Real-Time Analytical Processing with SQL Server

Per-Åke Larson, Adrian Birka, Eric N. Hanson,
Weiyun Huang, Michal Nowakiewicz, Vassilis Papadimos
Microsoft
{palarson, adbirka, ehans, weiyh, michalno, vasilp}@microsoft.com

ABSTRACT

Over the last two releases SQL Server has integrated two specialized engines into the core system: the Apollo column store engine for analytical workloads and the Hekaton in-memory engine for high-performance OLTP workloads. There is an increasing demand for real-time analytics, that is, for running analytical queries and reporting on the same system as transaction processing so as to have access to the freshest data. SQL Server 2016 will include enhancements to column store indexes and in-memory tables that significantly improve performance on such hybrid workloads. This paper describes four such enhancements: column store indexes on in-memory tables, making secondary column store indexes on disk-based tables updatable, allowing B-tree indexes on primary column store indexes, and further speeding up the column store scan operator.

Categories and Subject Descriptors

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

Keywords

In-memory OLTP, column store, OLAP, operational analytics, real-time analytics, hybrid transactional and analytical processing.

1. INTRODUCTION

Transactional processing (OLTP) and analytical processing are traditionally separated and running on different systems. Separation reduces the load on transactional systems which makes it easier to ensure consistent throughput and response times for business critical applications. However, users are increasingly demanding access to ever fresher data also for analytical purposes. The freshest data resides on transactional systems so the most up-to-date results are obtained by running analytical queries directly against the transactional database. This means that the database system must be able to efficiently handle transactional and analytical processing concurrently. SQL Server 2016 will include several enhancements that are targeted primarily for such hybrid workloads.

Over the last two releases SQL Server has added column store indexes (CSI) and batch mode (vectorized) processing to speed up analytical queries and the Hekaton in-memory engine to speed up OLTP transactions. These features have been very successful; customers have reported orders-of-magnitude improvements. However, each feature is optimized for specific workload patterns. Columnstore indexes are optimized for large scans but operations such as point lookups or small range scans also require a complete scan, which is clearly prohibitively expensive. Vice versa, lookups are very fast in in-memory tables but complete table scans are expensive because of the large numbers of cache and TLB misses and the high instruction and cycle count associated with row-at-a-time processing.

This paper describes four enhancements in the SQL Server 2016 release that are designed to improve performance on analytical queries in general and on hybrid workloads, in particular.

1. **Columnstore indexes on in-memory tables**. Users will be able to create columnstore indexes on in-memory tables in the same way as they can now for disk-based tables. The goal is to greatly speed up queries that require complete table scans.
2. **Updatable secondary columnstore indexes**. Secondary CSIs on disk-based tables were introduced in SQL Server 2012. However, adding a CSI makes the table read-only. This limitation will be remedied in SQL Server 2016.
3. **B-tree indexes on primary columnstore indexes.** A CSI can serve as the base storage for a table. This storage organization is well suited for data warehousing applications because it is space efficient and scans are fast. However, point lookups and small range queries are very slow because they also require a complete scan. To speed up such operations users will be able to create normal B-tree indexes on primary column stores.
4. **Column store scan improvements.** The new scan operator makes use of SIMD instructions and the handling of filters and aggregates has been extended and improved. Faster scans speed up many analytical queries considerably.

In this paper we use the terms **primary index** or **base index** for the index that serves as the base storage for the table and the term **secondary index** for all other indexes on a table. The corresponding terms traditionally used in the SQL Server context are **clustered index** and **non-clustered index** but they no longer seem to convey quite the right meaning.

The rest of the paper is organized as follows. Section 2 gives an overview of the design and status of column store indexes and of the Hekaton engine as implemented in SQL Server 2014. Sections 3 to 6 describe the four enhancements mentioned above including some initial performance results. Note that the experiments were run on early builds of the system that had not yet been fully optimized. Section 7 contains a brief summary of related work.

2. BACKGROUND ON SQL SERVER

As illustrated in Figure 1, SQL Server 2014 integrates three different engines.

1. The classical SQL Server engine primarily used for processing disk-based tables in row format. It can also process data from the two other stores albeit slower than the specialized engines.
2. The Apollo engine processes data in columnar format and is designed to speed up analytical queries.
3. The Hekaton engine processes data in in-memory tables and is designed to speed up OLTP workloads.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

Users and applications interact with the system through a common frontend in the same way they always have. Queries and transactions can involve all three engines but this is largely transparent to users. All engines run in the same multi-threaded service process. The backend of the system is also common. For example, all engines use the same log and are integrated with SQL Server's high availability solution (AlwaysOn). The integration greatly simplifies use and management of the system.

The following two sections give a brief overview of the features of Apollo (column store indexes and batch processing) and Hekaton (in-memory tables).

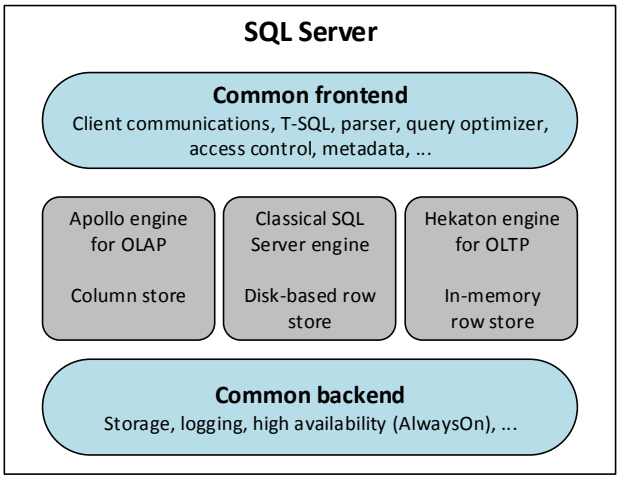

Figure 1: SQL Server now contains three engines but all share the same frontend and backend

2.1 Columnstore Indexes

A columnstore index (CSI) stores data column-wise instead of row-wise as is done in a B-tree index or heap. SQL Server was the first of the major commercial systems to integrate column-wise storage into the system. A CSI can be used as a base (primary) index or as a secondary index.

Column-wise storage has two key benefits: it saves space because columnar data can be better compressed and greatly improved performance on analytical queries that scan large numbers of rows but few columns. This section gives a brief overview of CSIs and batch mode processing in SQL Server 2014. Further details can be found in [7] [9] [13].

2.1.1 Index Storage

A CSI stores the data from a set of rows as any index does. The set of rows is divided into row groups of about one million rows each. Each row group is encoded and compressed independently, producing one compressed column segment for each column in the index. If a column uses dictionary encoding, such as a string column, the conversion also produces dictionaries, one global dictionary that contains the most common values and a local dictionary for each segment that contains values not found in the global dictionary.

Column segments and dictionaries are stored as SQL Server blobs (LOB). A directory keeps track of the location of segments and dictionaries so all segments comprising a column and any associated dictionaries can be easily found. The directory contains additional metadata about each segment such as number of rows, size, how data is encoded, and min and max values.

Frequently used column segments and dictionaries are cached in an in-memory cache (not in the buffer pool) which greatly reduces I/O. When a column has to be read in from disk it is done efficiently with very deep read-ahead.

2.1.2 Update processing

In SQL Server 2014 primary CSIs are updatable while secondary CSIs are not. Removing this limitation is one of the enhancements covered in this paper.

Two additional components are needed to make primary CSIs updatable: **delete bitmaps** and **delta stores**. The **delete bitmap** of an index indicates which rows in the index have been logically deleted and should be ignored during scans. A delete 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 RID of a deleted row. A RID consists of row group number and the row's position within the group.

