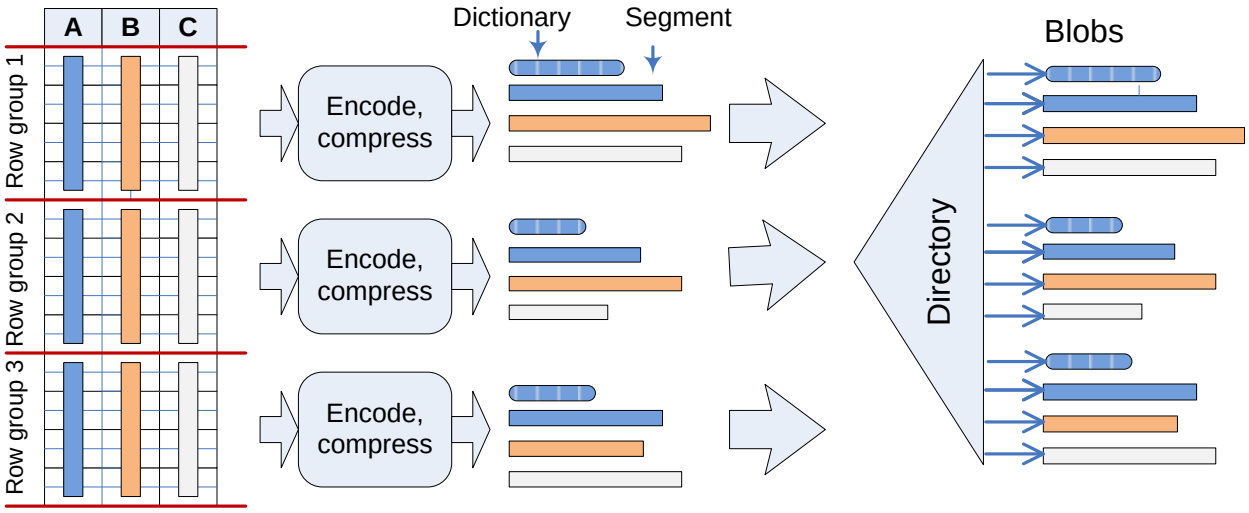

Figure 1: Illustrating how a column store index is created and stored. The set of rows is divided into row groups that are converted to column segments and dictionaries that are then stored using SQL Server blob storage

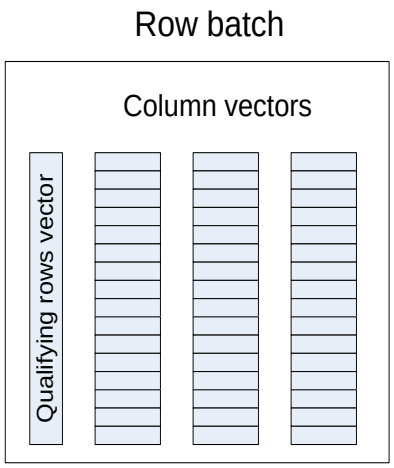

Figure 2: A row batch is stored column-wise and contains one vector for each column plus a bit vector indicating qualifying rows

A directory keeps track of the location of segments and dictionaries so all segments comprising a column and any associated dictionaries 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.

Storing the index in this way has several benefits. It leverages the existing blob storage and catalog implementation - no new storage mechanisms are needed - and many features are automatically enabled for the new index type, for example, storage space management, logging, recovery, and high availability.

2.2 Caching and I/O

Column segments and dictionaries are brought into memory as needed during query processing. They are not stored in the buffer pool but in a new cache for large objects. Each object is stored contiguously on adjacent memory pages. This simplifies and speeds up scanning of a column because there are no "page breaks" to worry about.

A blob storing a column segment or dictionary 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. Read-ahead is of course also applied when loading data dictionaries.

2.3 Batch Mode Processing

SQL Server traditionally uses a row-at-a-time execution model, that is, a query operator processes one row at a time. Several new query operators were introduced that instead process a batch of rows at a time. This greatly reduces CPU time and cache misses 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 indicates whether a row has been logically purged from the batch.

Row batches can be processed very efficiently. For example, to evaluate a simple filter like “Col1 < 5”, all that is 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 shown by the MonetDB/X100 project [2], this type of vector processing is very efficient on modern hardware; it enables loop unrolling and memory pre-fetching and minimizes cache misses, TLB misses, and branch mispredictions.

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 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 SQL Server 2012 implementation has limitations: the hash table must fit entirely in memory and it supports only inner join.

The scan operator scans the required set of columns from a segment and outputs batches of rows. Certain filter predicates and bitmap (Bloom) 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 compressed 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 supported by batch-mode operators.

3. CLUSTERED INDEXES

In SQL Server 2012 column store indexes could only be used as secondary indexes so data was duplicated. One copy of the data would be in the primary storage structure (heap or B-tree) and another copy would be in a secondary column store index. The upcoming release of SQL Server removes this restriction and allows a column store to be the primary and only copy of the data. Although column store data is not really 'clustered' on any key, we decided to retain the traditional SQL Server convention of referring to the primary index as a clustered index. This section briefly touches on the SQL Server engine enhancement needed to enable this feature.

3.1 Improved Index Build

The way a column store index is built has been improved to make the process more dynamic and improve the quality of the index.