New and updated rows are inserted into a **delta store** which is a traditional B-tree row store. A delta store contains the same columns as the corresponding column store. The B-tree key is a unique integer row ID generated by the system. Large bulk insert operations do not insert rows into delta stores but convert batches of rows directly into columnar format.

A CSI can have zero, one, or more delta stores. New delta stores are created automatically as needed to accept inserted rows. A delta store is closed when it reaches 1M rows. SQL Server automatically checks in the background for closed delta stores and converts them to columnar storage format. All delta stores are transparently included in any scan of the column store index.

2.1.3 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 900 rows where each column is stored as a contiguous vector of fixed-sized elements. A "selected rows" vector indicates which rows are still logically part of the batch. Row batches can be processed very efficiently.

In SQL Server 2014 most query operators are supported in batch mode: scan, filter, project, hash join (inner, outer, semi- and anti-semi joins), hash aggregation, and union. 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 on the compressed data, which can be significantly cheaper and reduces the output from the scan.

2.2 In-Memory OLTP (Hekaton)

The Hekaton engine in SQL Server 2014 is intended to greatly speed up OLTP workloads. Tables managed by the Hekaton engine are stored entirely in main memory. The engine is designed for high levels of concurrency. All its internal data structures are latch-free (lock-free) to avoid interference among concurrent threads – there are no semaphores, mutexes or spinlocks. It uses an optimistic, multi-version concurrency control (MVCC) scheme to guard against interference among concurrent transactions [8]. Durability is guaranteed by logging and checkpointing to durable storage [1][2].

Hekaton tables are queried and updated using T-SQL in the same way as other SQL Server tables. A T-SQL stored procedure that

references only Hekaton tables can be compiled into native machine code for further performance gains [3].

2.2.1 Tables and indexes

Hekaton tables and indexes are optimized for memory-resident data. Rows are referenced directly by physical pointers, not indirectly by a logical pointer such as a row ID or primary key. Row pointers are stable; a record is never moved after it has been created.

A table can have multiple indexes, any combination of hash indexes and range indexes. A hash index is simply an array where each entry is the head of a linked list through rows. Range indexes are implemented as Bw-trees which is novel latch-free version of B-trees optimized for main-memory [10].

2.2.2 Multi-versioning

Hekaton uses multi-versioning where an update creates a completely new version of a row. The lifetime of a version is defined by two timestamps, a begin timestamp and an end timestamp and different versions of the same row have non-overlapping lifetimes. A transaction specifies a logical read time for all its reads and only versions whose lifetime overlaps the read time are visible to the transaction.

Multi-versioning improves scalability because readers no longer block writers. (Writers may still conflict with other writers though.) Read-only transactions have little effect on update activity; they simply read older versions of records as needed. Multi-versioning also speeds up query processing be reducing copying of records. Since a version is never modified it is safe to pass around a pointer to it instead of making a copy.

3. CSI ON IN-MEMORY TABLES

The Hekaton engine stores tables in memory and is very efficient on OLTP workloads. SQL Server 2016 allows users to create column store indexes also on in-memory tables. This enables efficient and concurrent real-time analysis and reporting on operational data without unduly hurting the performance of transaction processing.

3.1 Architecture

A user creates a Hekaton table with a secondary CSI using the following syntax.

```
CREATE TABLE <table_name> (
    ...
   INDEX <index_name> CLUSTERED COLUMNSTORE
    ...
) WITH (MEMORY_OPTIMIZED = ON)
```

The table is defined with the MEMORY_OPTIMIZED option set to ON so it is stored in memory and managed by Hekaton. A secondary CSI covering all columns is also created. Figure 2 shows the components constructed when the table is created.

All rows are stored in the in-memory Hekaton table (shown on the left) which may have one or more hash or range indexes. Most of the rows (shown in blue) are also stored in the CSI in columnar format. The CSI is shown as containing two row groups, each with five compressed column segments. Data is duplicated between the Hekaton table and the CSI but in practice the space overhead is small because of compression. The space overhead is data dependent but typically in the range 10% to 20%.

The upper portion (in yellow) of the Hekaton table in Figure 2 represents rows that are not yet included in the CSI. We call this portion the tail of the table. New rows and new versions of rows are inserted into the Hekaton table only, thus growing the tail. A background thread is responsible for copying rows in the tail portion into the column store (see Section 3.3 for details).

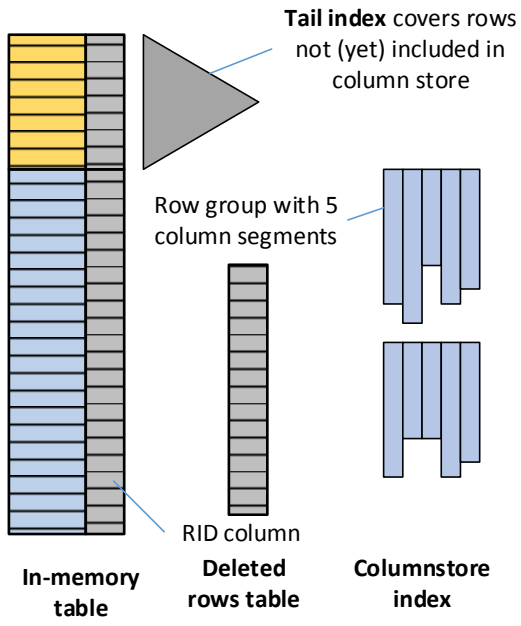

Figure 2: Components of design enabling column store indexes on Hekaton tables

The Hekaton table contains a hidden row ID (RID) column that indicates the location of the row in the column store. A RID consists of a row group ID that identifies the compressed row group and the position within the row group. The RID column allows a row in the column store to be efficiently located in CSI which is crucial for fast online processing of updates and deletes.

For rows in the tail of the Hekaton table, the value in the RID column is a special invalid value which makes it easy to identify rows still in the tail. These rows are included in a hidden Tail Index which is shown as a triangle to the right of the table in Figure 2. All rows not yet included in the CSI can be easily found by scanning the Tail Index. This is needed for table scans that use the CSI and also when migrating rows to the CSI.

The Deleted Rows Table shown in Figure 2 is a hidden Hekaton table that contains RIDs of rows in the CSI that have been deleted from the user table. This table is checked during scans to identify rows that have been logically deleted and thus should be skipped.

The compression and decompression algorithms are the same for this CSI as for regular SQL Server CSI but the storage for compressed data is different. The compressed row groups and all metadata about them are stored in internal Hekaton tables, which allows Hekaton versioning to work correctly for CSI scans.

Each row group is stored in a separate file on stable storage; this permits the system to evict a compressed segment from memory and reload it as needed, so that only segments that are in use will consume memory resources.

3.2 User Operations

Hekaton normally manages the most performance-critical tables of a database so it is crucial that adding a CSI not significantly reduce OLTP performance. Ideally, the overhead during normal operations should be no higher than that of a Bw-tree index.

Inserting a new row or row version into the user table consists of inserting the row into the in-memory Hekaton table and adding it

to all indexes, including the Tail Index. The data will eventually be copied to the CSI by a background task.

Deleting a row from the table consists of first locating the target row in the Hekaton table and deleting it from there. If the RID column contains a valid value, a row with that RID is inserted into the Deleted Rows Table, which logically deletes the row from the CSI.

To update a row it is first updated in the Hekaton table in the normal way, that is, a new version is created and added to the table. The new version is included in the Tail Index but not immediately added to the CSI. If the old version is included in the CSI (its RID column contains a valid value) a row is inserted into the Deleted Rows Table to logically delete the old version from the CSI.