SQL Server column stores use a form of dictionary encoding wherein frequently occurring values are mapped to a 32 bit id via the dictionary. SQL Server uses two forms of dictionaries, a global dictionary associated with the entire column and a local dictionary associated with a row group. The earlier implementation would fill in the entries in the global dictionary as it built the index. This did not guarantee that the most relevant values went into the global dictionary.

We modified the query plan to build the index in two steps. The first step samples the data for each column, decides whether or not to build a global dictionary for a column, and picks the set of values to include in the global dictionary. This ensures that the most relevant values are included in the global dictionary. The second step actually builds the index using the global dictionaries constructed in the first step.

The column store build process is quite memory intensive, hence we do a memory reservation up-front before starting the build process. Each thread needs enough memory to hold a full row group plus sufficient scratch space. Depending on the initial memory reservation and the memory estimate for each thread we then choose the number of threads that will participate in the build process. The number of threads does not change after we've made this initial determination. This static degree of parallelism (DOP) can sometimes cause the build process to use a suboptimal number of threads to build the index, because the initial memory estimate can be quite inaccurate. It is difficult to accurately estimate the memory requirement without looking at the actual data, because the memory required depends on the data distribution.

We solved this problem by enabling the build process to dynamically vary the number of threads that actively participate in the build. The build process continually monitors the memory being consumed by each thread and the amount of memory available to the query to calculate the optimal number of active threads.

3.2 Sampling Support

The SQL Server query optimizer uses statistics about the data distribution of columns involved in a query to generate the query plan. The statistics consists of histograms that are computed from a random sample of rows. To enable this scenario we implemented sampled scans on column stores. Non-clustered column stores did not need to support sampling because the statistics could be computed from the base data.

We implemented two forms of sampled scans. One implementation is optimized for performance while giving up some accuracy, whereas the second algorithm is highly accurate but has a higher IO and CPU cost.

The performance optimized sampled scan uses cluster sampling: a set of row groups is first randomly selected, followed by a random sample of rows within each group. The number of rows and row groups selected are mandated by the sampling percentage. Row groups that are not selected in the initial step are not read from disk. Sampling from a B-tree or heap also uses cluster sampling and selects a random subset of pages.

The second form of sampling is truly random row level sampling. This implementation scans all segments of a column and randomly selects a subset of rows. Truly random row sampling is always used when building a histogram from a column store for later use in query optimization. This produces more accurate histograms than for B-trees and heaps which use page sampling. Cluster sampling of the columnstore is only used to help with dictionary creation during the index build process, never to create histograms for query optimization.

3.3 Bookmark Support

In SQL Server terminology a bookmark is a value that uniquely identifies a row: a logical row pointer. The actual bookmark type depends on the clustered index; if it is a heap, the bookmark consists of <page ID, row ID> and if it is a B-tree it is the B-tree key, possibly augmented with an additional uniquifier column.

Any index in SQL Server storing the primary copy of a table must be able to locate a row given a bookmark. Bookmarks are used by a variety of query plans, the most frequent of which is a delete plan which first collects the bookmarks for a set of rows to delete before actually deleting them.

So a clustered column store index also needs to support bookmark lookup. Since a column store index does not have a key that uniquely identifies a row, we associated a unique tuple id with each row within a row group (simply its sequence number, which is not stored) and used the combination of row group id and tuple id to uniquely identify a row.

3.4 Other Enhancements

The SQL Server 2012 implementation did not support a number of data types such as numeric beyond precision 18, datetimeoffset beyond precision 2, GUID and binary columns. The upcoming version adds support for all the above data types. It also introduces support for storing short strings by value instead of converting all strings to a 32 bit id within a dictionary. This removes the extra overhead associated with the dictionary and helps improve the column store compression even further.

We also extended a number of other features and added several new features to improve feature parity between the column store indexes and row store indexes. For example, it is possible to add, drop and modify the set of columns that are part of a clustered column store, unlike the nonclustered columnstore index which does not allow such changes. We extended the data check functionality for column stores to do more in-depth data validity checks, and shrink file and shrink database functionality now work with column store indexes.

4. UPDATE HANDLING

Columnar storage greatly improves read performance but once compressed, the data is prohibitively expensive to update directly. Some compressed formats allow for append-only updates, one row at a time, but they achieve lower compression ratios than techniques that compress large blocks (segments) of data at a time. Compression is more effective if a significant amount of data is compressed at once, allowing the data to be analyzed and the best compression algorithm chosen.

The prime target for columnar storage is data warehouse fact tables which typically have a high rate of inserts and a very low rate of updates or deletes. Data past its retention period can be removed in bulk almost instantly from a partitioned table via partition switching. This being the case, it is crucial to achieve high performance for regular insert and bulk insert operations while the performance of update and delete operations is less critical.

Two new components were added to make SQL Server column store indexes updatable: delete bitmaps and delta stores.

Each column store index has an associated **delete bitmap** that is consulted during scans to disqualify rows that have been deleted. A bitmap has different in-memory and on-disk representations. In memory it is indeed a bitmap but on disk it is represented as a B-tree with each record containing the row ID of a row that was deleted.

New and updated rows are inserted into a **delta store** which is a traditional B-tree row store. An index may have multiple delta stores. Delta stores are transparently included in any scan of the column store index.

With this infrastructure in place, insert, delete and update operations on column stores become possible.

- **Insert**: The new rows are inserted into a delta store. This can be done efficiently because delta stores are traditional B-tree indexes.

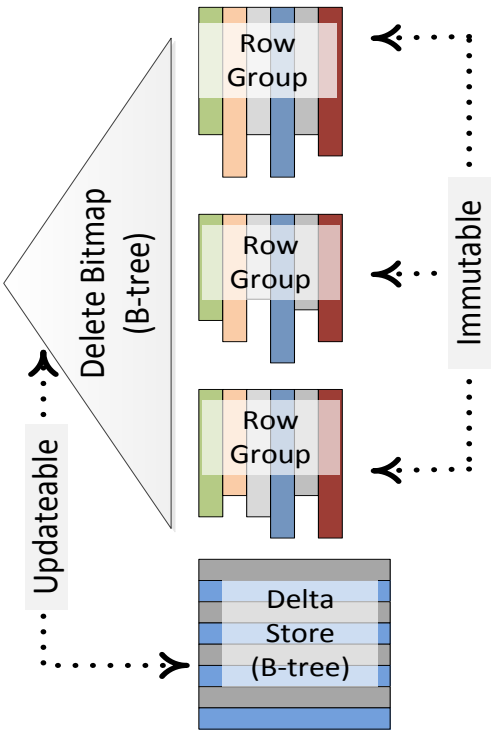

Figure 3: A column store index may include a delete bitmap and a number of delta stores which are both implemented as B-trees.

- **Delete**: If the row to be deleted is in a column store row group, a record containing its row ID is inserted into the B-tree storing the delete bitmap. If it is in a delta store, the row is simply deleted.
- **Update**: An update operation is simply split into a delete and an insert action. That is, the old row is marked as deleted and the new row is inserted into a delta store.
- **Merge**: A merge operation is split into corresponding insert, delete, or update actions.

A delta store contains the same columns as the corresponding column store index. The B-tree key is a unique integer row ID generated by the system (column stores do not have unique keys).

A column store can have zero, one, or more delta stores. New delta stores are created automatically as needed to accept inserted rows. A delta store is either open or closed. An open delta store can accept rows to be inserted. A delta store is closed when the number of rows it contains reaches a predefined limit.

SQL Server automatically checks in the background for closed delta stores and converts them to columnar storage format. This periodic background task is called the Tuple Mover. After the Tuple Mover has converted a delta store, it is deallocated. The Tuple Mover does not block read scans but concurrent deletes are forced to wait for the conversion to complete. The Tuple Mover can also be invoked on demand.

The Tuple Mover reads one closed delta store at a time and starts building the corresponding compressed segments. During this time scans continue to see and read the delta store. When the Tuple Mover has finished compressing the delta store, the newly created

segments are made visible and the delta store is made invisible in a single, atomic operation. New scans will see and scan the compressed format. The Tuple Mover then waits for all scans still operating in the delta store to drain after which the delta store is removed.

Because the Tuple Mover does not block reads and inserts, it has minimal impact on overall system concurrency. Only concurrent delete and update operations against a delta store may be blocked while the delta store is converted by the Tuple Mover.

4.1 Trickle Inserts

Normal, non-bulk insert operations – here called trickle inserts - are handled by transparently intercepting the data to be inserted into a column store index and writing it instead into a delta store. The Query Processing layer is unaware of this and operates as if it had inserted the data into a column store. The internal Access Methods layer is responsible for managing the delta stores, for including their contents in column store scans, for creating delta stores and for locating a delta store when needed.

4.2 Bulk Inserts

Large bulk insert operations do not insert rows into delta stores but convert batches of rows directly into columnar format. The operation buffers rows until a sufficient number of rows has accumulated, converts them into columnar format, and writes the resulting segments and dictionaries to disk. This is very efficient; it reduces the IO requirements and immediately produces the columnar format needed for fast scans. The downside is the large memory space needed to buffer rows.

Bulk insert operations must compress and close the data when the statement finishes. The batch_size parameter of the bulk insert API determines how many rows are processed by one bulk insert statement. For efficient compression a large batch size of 1M rows is recommended.

But even with a large batch size there may be cases where there simply are not enough rows to justify a compressed row group. SQL Server handles this situation transparently by automatically switching to a delta store if a statement finishes without having enough rows accumulated. This simplifies the programming and administration of column store ETL operations because there is no requirement to force a large volume of rows to accumulate in the ETL pipeline. SQL Server will decide the proper storage format and, if the data was saved in the row store format, it will later compress the delta store when it has accumulated enough rows. The number of rows in a row group is kept high (around one million) to achieve high compression ratios, and maintain a large column store segment size to yield fast, low-overhead query processing.

Given the significant performance advantage of the bulk insert API over traditional insert statements, the query execution engine will internally use the bulk insert API for "insert into … select from …" statements that are targeting a column store. This operation is fairly frequent in ETL pipelines and this change allows it to create directly the highly efficient compressed format.

4.3 Deletes and Updates

Delete operations operate differently on compressed row groups vs. delta stores. For rows stored in compressed format, a delete operation will insert the row ID (ordinal position number within the row set) of the deleted row in the deleted bitmap. For rows stored in a delta store, the row is removed because B-trees support efficient row removal. Recall that an update operation is executed as a delete operation and an insert operation.

4.4 Effect on Query Execution

The handling of delta stores and the deleted bitmap is done entirely in the Access Methods layer. The Query Execution layer will see the compressed data and the delta store data as one uniform set of rows that has columnar storage characteristics, exposed for batch processing. This allows the system to maintain the orders-of-magnitude query speedups possible with column stores and batch mode, even with the introduction of row-based delta stores.