Point lookups or range scans using one of the table's indexes are entirely unaffected by the presence of the CSI. Analytic queries scan the compressed row groups in parallel, in the process skipping rows whose RIDs occur in the Delete Rows Table. They also scan the tail of the Hekaton table using the Tail Index and append those rows to the result. This implementation ensures correct snapshot semantics for table scans because the Deleted Rows Table and the Hekaton table both contain versioned rows. All usual optimizations that CSI allows in SQL Server applies to scans in this scenario, such as pushing down predicates to the scan operator, elimination of segments that cannot match a predicate based on max/min value stored in the segment, etc.

3.3 Data Migration

A background task, known as the **data migration task**, periodically copies data from the tail of the Hekaton table into the compressed column store. This operation is transparent to user transaction and does not affect user's snapshot-based view of the table. It also needs to avoid disrupting user workloads too much (see Section 3.4). Data migration proceeds in two stages.

Stage 1 (everything occurs in a single Hekaton transaction): The tail of the Hekaton table is scanned, using the Tail Index, and some, but not necessarily all, rows are selected to form a new compressed row groups. The intent is to exclude rows that are likely to be frequently updated because they will end up polluting the CSI with rows that are logically deleted but still physically present. Such rows both waste space and slow down scans.

SQL Server uses a time-based policy to determine which rows to migrate – rows that have been recently changed are presumed to be likely to change again and are therefore not migrated. Temporal statistics are kept about rows, and data migration is not even started until there is a sufficient number of "cool" rows (at least a million). A row is considered "cool" if it has not been modified within some specified period of time (at least an hour). Once started, data migration will only migrate "cool" rows.

The row group created is then compressed and written into the internal tables that implement CSI storage for Hekaton. The compression process assigns RIDs to each row it compresses. These RIDs are inserted into the Deleted Rows Table and the Stage 1 transaction commits.

After Stage 1 commits, the algorithm has physically migrated rows to the CSI but logically the rows are still invisible in the column store because the rows are hidden by the entries in the Deleted Rows Table. However, the rows are still in the tail of the Hekaton table since the RID column for them is still invalid. Therefore, scans see no change in the system.

To avoid millions of inserts in this stage, the Deleted Rows Table actually stores rows representing ranges of RIDs. In Stage 1, about one thousand Deleted Rows Table inserts are sufficient to hide all rows in a compressed row group with one million rows.

Stage 2: This stage uses a sequence of short transactions. Each transaction deletes a row from the Deleted Rows Table that was inserted in Stage 2 and updates the RID column of the Hekaton table for the rows in the range covered by that Deleted Rows Table row. This makes the corresponding rows in the column store visible to future transactions, but it also means future transactions will not see the rows as part of the tail of the Hekaton table. This guarantees that scans do not see duplicates between the column store and the tail of the Hekaton table, and that all scans observe correct snapshot semantics.

3.4 Minimizing data migration overhead

Stage 2 of the data migration algorithm will update a row in the Hekaton table for every row in the new compressed row group. Normally this would generate a large log overhead, strongly impacting user workloads. SQL Server addresses this problem by not logging changes to the RID column. Instead the values in this column are recovered at database recovery time by scanning the compressed row group, extracting the primary key for each row from it, looking up the corresponding row in the Hekaton table, and updating the value of the RID column appropriately.

Another potential issue is that Stage 2 transactions may cause write-write conflicts with user transactions (i.e., both transactions trying to update the same row in the Hekaton table, which in a snapshot-based system results in one of the transactions aborting). It is important both to minimize the chances of this and to avoid aborting user transactions when this does happen, because user transactions would need to be retried when this occurs, hurting performance.

The reason Stage 2 is done in batches of small transactions is precisely to minimize the chances of write-write conflict with user transaction. In addition, SQL Server ensures that a background Stage 2 transaction never causes a user transaction to abort with a write-write conflict. When such a conflict is detected, if the background transaction has not yet committed, it is always chosen as the victim in the write-write conflict. If it has committed, SQL Server uses special handling logic which causes the user transaction to update the row version created by the background transaction, instead of the row version the user transaction wanted to update originally. This is possible, since the background transaction does not change any user-visible data, so having a user transaction override its changes is acceptable.

3.5 Row Group Cleanup

When a row group has many deleted rows (i.e., many entries in the Deleted Rows Table), scan performance is reduced and memory is wasted on storing the compressed data for the deleted rows as well as Deleted Rows Table entries for them. To address this issue, SQL Server implements an automatic cleanup algorithm. When the number of deleted rows in a row group exceeds a certain threshold (currently 90%), a background task starts that essentially runs the data migration algorithm of Section 3.3 in reverse, causing all valid rows in the row group to be included in the tail. A subsequent data migration will combine them with other rows to create a pristine row group with no deleted rows. The optimization of Section 3.4 are applied to this row group cleanup as well, in order to reduce its impact on user transactions.

3.6 Performance

Recall that the main goal of this improvement is to greatly reduce the run time of analytical queries without maintenance of CSIs significantly slowing down the transactional workload. In this section we report results from micro-benchmarks measuring query speedup

and maintenance overhead. We are primarily interested in the effect on the elapsed time of transactions, i.e., the upfront cost. There will be additional background work for copying data to the CSI but this is expected to have only minimal effect on transaction throughput.

For the experiments we used the lineitem table from the TPC-H database with 60M rows. We created two versions of the table, LI-csi and LI_nocsi. LI_csi had a range index on (l_orderkey, l_linenumber) and a column store index covering all columns. LI_nocsi had two range indexes, one on (l_orderkey, l_linenumber) and one on l_shipdate. The experiments ran on a machine with 16 cores (Intel Xeon E5-2660 CPUs, 2.20 GHz with 8 cores in 2 sockets).

3.6.1 Query speedup

Our first experiment measured the speedup of a simple aggregation query against a newly created instance of the table. The CSI contained 107 row groups, about 800K rows were left in the Tail Index and the Deleted Rows table was empty. We ran the query

```
Select l_discount, sum(l_quantity*l_extendedprice*l_discount)
from lineitem where l_partkey < 1000000
group by l_discount
```

We ran the query against the row store in interop and in a native stored procedure and against the CSI. (In interop the query is executed by the classical SQL Server engine making use of a scan operator for Hekaton tables.) There is no index on l_partkey so a complete scan of the table is necessary. Scans of in-memory row stores run single-threaded while scans against CSIs are parallelized. The measured elapsed times are shown in Table 1.

Table 1: Elapsed time (sec) of test query

Scan type	Elapsed time (s)	Speedup
Row store scan, interop	44.441	
Row store scan, native	28.445	1.6X
CSI scan, interop	0.802	55.4X

Row store scans are rather slow because the rows are scattered in memory resulting in random memory accesses and high cache miss rate. Compiling the query to native code helps some but it can't cure the cache-miss problem.

Scanning the CSI instead gives a 55X performance boost. This is caused by several factors: sequential memory access pattern, early data reduction by evaluating the filter predicate in the scan operator, and segment elimination. The operator needed to scan only 45 out of 107 segments, the rest were eliminated because metadata indicated that they contain no rows satisfying the filter predicate.

3.6.2 Effects of inserts, updates and deletes

All indexes increase the cost of inserts, updates, and deletes and column store indexes are no exception. Furthermore, changes to the data also affect subsequent column store scans. Inserts and updates add rows to the tail index which increases scan costs, at least temporarily until the data migration task runs. Updates and deletes add rows to the Deleted Rows table which also increases the scan costs.

We measured the effects by a few micro-benchmarks on the two versions of the lineitem table mentioned above. The results of are shown in Table 2.

The second line in the table shows the elapsed time of inserting 400,000 rows. Having a CSI on the table increased the insert time by 11.9% because the Tail Index must be updated. We then ran our test query again. The elapsed time increased 8.4% to 0.869 sec (from 0.802 sec) because there were more rows in the Tail Index.

Table 2: Cost of inserts, updates, deletes and their effect on query time

Operation	Elapsed time (sec)		Increase (%)	
	With CSI	No CSI	Update	Query
CSI scan, interop	0.802			Base
Insert 400,000 rows	53.5	47.8	11.9%	
CSI scan, interop	0.869			8.4%
Update 400,000 rows	42.4	28.9	46.7%	
CSI scan, interop	1.181			47.3%
Delete 400,000 rows	38.3	30.5	25.6%	
CSI scan, interop	1.231			53.5%

Next we updated 400,000 random rows. The cost of maintaining the CSI increased the elapsed time by 46.7% (from 28.9 to 42.4 sec). The additional cost stems from inserting rows in the Deleted Rows table to logically delete the old versions and adding the new versions to the Tail Index. We then ran the test query again. Its elapsed time had now increased by 47.3% (to 1.181 sec from 0.802 sec). This is caused by two factors: the Tail Index now contains 400,000 more rows as does the Deleted Rows table.

In the third step we deleted 400,000 randomly selected rows. The CSI increased the cost of deletions by 25.6% (from 30.5 sec to 38.3 sec). The increase is due to insertions into the Deleted Rows table. The additional 400,000 rows in the Deleted Rows table slightly increased the time for the test query from 1.181 sec to 1.231 sec.

4. UPDATING SECONDARY COLUMN STORES

Few OLTP applications need to store all their data in memory and can instead use traditional disk-based tables for less performance critical tables. Users obviously want real-time analytical queries to run fast regardless of where the tables are stored. This can be achieved by creating secondary CSIs on tables used for analytical queries. Support for CSIs on disk-based tables has been available since SQL Server 2012 but adding a CSI made the table read only. In SQL Server 2016, CSIs on disk-based tables become updatable. This section describes how updates are handled.

4.1 Components Added to Support Updates

Figure 3 illustrates the main components of the implementation and how they are used in a column store scan. The colors represent how frequently rows are updated: cold (blue) rows are updated infrequently and hot (red) rows are frequently updated.

All rows are stored in the base index which can be a heap or a B-tree and also included in the CSI, either in a compressed row group or a delta store. The coldest rows are included in compressed row that have exactly the same structure as for primary CSIs. Primary and secondary CSIs have identically structured delete bitmaps but secondary CSIs use the delete bitmap slightly differently.

Hot, recently inserted or modified rows are stored in delta stores. A delta store for a secondary CSI is organized as a B-tree with the locator of the base index as its key. The locator is a row ID if the base index is a heap and the key columns of the base index if it is a B-tree.

When a row is deleted from the base index we must delete it from the CSI as well. We know the row's locator so we can easily check whether it is still in a delta store. If so, we simply delete it from the delta store. Otherwise, we know that the row is in one of the compressed row groups. To logically delete it we need to insert an entry into the delete bitmap indicating which row group it belongs to and its position within the row group. Determining the row's location would require a complete scan of the compressed row groups, which would make deletes unbearably slow.

This is where the **delete buffer** comes in. Deletes from compressed row groups are not immediately added to the delete bitmap but saved in the delete buffer and later applied in bulk by a background task. The delete buffer is a B-tree containing unique keys of recently deleted rows. The delete buffer batches recent deletes in the same way that delta stores batch recent inserts so that they can be applied more efficiently in bulk later on.

However, as illustrated in Figure 3, a scan now has to check both the delete bitmap and the delete buffer to eliminate logically deleted rows. Conceptually, the scan computes an anti-semijoin between the compressed row groups and the delete buffer.

Note that a secondary CSI need only include columns relevant to analytical queries, thereby saving space and speeding up compression.

4.2 Handling Deletes

Most differences between the designs for primary and secondary CSIs can be attributed to the handling of deletes. Inserts are treated the same in both implementations and updates are split into deletes followed by inserts. Primary CSIs are targeted for data warehousing applications where deletes are rare and scan performance crucial so we are willing to accept a higher delete cost if we can retain high scan performance. Secondary CSIs are indexes on OLTP tables with frequent updates and deletes and some scan activity. Here the tradeoff is the opposite: reduce the cost of deletes (and thereby also of updates) in exchange for a small reduction in scan performance.

Deletes are processed by first finding the locator of each matching row in the base index. The locator is the identifier of the row in the underlying base index, which is either a heap or a B-tree. After the system locates a row in the base index, it proceeds to locate and delete the row in secondary indexes using their respective keys (and also to process referential constraints and maintain indexed views).

In case of a secondary CSI, if the index is filtered and the row being deleted does not satisfy the filter condition, no further action is needed. Otherwise, we iterate through the delta stores, seeking for the row using its key. If we find the row in a delta store (the most common case, assuming that most deletes and updates target hot rows), we can immediately delete it.

If the row is not found in any of the delta stores, we know that it must occur in one of the compressed row groups but we don't know where. Locating the row so we can add it to the delete bitmap would require a complete scan. However, scanning all compressed row groups to locate the row is not practical in a system for transactional processing. To overcome this problem we chose to temporarily buffer the keys of deleted rows in a delete buffer and periodically apply them using a single scan.

Since compressed row groups can collectively contain multiple old versions of a row with the same key, each delete buffer entry is also tagged with a generation. When the system begins compressing a delta store, it increments the CSI's generation number and assigns it to the delta store (which is now immutable) and all the resulting compressed row groups. Delete buffer entries are tagged with the highest generation number assigned at the time of the delete. A delete buffer entry of the form <K, GD> marks as deleted all rows with key K appearing in any immutable row group with a generation number less than or equal to GD.

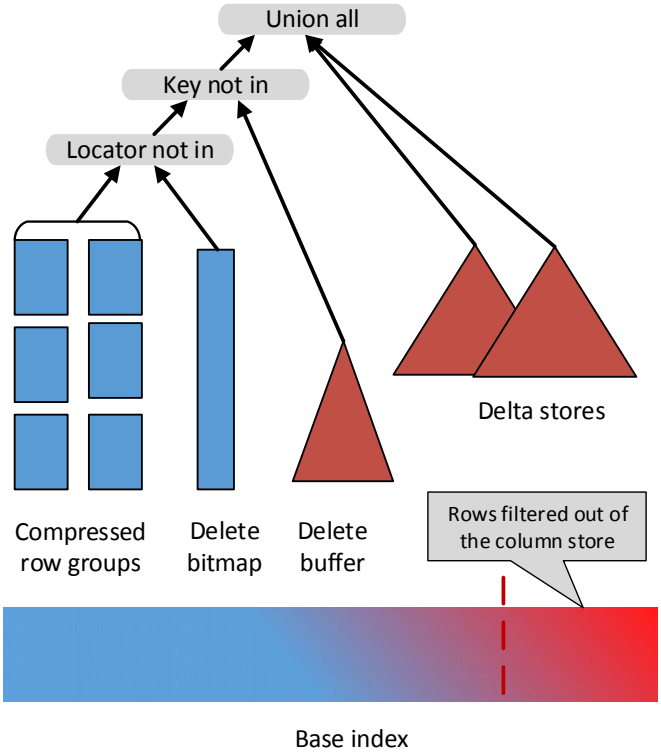

Figure 3: Main components of an updatable secondary column store index on a disk-based table

4.3 Scan Processing

Scans of secondary CSIs are necessarily less efficient than scans of primary CSIs since they need to consult not just the delete bitmap but also the delete buffer. In the worst case, the delete buffer overhead includes fetching the segments of the key columns (even if the scan does not otherwise involve these columns) and then performing a batch mode hash anti-semi-join with the delete buffer contents. A logical scan of a secondary CSI is converted during runtime plan generation into a batch mode anti-semi-join. The build side of this anti-semi-join is the delete buffer and the probe side is the CSI contents.