The deleted bitmap is consulted by scans and if the current row id is present in the deleted bitmap, the row is skipped. This is handled internally in the Access Methods layer; thus deleted rows are never surfaced to query processing.

The process of segment elimination during scans (by checking segment metadata containing the minimum and maximum values in columns) does not need to consult the deleted bitmap. The interval between the minimum and maximum values within a column cannot grow when rows are deleted. Therefore, the original minimum and maximum values computed during column store segment creation can safely be used for segment elimination even after deletes.

A large number of deleted rows can reduce scan performance. The cost of IO for reading the segments is not reduced by deleting rows because compressed segments are immutable and the cost of reading and consulting the bitmap is added. The workloads targeted by columnar storage typically have a low frequency of updates and deletes so having many deleted rows rarely occurs. Column store indexes with a large volume of deleted rows can be rebuilt to restore performance and reclaim space.

Parallel scans assign each delta store to a single thread of execution. A single delta store is too small to justify scanning in parallel but multiple delta stores can be scanned in parallel. Scanning delta stores is slower than scanning data in columnar format because complete records have to be read and not just the columns needed by the query. This is mitigated by the relatively small size of delta stores and by the fact that the engine actively converts delta store data into the compressed columnar format. This keeps the number of delta stores low, so under normal operating conditions, the great majority of data stays in columnar format, optimized for efficient space usage and query execution.

5. QUERY PROCESSING AND OPTIMIZATION

Batch processing in SQL Server 2012 supported only the most heavily used query patterns in data warehousing scenarios, for example, inner but not outer joins, and group-by-aggregate but not scalar aggregates. Query plan segments using batch processing had a rigid shape; the join order was fixed and generated heuristically based on cardinality estimates. Query plans had to prepare for the possibility of a "bailout" to row-by-row processing, in case of insufficient memory during execution.

The upcoming release of SQL Server extends batch processing capabilities in several ways. We consider batch execution for iterators anywhere in the query plan, regardless of whether their inputs are using batch execution, and regardless of whether the data originates in a column store or row store. The join order for batch execution is no longer a fixed one generated heuristically. Batch processing is supported for all SQL Server join types, union all, and

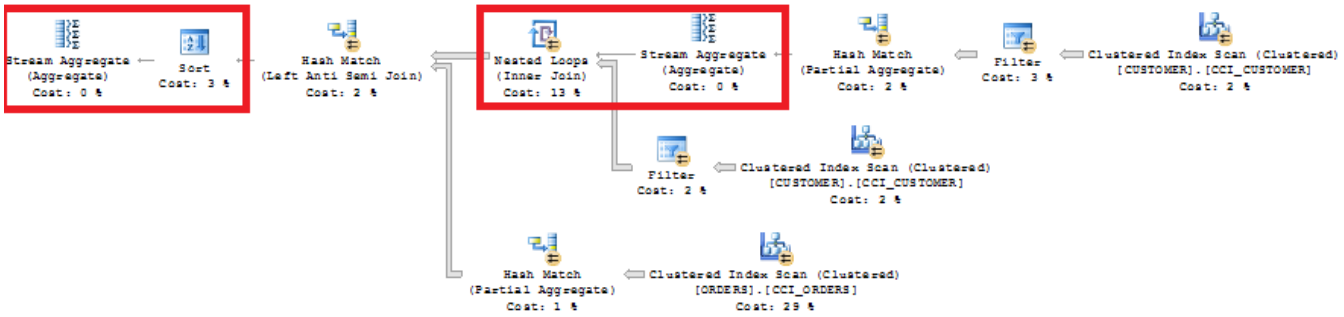

Figure 4: Query plan with mixed batch and row mode processing. The operators enclosed in red boxes run in row mode, the rest in batch mode.

scalar aggregation. Finally, we have done extensive work on memory management and spilling support for batch mode operators.

5.1 Mixed Execution Mode

In SQL Server 2012 the transition between row and batch processing happens only at prescribed points in the plan, and the transition between batch and row processing occurs only when absolutely necessary. This limitation is cumbersome and the forthcoming release of SQL Server has a completely new model for dealing with batch to row mode transitions.

Like the sortedness of output rows, the execution mode is treated as a physical property of a query plan iterator, and the physical property framework is used to manage execution mode transitions. It is now possible to transition from one execution mode to another at any point in an execution plan. These transitions have a cost associated with them, thus ensuring that the optimizer does not become too transition-happy. The ability to freely transition between batch and row modes allows the optimizer to explore the logical space only once, and moves all decisions related to execution model into the implementation phase.

The example plan in Figure 4 illustrates how it is now possible to have mixed execution modes. The query is TPCH Q22, and most of the execution occurs in batch mode, except for the four unsupported iterators (inside the red boxes) that run in row mode.

5.2 Hash Join

Improvements in query execution were focused mainly on the batch-mode hash join operator. Its original implementation had limitations that meant it could not be used in some important scenarios.

Firstly, only inner joins were supported. The current version handles the full spectrum of join types: inner, outer, semi- and anti-semi joins.

Secondly, there had to be enough memory available for the query to build a hash table containing all the rows coming from the build side of the join. It the hash table did not fit entirely into memory, join processing switched on the fly from batch mode to row mode. Row-mode hash join is able to execute in low memory conditions by spilling some of the input data temporarily to disk. This solution, however, was not satisfactory because query performance dropped significantly when the switch happened. We enhanced batch-mode hash join by adding spilling functionality.

Thirdly, we improved our implementation of bitmap filters for batch-mode hash joins. Bitmap filters are used to reject at an early processing stage rows on the probe side of a join that do not have matching rows on the build side. Bitmap filters have been enhanced to support any number of join key columns of any type.

5.2.1 Spilling

When forced to spill to disk, it is important to minimize the amount of data that is written to disk. Let us first consider a single join that spills a fraction of its input data from the build side. Before building the hash table, this data is partitioned in memory into many buckets based on a hash function. Each time we decide to spill more data, another bucket is chosen and marked for spilling. A new temporary file is created and all rows that have already been assigned to this bucket are written to it and released from memory. All rows that are assigned to this bucket in the future will be appended to this file directly. After processing in this way all input from the build side, a hash table is created for the buckets that remain in memory.

When processing the probe side of a join, we do not partition incoming rows, but before doing a hash table lookup for a row we check whether its corresponding bucket on the build side has been spilled. If so, the row is written to another temporary file that stores probe side rows belonging to the bucket. After having processed all input rows on the probe side the hash table is released.

We are then left with pairs of files generated for the build and the probe side, one pair for each spilled bucket. For each pair the same join algorithm is executed but this time it is getting inputs from files instead of child operators. The new iteration of a join deals with less data. Spilling may happen again, of course, but fewer rows will be spilled.

We use a slightly modified version of TPCH Q10 to illustrate the effects of spilling. The query joins four tables (lineitem, orders, customer, nation) and then applies a group-by and a top operator. The query plan contains a sequence of batch-mode hash joins: lineitem is first joined with orders, then customer, and finally nation. Consequently, three hash tables (on orders, customer, and nation) are present in memory at the same time, competing for space.

In order to measure performance under spilling, we gradually reduced the memory available to the query, with the expectation that query performance will degrade slowly and not show any cliff-like behavior. The x-axes in Figure 5 represent the percentage of memory available to the query, 100% being the “desired” memory level that avoids all spilling. The left chart shows the impact of spilling on response time as the memory situation becomes tighter. The performance of the new implementation (labeled SQLNext) degrades gracefully. With just 6% of the desired memory, the

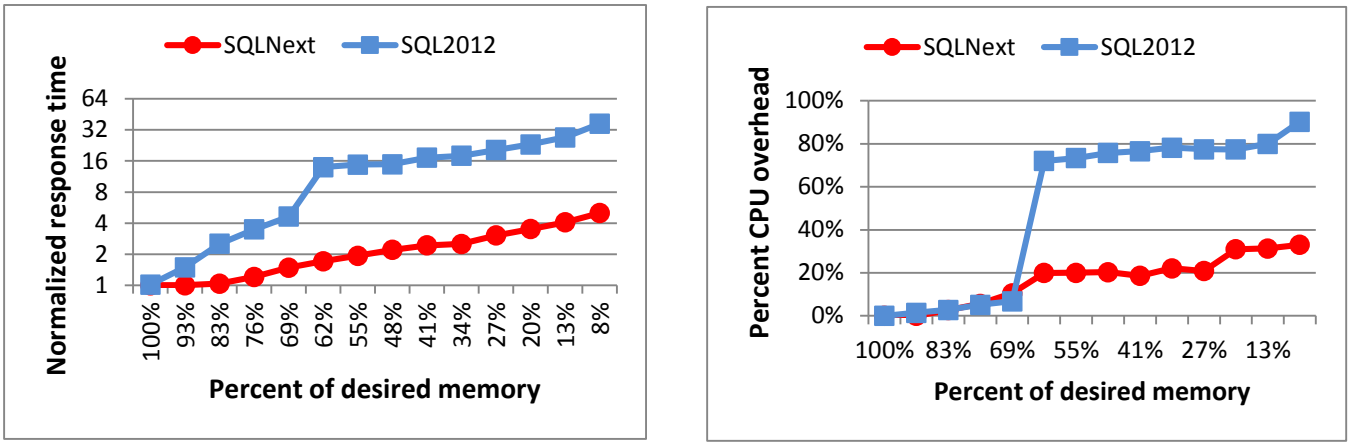

Figure 5: Performance comparison of old and new hash join implementation under memory pressure.

response time degrades by 5X, whereas the SQL Server 2012 implementation degrades by almost 37X. (Note that both the x and y axes are normalized. Actual response time and absolute memory requirements are both lower for SQLNext than for SQL2012).

Most of the cost of spilling involves IO, but there is also a CPU-time penalty. The chart on the right of Figure 5 shows that the CPU overhead grows with the amount of spilling involved and reaches 90% for SQL Server 2012 and a much lower 33% for SQLNext.

What is new in batch-mode hash join compared to its row-mode equivalent is the way a right-deep chain of joins is treated, e.g. a join between a fact table and several dimension tables. All joins share the same memory pool, which allows some flexibility in deciding how much each join should spill when approaching the limit for available memory.

We use the simple rule of always spilling from a hash table with more rows before spilling from a table with fewer rows. The number of rows can only be estimated, of course. The logic behind the rule can be explained by an example. Suppose we join a large fact table F containing 10M rows with two dimension tables, D1 containing 1,000 rows and D2 containing 10,000 rows. On average each D1 row will join with 10,000 F rows and each D2 row will join with 1,000 F rows. This means that if we spill one row from D1, we will on average have to spill 10,000 F rows but only 1,000 F rows for each row spilled from D2. It follows that spilling rows from the larger dimension table is expected to minimize the expected spill on the (much larger) probe side.