4.3.1 Using Bloom filters

We short-circuit the anti-semi-join, pre-qualifying data using a Bloom (bitmap) filter. The bitmap filter is constructed from the delete buffer contents during the hash build. If the bitmap filter test fails (i.e. the key is definitely not in the delete buffer) then the row is output into the scan. If the test succeeds (the row might be in the delete buffer) then the build-side hash table is examined. Selection vectors are merged to combine rows qualified by the bitmap filter and rows qualified by a full hash lookup. The hash values of the keys from the column store to use to probe the bitmap filter and hash values are calculated efficiently by using a vectorized approach. The bitmap filter and vectorized hashing capability is common to all batch mode joins.

4.3.2 Exploit segment metadata

We store min-max metadata for each column segment and compute the min-max range for the values in the corresponding delete buffer column during hash build. If the two ranges do not overlap, there is

no need to check against the delete buffer for the current row group. A fast code path in the anti-semi-join then bypasses bitmap filter and hash table lookup.

4.4 Experimental Results

Here are some performance results for updatable secondary CSIs. These are based on an early build that supports acceleration using bitmap filters for the anti-semi-join, but not row group pre-qualification based on min/max metadata. We started with the LINEITEM table from a 30GB TPC-H database, containing ~180M rows. We report warm numbers from a machine with 256GB of memory so there was virtually no IO involved. We restricted all statements to use a single core (Intel Xeon E5-2695 running at 2.4GHz) and measured insert, update, and scan performance for these configurations:

a) A clustered B-tree index on L_SHIPDATE, with page compression (the highest level of row store compression in SQL server, it reduces memory usage but increases the CPU cost for both reads and writes. Columnstore delta stores also use page compression).
b) A secondary columnstore index on top of (a), configured with a single delta store.
c) A primary columnstore index, also configured with a single delta store.

Table 3: Elapsed time (sec) of single-threaded insert and update operations

Operation	Rows affected	Row store	Secondary CSI	Primary CSI
1000 updates	10,000	0.893	1.400	6.866
10% insert	18.00M	233.9	566.0	291.4
2% update	3.96M	123.2	314.3	275.9

Table 3 summarizes insert and update performance. The first row shows elapsed time for 1000 small update statements, each updating 10 rows (by picking a day and updating the first 10 rows for that day). As expected, small updates are much faster (4.9X) for the secondary CSI than for the primary CSI which was the main goal of this improvement.

The second row shows elapsed times for inserting an additional ~18M rows to each index (a copy of the top 10% of existing rows, adding one year to L_SHIPDATE). We ran this as a single-statement, single-threaded insert, to show the relative overheads of inserting in the different indexes. We disabled the functionality that directly compresses large columnstore inserts without first landing them in delta stores. As expected, inserts into a table with a secondary CSI are significantly slower than insertions into a primary CSI because each row has to be inserted into both the base index and into the CSI delta store.

We then updated 2% of each index's rows to set the value of L_LINESTATUS, but with a skewed distribution such that 90% of the updates affect the newly inserted rows and the remaining 10% affects the original index rows. The update again ran in a single thread in a single statement. In the B-tree configuration this is an in-place update. For the columnstore configurations, it is processed as a delete followed by an insert (or an insert in the delete buffer followed by an insert in a delta store for compressed rows for a secondary CSI), which involves significant additional logging. Elapsed times are shown in the third row of Table 3.

The updates did not change the clustered B-tree structure, since they were all performed in-place. The secondary CSI has ~180M rows in compressed row groups, a delta store with ~20M rows, an empty delete bitmap, and a delete buffer with ~2M keys. The primary CSI similarly has ~2M entries in its delete bitmap. To show the effect of processing the delete buffer for secondary CSIs, we have also disabled the background process that flushes it to the delete bitmap.

The 10% insert and 2% update cases are bulk operations and would not be common in an OLTP environment. So the fact that they are 2-3 times slower than when using only a row store is not expected to significantly impede use of the updatable secondary CSI feature. The benefit to scan queries is substantial and should compensate for longer bulk update times. The key targets of the improvements are situations with OLTP updates touching a small number of rows each, and larger scans for analytical queries. These are covered by line one in Table 3 and lines 1 and 2 in Table 4.

Table 4: Elapsed time (sec) of single-threaded scan.

	Millions of rows	Row store	Secondary CSI	Primary CSI
Newly built	180	99.1	4.7	1.71
After 1000 updates	180	99.4	5.4	1.75
After 10% inserts	198	108.7	14.5	9.5
After 2% updates	198	109.5	16.8	10.0

Table 4 summarizes warm elapsed times for a simple query run before and after each step in Table 4. The query computes sum(L_QUANTITY * L_EXTENDEDPRICE), using a single-threaded full scan of each index. The first row shows results immediately after the indexes were created (so the delta stores, delete bitmaps, and delete buffers for the column stores are all empty). The column store scans perform much better than the page-compressed B-tree scan, both because they need to decompress much less data and because they avoid row-at-a-time processing. The row store index uses 2,145,959 8KB pages. The two columns requested by the query, L_QUANTITY and L_EXTENDEDPRICE, compress down to just 150,082 pages in the secondary CSI, and 141,350 pages in the primary CSI. The secondary CSI needs to additionally fetch and decompress the two key columns, L_SHIPDATE (22,365 pages) and the clustered B-tree's uniquifier (80,404 pages).

The scan performance after the initial updates (second row) is generally fairly close to the "clean" columnstore performance. The secondary CSI is affected more because there is now a non-empty delete buffer.

Scan performance after inserting 10% additional rows drops significantly for the columnstore configurations (third row of Table 3), because we now need to scan rows and convert them to batches. Columnstore performance degrades further for columnstore scans after updates (fourth row of Table 4), due to the increased size of the delete buffer and delete bitmap, but it is still significantly better than that of the row store scan.

The experiments clearly show that a non-empty delete buffer slows down scans of the column store so it is important to empty it as soon as possible.

5. B-TREES ON PRIMARY COLUMN STORES

A primary CSI is a space efficient way to store a table and provides superior scan performance which makes it an excellent choice for storing large tables that are used mostly for analytic queries. But even so, an analyst may sometimes need to drill down to a small portion or even a single row of data. Unfortunately, this still requires scanning the whole table if only a CSI is available.

In SQL Server 2014 a table with a CSI as its base index cannot have any primary key, foreign key or unique constraints. The reason for this restriction is that enforcing such constraints would be prohibitively expensive with only a CSI available.

To solve both these problems, SQL Server 2016 enables creation of B-tree indexes on top of primary CSIs. Point lookups and small range scans can then be done efficiently in B-trees, which will also makes it possible to enforce constraints efficiently.

5.1 Row Locators

A secondary B-tree index contains some subset of the columns in the underlying table, some of which comprise the key of the index. In addition, each index entry must contain a locator, that is, some information that uniquely identifies the source row in the base index. What the locator contains depends on the base index type: for a heap it is a row ID (page ID plus slot number) and for a B-tree it is the key of the base index, possibly augmented with a uniquifier column.

In a B-tree index over a CSI the locator consists of a row group ID and position of the row within the row group. However, rows in a CSI may be moved with creates a challenge. A primary CSI uses delta stores to absorb updates and (non-bulk) inserts. New rows may be inserted into a delta store first and later moved to a compressed row group by a background Tuple Mover task. When a row has moved, its old locator, which may be stored in multiple secondary indexes, becomes out-of-date. If so a lookup through a secondary index would fail to find the base row, which of course is not acceptable.