For each right-deep chain of joins, we sort the hash tables based on estimated size and start building from the smallest. However, we make exceptions from this rule when we need to satisfy bitmap dependencies. If a bitmap created by join A is to be used as a filter on the build side of join B, then we force hash table A to be built before hash table B.

5.3 Bitmap Filters

The next release of SQL Server also has many changes around how bitmap filters are generated and placed in the plan. In SQL Server 2012 bitmaps were generated only in cases where the optimizer was able to predetermine their exact final placement. This choice was working well because it did not force the optimizer to spend time moving bitmap filters around but at the same time it seriously limited the potential benefits of bitmap filters.

With the support for outer joins and semi-joins this approach began to show its limitations. Adding filters below each join and relying on exploration rules to push the filters into the right position quickly proved infeasible because it leads to an explosion of the logical plan space. To avoid this explosion, hash join operators now store information about bitmaps and their estimated selectivity. At implementation time, the selectivity information is passed to the child, enabling it to adjust its cost to account for the bitmap selectivity. In order to limit the amount of optimization requests to child groups the optimizer does not treat each request as different, but rather clusters requests based on their selectivity thus being able to reuse far more than it would otherwise. Once a plan is chosen, the optimizer performs an extra pass over the plan to push down all bitmap filters from their respective join operators to the lowest possible location.

There are two kinds of bitmaps that can be generated from a batch-mode hash join. The first is called a simple bitmap, which is an array of bits indexed by an integer column value relative to some base value. The second is a complex bitmap, which is a modified Bloom Filter optimized for better use of CPU caches. In SQL Server 2012 bitmap use was limited to single column keys, and to data types that can be represented as 64-bit integers. Currently complex bitmaps can be used for multiple columns and all data types. Only for the previously supported data types can they be pushed down as far as the scan in the storage engine layer. For the newly supported data types a filter iterator is inserted in the query execution plan. If a bitmap is generated for multiple integer key columns, we still try to split it into multiple single-column bitmaps and push them down to the storage engine.

Bloom filters may return false positives, and there is a trade-off between the false positives rate and the size of the filter. Compared to SQL Server 2012, we improved both the performance and false positives rate for complex bitmaps. Based on the actual cardinality of the input set and other statistics, computed during the repartitioning phase of a batch-mode hash join, we decide whether to pick a simple or complex bitmap. In case of a complex bitmap, we also decide how many bits per key value to use. For small cardinalities we use larger complex bitmaps to achieve a lower false positive rate.

One of the problems observed in SQL Server 2012 was that when a hash join runs out of memory and spills some data to disk, bitmap filters are not created and therefore in some cases a lot more data

on the probe side must be processed. When adding spill support to batch-mode hash join we allowed it to create a complex bitmap also when the data does not fit into memory. We reserve some memory for that purpose but if the bitmap grows too large, its creation may be abandoned.

6. ARCHIVAL COMPRESSION

Most data warehouses have some data that is frequently accessed and some that is accessed more infrequently. For example, the data may be partitioned by date and the most recent data is accessed much more frequently than older data. In such cases the older data can benefit from additional compression at the cost of slower query performance. To enable this scenario we added support for archival compression of SQL Server column stores.

The archival compression option is enabled on a per table or partition (object) basis; a column store can contain objects with archival compression applied and objects without. To allow for easy extensibility of existing on-disk structures, archival compression is implemented as an extra stream compression layer that transparently compresses the bytes being written to disk during the column store serialization process and transparently decompresses them during the deserialization process. Stream decompression is always applied when data is read from disk. Data is not cached in memory with stream compression.

We used the Xpress 8 compression library routine for compression. Xpress 8 is a Microsoft internal implementation of the popular LZ77 algorithm. For performance and scalability, it is designed to work in a multi-threaded environment and uses data streams up to 64KB in size.

Table 1 shows the compression ratios achieved with and without archival compression for several real data sets. The further reduction obtained by archival compression is substantial, ranging from 37% to 66% depending on the data. We also compared with GZIP compression of the raw data. Archival compression consistently achieved a better compression ratio, sometimes considerably better.

7. PERFORMANCE RESULTS

In this section we present measurements for query, update, and load performance, as well as compression rates, given the new capabilities recently added to SQL Server.

7.1 Batch Mode Performance

SQL Server 2012 introduced vectorized *batch-mode* query execution for data coming from non-clustered (secondary) column store indexes. For some operations, this can reduce CPU cycles per row by over 40X, and improve cycles per instruction as well. This is a critical component that goes hand-in-hand with columnar storage to improve query speed. It is even more important than columnar format if all data fits in memory.

To illustrate the kind of performance that can be achieved in batch mode, we ran some queries on the TPC-DS database at the 100GB scale factor. We compared the results from a copy of this database with clustered column store indexes on every table, to a copy of the same database with B-tree indexes on every table.

A 16 core machine with 48GB RAM and 4 hard drives was used for the tests. We focused on the table *store_sales* which contains approximately 288 million rows, and ran the following five queries.

Table 1: Comparison of compression ratios with and without archival compression and GZIP compression.