A locator also serves a second role: as a unique identifier of an entry in a secondary B-tree index. When we delete a row in the primary CSI, the locator value is used to uniquely identify the entry in the secondary B-tree index to delete. When a row is moved, its locator changes so it no longer identifies the correct index entry.

We could solve this problem by updating a row's locator whenever the row is moved. However, this would be inefficient for several reasons. A base row may participate in multiple indexes and we would have to update the locator in each one of them. A compressed row group consist of about one million rows. When creating a new compressed row group we would then have to update a million entries in each B-tree index. Not only is this slow, it also puts a lot of pressure on the transaction log. So we rejected this approach and looked for an alternative solution.

5.2 The Mapping Index

Our solution relies on an auxiliary table, a **Mapping Index,** that keeps track of row movement. There is only one Mapping Index even if the CSI has multiple B-tree indexes. The index maps a row's **original locator to its current locator.** It is stored as a B-tree with the original locator as the key. Whenever a row is moved, its entry in the mapping index is updated to indicate its current location.

When a row is created it is first inserted into a delta store or a compressed row group. In either case, it gets an original locator value

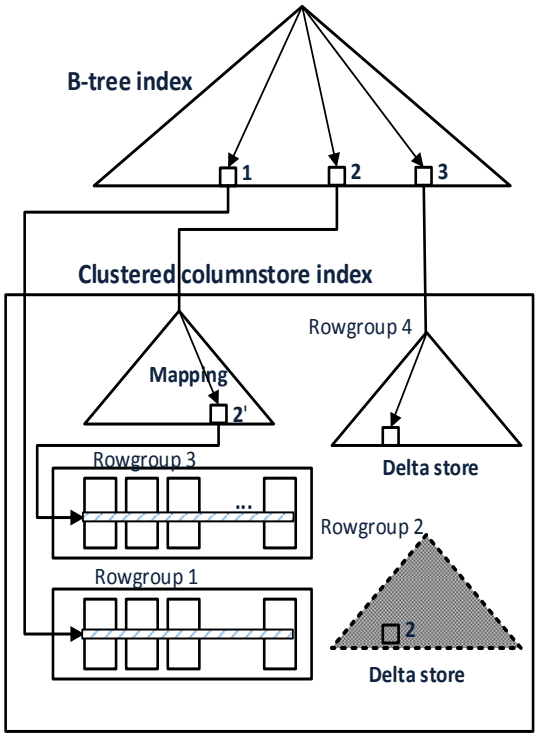

Figure 4: Main components of the design for B-tree indexes on primary column store indexes.

that is included in its index row in every secondary index. If the row is inserted into a compressed column store, the original locator is a row group ID and position within the row group. If it is inserted into a delta store, the original locator is the delta store's row group ID and a unique row number within the group which is assigned automatically when the row is inserted.

However, not all rows in a primary CSI will be moved. Rows inserted by a bulk load are compressed on the fly and are unlikely to be moved. To reduce the size of the mapping index, it only tracks rows that have moved. If a row is never moved, it will leave no trace in the mapping index. When a row is moved, its current location is recorded in the mapping index which avoids having to update its locator in multiple indexes.

Since rows in an entire row group are moved at the same time, we can avoid tracking the movement of each row individually and instead track the movement of ranges of rows. For example, a row in the mapping index may indicate that rows 1 to 1000 in (uncompressed) row group 2 have moved to (compressed) row group 3. This is much more efficient than modifying 1000 row locators, possibly in multiple B-tree indexes. The mapping index introduces a level of indirection for lookups via a B-tree index but it does not affect performance of scanning the column store.

A row in a primary CSI must also remember its original locator when it is moved. The original locator is used to uniquely identify the corresponding entry in a B-tree index which is required, for example, when deleting a row from the CSI. To this end an additional column (called "original locator") is added to the CSI when its first B-tree index is created. This column remains null if the row has not been moved. If the row is moved, it is set to the original locator value which is the same as the row's locator value in a B-tree index. Note that this operation can be done very efficiently because rows are moved one row group at time – it just adds another column segment to the new compressed row group.

5.3 Index Lookups

Once the mapping index is created and maintained, point lookups are straightforward. Given a row-group ID and row ID, first check whether the row group is visible. If it is visible, no lookup in the mapping index is needed. If it is not, do a lookup in the mapping index to get the row's current row-group ID and row Id. Then perform the lookup in the target row group or delta store.

Figure 8 shows the components of our design for B-tree indexes on a column store. When the first B-tree index on the table is created, a mapping index is also created. When the B-tree index in the figure was created, the CSI had only one compressed row group ("row group 1" in the figure) and a delta store ("row group 2"). Later on, row group 2 became full and was compressed into row group 3. At that point row group 2 became invisible and eventually its resources were released. A new delta store ("row group 4") was then created to accept new rows. When a new row is inserted into row group 4, a corresponding row is inserted into the B-tree index as well.

A lookup of a row using the B-tree index locates the target row in the base index in one of three ways as illustrated in the figure.

1. Index row 1 has a row locator pointing to a row in compressed row group 1 of the column store. Since row group 1 is visible, we locate the row directly by its position in the column segments of row group 1.
2. Index row 3 points to a row in row group 4. The row group is a delta store so we can use the locator to do a direct lookup on the key of the delta store.
3. Index row 2 has a row locator 2 pointing to the invisible row group 2. When we look up the base index row, we realize that row group 2 is invisible so its rows have all moved somewhere else. We then do a lookup in the mapping index and get the new locator 2' which directs us to a row in row group 3.

When row group 3 was created (from a delta store), a column containing the original locators of its rows was added to the row group. The locator value is the old row group ID plus the row's unique number within row group 2. So if we want to delete the row with locator 2' from row group 3 in CCI, the row will have a non-null "original locator" column with a value equal to locator 2. We can use the B-tree index key values plus locator 2 from the base row to identify the row in the B-tree index to delete.

5.4 Performance Results

We ran several experiments to evaluate the lookup performance and overhead on bulk inserts of a B-tree index on a column store.

5.4.1 Lookup performance of a B-tree on CSI

In the first experiment, we created two tables with one bigint column, one datetime column, and 20 payload columns of type varbinary(20). We randomly generated 10 million rows and bulk inserted them into the two tables.

Table A has a primary CSI and a secondary B-tree index on the bigint column. Table B has a B-tree base index on the datetime column and a secondary B-tree index on the bigint column. We then ran tests to compare the lookup performance on the two tables. If all the required columns are in the secondary index, lookup performance will be the same regardless of the base index type. Note that this is the case for indexes used for checking uniqueness and foreign keys.

However, if some columns have to be retrieved from the base index, lookups in table A will be slower than in table B because retrieving column values is more expensive in a column store than a row store. To get a column value we have to locate the corresponding column segment, possibly reading it from disk, and then find the correct position. If the column is dictionary encoded we also have to look up the actual value in the dictionary.

The experiment consisted of repeatedly looking up randomly selected rows and retrieving 4, 8 or 20 of the varbinary columns. We tested four different cases: table A with and without a mapping index and table B with and without page compression. In all experiments the data was entirely in memory.

The results of the experiments are shown in Table 5. The first column (without mapping) is for the cases where no mapping index is involved. This corresponds to a situation where the B-tree is created after all the data has been compressed or all data was bulk loaded. We report average elapsed times for 4, 8 and 20 output columns.

Table 5: Elapsed time per lookup in a B-tree index

Columns projected	B-tree over CSI (ms)		B-tree over B-tree (ms)	
	Without mapping	With mapping	No compression	Page compression
4	3.92	5.28	2.41	3.65
8	4.33	5.73	2.32	3.85
20	6.67	8.07	2.55	4.44

For the case "with mapping", we first created an empty table with the B-tree index defined, inserted all data into delta stores, and then compress all row groups. All lookups then have to go through the mapping index to locate a base row which increases the lookup time.

For the case of a secondary B-tree index over a B-tree base index we also tested two cases: with and without page compression enabled on the base index. Turning compression on increased the elapsed time by 50-75%.

As expected, lookups in a table A were slower than lookups in table B but only by a factor of 2 to 3 depending on how many columns must be retrieved from the base index. While this may sound slow, it is certainly much faster than a complete scan. Furthermore, keep in mind that primary CSIs are normally used for data warehousing applications and not for high-performance OLTP applications.

5.4.2 Overhead when adding a new data partition

In data warehousing applications, tables are often partitioned and partition switching is a common way to load large batches of data into the table. To add a new partition of data into the table, a user just needs to make sure that the new partition has exactly the same organization. The extra cost of having a B-tree index over a CSI is the cost to create the B-tree index on the partition to be switched in.

In this experiment, we used a table with the same schema as table A. Table 6 compares the cost of adding a new partition data with or without the B-tree index.

Table 6: Overhead on data loading of having an NCI

New partition size (million rows)	Time for bulk load into CSI (ms)	Time to create B-tree (ms)	Ratio (index creation / bulk load)
1	5,327	66	1.24%
5	5,335	77	1.45%
10	5,354	83	1.55%

Due to parallel execution, the elapsed time for bulk loading 1M or 10M rows remained about the same – multiple row groups were compressed in parallel. The overhead of creating the B-tree index was minor, around 1.5%.

6. CSI SCAN IMPROVEMENTS

This section outlines several improvements aimed at speeding up columnstore scans. Section 6.1 briefly describes the main operations involved in scanning: decompression of data and evaluating predicates. Section 6.2 explains architectural changes in the scan operator and the motivation behind them. The following sections focus on three scan improvements: performing additional operations on compressed data, opportunistic pushing of additional operations into scan operator, and using vector instructions (SIMD) available in modern CPUs. The last section shows performance results on a few example queries.

6.1 Compression and Scan Functionality

A compressed row group consists of a column segment for each column. The values in a column are first encoded (converted to a 32-bit or 64-bit integer) using either value encoding or dictionary encoding. Value encoding applies a linear transformation on numerical values to convert them to integers that can be represented with a smaller number of bits. Dictionary encoding is used when the number of distinct values is much smaller than the size of the segment and, unlike value encoding, can be used for both numeric and non-numeric data. The actual values are stored in dictionaries and replaced by data ids in the column segment. In case of non-numeric data, mainly strings, values in the dictionary are packed using Huffman encoding. A column segment may reference two dictionaries: a shared global dictionary and a local dictionary, which is associated with a single row-group. After encoding each column segment is compressed using a mixture of RLE compression and bit packing.

Every compressed column segment contains two arrays: an RLE array and bit-packed values array. The RLE array partitions an ordered sequence of column values into pure runs (the same value repeated multiple times) and impure runs in which each value is encoded separately. Values from impure runs are stored in a bit-packed values array. As few bits as possible are used to encode a column values but the bits of a single value cannot cross a 64-bit word boundary.

The first stage of decompression yields pure sequences of values or impure sequences of values after bit unpacking which then pass through a decoding stage. Decoding numeric values involves either a linear transformation of the value or a dictionary lookup.

In SQL 2014 the scan operator is responsible for the following transformations for each column requested: bit unpacking, decoding, filtering and normalization. Some filters in the execution plan can be pushed down to the scan and evaluated in the storage engine layer. In SQL 2014 all filters are evaluated on decoded values. Normalization is a data transformation process required for outputting rows qualified by the filter in a structure called a row batch used in the query execution layer for exchanging data between operators.

6.2 New Scan Design

The scan operator was redesigned to improve scan performance by means of the following techniques:

a) Operating on encoded data directly for evaluating filters or aggregation whenever possible,
b) Using SIMD instructions to operate on multiple scalar values simultaneously,
c) Avoiding conditional expressions and related branch misprediction penalties.

The improvements were mostly aimed at impure sequences of values. SQL Server is already taking advantage of RLE compression when processing filters, joins and grouping on columns with pure sequences of values. The primary goal was to better exploit data organization and information about distribution of values resulting from dictionary encoding and bit-packing.

In SQL 2014 processing inside the scan operator is organized as follows. A row group is processed in units corresponding to output batches, also called row buckets, each with around one thousand rows. All output columns are processed one by one. Processing of a column is done as a single loop over all involved rows. Iteration of a loop does end to end processing of a single column value: bit unpacking, decoding, evaluating filters, and normalization. This loop is done by calling a function pointer, determined during query compilation, corresponding to one of over ten thousand specialized statically compiled implementations generated for all possible combinations of data type, encoding parameters and filter types.

We redesigned the scan so that processing of a single row-bucket column is split into several more basic precompiled operations each running as a sequential pass over all the values. The new strategy resulted in code consisting mostly of small independent blocks, which simplified greatly adding functionality to the scan, like new variants of filters and aggregation, and especially eased the transition to using SIMD instructions.

6.3 Operating Directly on Compressed Data

Unlike in earlier version, filters pushed down to columnstore scan can be evaluated on data before decoding. This works for value based encoding (linear transform of numerical values) and dictionary encoding for both string and numeric dictionaries.

Arbitrary filters on a dictionary encoded column are translated into a bitmap filter. The bitmap in this case contains one bit for each dictionary entry that indicates whether the entry satisfies the filter. If the resulting set of bits set correspond to one contiguous range of bits, it is further transformed into comparison filter. Filtering this way is beneficial because typically dictionaries contain significantly fewer entries than there are rows referencing them. The result is fewer evaluations of the filter. If the dictionary is larger than the row group, the bitmap filter will not replace the original filter.

Scalar aggregates can be evaluated before decoding for value-based encoded columns, in which case the linear transformation is only applied to the result.

6.4 Opportunistic Pushdown of Operations

Pushing down data-reducing operations to the scan is particularly important for performance, especially if these operations can be performed before later stages of data decompression. The new scan operator supports a broader repertoire of filters on string columns and scalar aggregates.

Evaluation of the newly added operations happens opportunistically. During query execution both ways of processing of rows are available: the old way that uses traditional data flow through query execution operators (slow path) and the new way that uses its replacement implemented in scan (fast path). For each row-bucket the decision can be made separately as to which path will be taken for its rows. For example, the scan operator can implement a simplified version of sum computation, that optimistically assumes that there will be no overflow and in case of any risk that this condition may not be satisfied reverts to the robust implementation provided by the slow path. Having two ways of processing rows available simultaneously also means that no changes are needed for delta stores, and support for compressed row-groups can be limited based on type of encoding and its parameters such as size of dictionaries.

6.5 Exploiting SIMD Instructions

Modern CPUs, such as the latest Haswell family from Intel, have a special set of 256-bit registers and a set of instructions that operate on them that allow for simultaneous processing of multiples of 1, 2, 4 or 8-byte floating point or integer values. SQL Server 2016 utilizes integer variants belonging to the AVX2 instruction set. Every operation that uses SIMD also has a variant working with previous generation on 128-bit registers with the SSE4.2 instruction set as well as plain non-SIMD implementation, with the right version chosen at run-time based on CPU and OS capabilities.

SIMD is used for column operations on bit-packed or encoded data. The benefits of SIMD instructions are higher with shorter data values because more of them can fit into a single wide register. Information about bits used for bit packing compression allows to choose the smallest possible element size for representing encoded column values. We use specialized implementations for 2, 4 and 8-byte encoded values.