Database Name	Raw data size (GB)	Compression ratio		
		Archival compression?		GZIP
		No	Yes	
EDW	95.4	5.84	9.33	4.85
Sim	41.3	2.2	3.65	3.08
Telco	47.1	3.0	5.27	5.1
SQM	1.3	5.41	10.37	8.07
MS Sales	14.7	6.92	16.11	11.93
Hospitality	1.0	23.8	70.4	43.3

Q_count:

```
select count(*) from store_sales
```

Q_outer:

```
select  item.i_brand_id brand_id, item.i_brand brand,
        sum(ss_ext_sales_price) ext_price
from item left outer join store_sales
          on (store_sales.ss_item_sk = item.i_item_sk)
where item.i_manufact_id = 128
group by item.i_brand_id, item.i_brand
order by ext_price desc, brand_id
```

Q_union_all:

```
select d.d_date_sk, count (*)
from (select ss_sold_date_sk as date_sk,
             ss_quantity as quantity
      from store_sales
         union all
      select ws_sold_date_sk as date_sk,
             ws_quantity as quantity
      from web_sales) t, date_dim d
where t.date_sk = d.d_date_sk
and d.d_weekend = 'Y'
group by d.d_date_sk;
```

Q_count_in:

```
-- Here, store_study_group contains a
-- set of 100 IDs of interesting stores.
select count(*)
from store_sales
where ss_store_sk
  in (select s_store_sk from store_study_group);
```

Q_not_in:

```
-- bad_ticket_numbers contains a set of ticket numbers
-- with known data errors that we want to ignore.
select ss_store_sk, d_moy, sum(ss_sales_price)
from store_sales, date_dim
where ss_sold_date_sk = d_date_sk and d_year = 2002
and ss_ticket_number
       not in (select * from bad_ticket_numbers)
group by ss_store_sk, d_moy
```

Table 2: Comparison of execution times with and without column store indexes.

Query	Rowstore		Columnstore		Speedup	
	Cold	Warm	Cold	Warm	Cold	Warm
Q_count	13.0	4.33	0.309	0.109	**42.1**	**39.7**
Q_outer	263	1.03	4.1	0.493	**64.1**	**2.1**
Q_union_all	20.8	19.0	3.0	1.41	**6.9**	**13.5**
Q_count_in	62.5	24.0	2.29	1.15	**27.3**	**20.9**
Q_not_in	12.0	10.2	6.95	1.31	**1.7**	**7.8**

Table 2 contains a summary of the performance for these queries with the recent extensions to SQL Server. All times are in seconds.

Prior to the recent query processing enhancements of batch mode query execution, the warm start execution times were roughly equivalent for column store and row store, because the queries did not use batch mode. Column stores still benefited in the past for cold start from reduced I/O.

The large cold speedup (64.1X) for Q_outer is in large part due to selection of a plan with an index nested loop join when using the row store, which caused random I/O in a cold start situation. This illustrates the fact that column store performance is more consistent compared with index-based plans using B-trees because sequential scan of the column store is always used.

7.2 Storage Requirements

In clustered column store format, the store_sales table requires 13.2GB of space, or 46 bytes per row. This is versus 35.7GB in the uncompressed row store (clustered B-tree) format, plus an additional 7.7GB of non-clustered B-trees, or 43.5GB total and 151 bytes per row. This data set is not particularly compressible since it contains randomly generated data. Real data sets tend to be more compressible.

7.3 Delete Performance

Delete performance was measured on another table "Purchase" containing 101 million rows of movie ticket purchase information. "Purchase" is a data warehouse fact table with 19 columns. About 5.5% of the rows (about 5.5M rows) were deleted from the Purchase table at random throughout the table using this statement:

```
delete from Purchase where MediaId % 20 = 1;
```

In our experience with customers, they sometimes delete or modify around 5% of the rows in a table to correct errors or adjust data with late-arriving values, so this level of deletions is interesting from a practical perspective.

On a 4-core machine, with the data on one disk, the delete statement finished in 57 seconds. On the same machine, when the Purchase table was stored as a clustered B-tree, the same delete statement took 239 seconds. The reason the deletes are faster with the column store is that we simply insert a set of <row_group_id, row_number> pairs in a B-tree (the delete bitmap) to mask the rows. For the clustered B-tree, the rows are actually removed from pages. This generates more log data and requires more storage reorganization, so it takes longer. If enough data is deleted (say >>10%) then a manual column store index rebuild is recommended to reclaim space.

7.4 Bulk and Trickle Load Rates

Bulk load rates for clustered column store have been measured at about 600GB/hour on a 16 core machine, using 16 concurrent bulk load jobs (one per core) targeting the same table

We did a trickle load test on a single thread whereby we inserted 3.93 million rows, one at a time in independent transactions, into an empty column store index. This was done on a machine with 4 cores and 8 hardware threads. The test was implemented with a cursor reading from a source table and inserting one row at a time into the target. The data was drawn from the Purchase table mentioned earlier. The test took 22 minutes and 16 seconds. The insertion rate was 2,944 rows/second.

The tuple mover had completed compressing three row groups to column store format at the end of the test, and the remainder was in one open delta store, so these figures include compression time. The table could be queried with interactive response time during the insertions.

Much higher load rates can be obtained by batching groups of rows together (say 1,000 at a time) in small bulk loads and by using concurrent streams to add data, rather than a single thread as was done in this test.

As a basic demonstration of this, we loaded 20 million rows in batches of 1,000 rows using the SQL Server bcp program, our external bulk loader. The test was single-threaded. The target was an empty version of the Purchase table, with a clustered column store index. The 20 million rows were loaded in 9 minutes 46 seconds, which is a rate of 34,129 rows per second. This is 11.5 times faster than when inserting a row at a time. During this trickle bulk load test, the data could be queried with interactive response time for full scans. Immediately after the bcp job completed there were 20 total row groups, with 18 compressed row groups, one closed delta store, and one open. In just over a minute, the tuple mover had moved the closed row group to compressed format, leaving 19 compressed row groups and one open row group.

8. RELATED WORK AND SYSTEMS

The idea of decomposing records into smaller subrecords and storing them in separate files goes back to the seventies. Hoffer and Severance [6] published a paper on the optimal decomposition into subrecords in 1975. A 1979 paper by Batory [1] considered how to compute queries against such files. A 1985 paper by Copeland and Khoshafian [3] discussed fully decomposed storage where each column is stored in a separate file, that is, full columnar storage.

Many prototype and commercial systems relying on columnar storage have been developed. MonetDB was one of the early pioneers; its development began in the early nineties at CWI [4]. Sybase launched Sybase IQ, the first commercial columnar database system, in 1996. A 2005 paper by Stonebraker et al [12] on C-Store rekindled interest in column stores.

Several commercial systems using column-wise storage are available today. Most are pure column stores but some are hybrid systems that support both column-wise and row-wise storage.

The earliest pure column stores are Sybase IQ [21] and MonetDB [18], which have been available for well over a decade. Newer players include Vertica [23], Exasol [14], Paraccel [19], InfoBright [16] and SAND [20]. In addition to SQL Server, three other systems support both row-wise and column-wise storage: Actian VectorWise [17], Greenplum [15], and Teradata [22].

VectorWise [17] originated in the MonetDB/X100 project [2] and is now embedded in the Ingres DBMS [8]. VectorWise began as a pure column store but now also supports hybrid storage. Updates are handled by a technique called Positional Delta Trees (PDT). Conceptually, a PDT is an in-memory structure that stores the position and the change (delta) at that position. Scans merge the changes in PDTs with data stored on disk. PDTs only use a configurable amount of memory. Once the memory pool is exhausted the PDT changes must be written to persistent storage. This can be an expensive operation since it effectively rewrites the entire table.

Greenplum [15] began as a row store but added column store capabilities. Their Polymorphic Storage feature allows different partitions of the same table to be stored in different form, some row-wise and some column-wise. We have not been able to find information on how deeply column-wise processing has been integrated into the engine and whether data stored column-wise can be updated.

Teradata introduced columnar storage in Teradata 14 [22]. In their approach, a row can be divided into sub-rows, each containing a subset of the columns. Sub-rows can then be stored column-wise or row-wise. Whether Teradata 14 uses any form of vectorized or batch processing is not clear. Deletes and updates may be expensive because Teradata appears not to use any form of delta store so all affected columns have to be accessed and updated.

9. REFERENCES

[1] Batory, D. S.: On searching transposed files. ACM Trans. Database Syst. 4, 4 (1979), 531-544.

[2] P. A. Boncz, M. Zukowski, and N.Nes, MonetDB/X100: Hyper-pipelining query execution. CIDR, 2005, 225-237.

[3] G. P. Copeland and S. Khoshafian, A decomposition storage model. SIGMOD, 1985, 268 -279.

[4] Harizopoulos, S., Liang, V., Abadi, D.J., and Madden, S.: Performance tradeoffs in read-optimized databases. VLDB, 2006, 487-498.

[5] Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, Peter A. Boncz: Positional update handling in column stores. SIGMOD, 2010: 543-554.

[6] J. A. Hoffer and D. G. Severance, The use of cluster analysis in physical data base design, VLDB, 1975, 69-86.

[7] M. Holsheimer and M. L. Kersten, Architectural support for data mining, KDD, 1994, 217-228.

[8] D. Inkster, M. Zukowski, and P. A. Boncz, Integration of VectorWise with Ingres, SIGMOD Record, 40(3):45-53, 2011.

[9] 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, SIGMOD, 2011, 1177-1184.

[10] Microsoft, Column store Indexes in Books Online for SQL Server 2012, available at http://msdn.microsoft.com/en-us/library/gg492088.aspx.

[11] Microsoft, SQL Server Column store Index FAQ, http://social.technet.microsoft.com/wiki/contents/articles/3540.sql-server-column store-index-faq-en-us.aspx.

[12] M. Stonebraker et al. C-Store: A Column-oriented DBMS. VLDB, 2005, 553-564.

[13] TPC Benchmark DS (Decision Support), Draft Specification, Version 32, http://tpc.org/tpcds.

[14] ExaSolution, http://www.exasol.com

[15] Greenplum Database, http://www.greenplum.com

[16] InfoBright, http://www.infobright.com

[17] Actian VectorWise, http://www.actian.com/products/vectorwise.

[18] MonetDB, http://monetdb.cwi.nl

[19] ParAccel Analytic Database, http://paraccel.com

[20] SAND CDBMS, http://www.sand.com

[21] Sybase IQ Columnar database, http://www.sybase.com/products/datawarehousing/sybaseiq

[22] Teradata Columnar, http://www.teradata.com/products-and-services/database/teradata-14

[23] Vertica, http://www.vertica.com