The scan code contains SIMD variants of the following column operations: bit unpacking, comparison filters, bitmap filters, compaction, and aggregation. Compaction is an operation that takes two vectors: one with input values and one with Booleans, and removes from the first one all entries marked in the second one, densely packing the remaining ones. It allows to avoid conditional expressions in the code, which are especially inconvenient for SIMD code where multiple adjacent array entries are processed together.

The bitmap filter implementation has two variants, one when the bitmap size does not exceed a single wide register and is stored there directly, while the other references bitmap bits in memory and uses gather instruction in AVX2 to load multiple words at different offsets into a single SIMD register.

Bit unpacking with SIMD presents some engineering challenges, because every case of different numbers of bits requires a slightly different code path and set of constants. Our solution was to create a C++ code generator to handle most of these tasks automatically.

Another common problem with code using SIMD is the handling of the tail of each array of values, which may be smaller than the number of elements in a SIMD register. We chose to reserve extra space at the end of each array involved and use a slightly modified SIMD loop that masks out the extra elements at the end.

Table 7: Comparing performance of basic operations with and without SIMD instructions

Operation	Billions of values per second		Speedup
	No SIMD	SIMD	
Bit unpacking 6bits	2.08	11.55	5.55X
Bit unpacking 12 bits	1.91	9.76	5.11X
Bit unpacking 21 bits	1.96	5.29	2.70X
Compaction 32 bits	1.24	6.70	5.40X
Range predicate 16 bits	0.94	11.42	5.06X
Sum 16 bit values	2.86	14.46	5.06X
128-bit bitmap filter	0.97	11.42	11.77X
64KB bitmap filter	1.01	2.37	2.35X

Table 7 compares the performance of elementary building blocks of scan code executing with SIMD (AVX2) and without SIMD instructions. The table shows billions of input values processed per second on a single core of an Intel Haswell CPU with 2.30 GHz clock. The test was designed so that the input data resides in the L1 cache (no cache misses). The data does not include other overheads such as the costs of initializations, metadata checks or context switching that appear during actual execution of the query.

6.6 Query Performance

In this section we present performance results for the old and the new columnstore scan implementation.. The data was collected using a single Intel Haswell CPU with 12 cores, 2.30 GHz and hyper-threading disabled. We did not observe any significant difference for this experiment with hyper-threading enabled. For all of the queries we made sure that query plans were the same for both compared implementations. The database used was TPC-H 300G database with primary CSIs created on every table with no delta stores and no partitions. We made sure all input data needed was fully cached in-memory before running queries. There was no row-group elimination based on filters. All queries used new features: SIMD, string filter push down, scalar aggregate push down.

Table 8: Query performance with and without scan enhancements (using 12 cores)

Predicate or aggregation functions	Duration (ms)		Speed up	Billion rows per sec
	SQL 2014	SQL 2016		
Q1-Q4: select count(*) from LINEITEM where *<predicate>*				
L_ORDERKEY = 235236	220	140	1.57x	12.9
L_QUANTITY = 1900	664	68	9.76x	26.5
L_SHIPMODE='AIR'	694	147	4.72x	12.2
L_SHIPDATE between '01.01.1997' and '01.01.1998'	512	87	5.89x	20.7
Q5-Q6: select count(*) from PARTSUPP where *<predicate>*				
PS_AVAILQTY < 10	50	27	1.85x	8.9
PS_AVAILQTY = 10	45	15	3.00x	16
Q7-Q8: select *<aggregates>* from LINEITEM				
avg(L_DISCOUNT)	1272	196	6.49x	9.1
avg(L_DISCOUNT), min(L_ORDERKEY), max(L_ORDERKEY)	1978	356	5.56x	5.1

The queries used were of two types. The first six queries counted rows qualified by a filter on a single column with filters on columns of different data types and encoded in a different ways. The last two queries evaluated scalar aggregates on the entire table.

The filter column in Q1 used 32-bit packing with no dictionary encoding and no RLE compression. The memory read bandwidth used was approaching 50GB/s, which is close to hardware limits. In Q2 the filter is on a column with 6-bit packing, dictionary encoding and no RLE compression. The great speedup comes from evaluating the predicate before doing dictionary lookup using a bitmap filter computed from the dictionary and the given predicate. In this case the bitmap is small enough to be stored in a single SIMD register which makes bit lookups very fast. Query Q3 shows the improvement coming from string filter push-down. The column uses RLE compression and both SQL 2014 and SQL 2016 takes advantage of this when evaluating the filter. The only difference is that the scan in SQL 2016 applies string filter to a dictionary used

in column encoding first and transforms the original predicate into search for a specific integer within encoded column.

Queries Q4 and Q5 use the same column with 16-bit packing, dictionary encoding and no RLE compression. The difference is that the first of the two uses range predicate, which gets converted to a bitmap filter and the second one uses equality which becomes equality predicate on encoded value. SQL does not use sorted dictionaries and therefore range predicates on a column with dictionary correspond to a set of indices that do not usually make up a single contiguous range.

7. RELATED WORK

This section gives a brief overview of systems that include both row stores and column stores. It does not cover systems that are either pure row stores or pure column stores.

SQL Server was the first to integrate columnar storage into a row-based DBMS but all major commercial database systems have now done the same. However, the designs vary considerably. IBM markets columnar storage under the common term "BLU Acceleration" but there are several different implementations. DB2 for Linux, Unix and Windows supports column-organized tables similar SQL Server's primary CSI but does not allow additional indexes on such tables [16]. It also supports shadow tables which are similar to SQL Server's secondary CSI but shadow tables are maintained by replication so their content lags behind the primary tables. Thus queries cannot combine data from regular tables and shadow tables. The Informix Warehouse accelerator is based on an earlier version of BLU technology and supports what amounts to shadow tables stored in memory [4]. DB2 for z/OS relies on an attached analytics accelerators based on Netezza [5].

Oracle has recently added columnar storage [14]. However, the column store is only an in-memory cache – column segments, dictionaries, etc. are not stored persistently but computed on demand or at system startup.

Pivotal Greenplum Database began as a row store but now includes column store capabilities [15]. Their Polymorphic Storage feature allows different partitions of the same table to be stored in different form, some row-wise and some column-wise.

Teradata introduced columnar storage in Teradata 14 [18]. 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.

The systems listed above use traditional disk-based storage for data in row format. SAP HANA [17] and MemSql [11] are two hybrid commercial systems that require all data to be stored in memory as does HyPer [6], a prototype system developed at the Technical University of Munich.

8. CONCLUDING REMARKS

Columnstore indexes and batch processing are key to efficient processing of analytical queries. With the enhancements added in SQL Server 2016, columnstore indexes can be used anywhere where fast analytical processing is needed. In data warehousing applications they can be used as base indexes and, if needed, augmented with secondary B-tree indexes to speed up lookups and enable uniqueness and foreign key constraints. In OLTP applications they can be used as secondary indexes on both in-memory and disk-based tables to enable real-time analytics concurrently with transactional processing.

9. ACKNOWLEDGMENTS

We thankfully acknowledge the contributions of the many members of the Apollo and Hekaton teams. Without their hard work and dedication the improvements described in this paper would not have been possible.

10. REFERENCES

[1] Kalen Delaney, *SQL Server In-Memory OLTP Internals Overview*, Red gate books, 2014.

[2] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, Mike Zwilling: Hekaton: SQL server's memory-optimized OLTP engine. *SIGMOD 2013:* 1243-1254

[3] Craig Freedman, Erik Ismert, Per-Åke Larson: Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Engeenering. Bulletin* 37(1): 22-30 (2014)

[4] IBM, IBM DB2 Analytics Accelerator for z/OS, http://www03.ibm.com/software/products/en/db2analacceforzos

[5] IBM, Informix Warehouse, http://www-01.ibm.com/software/data/informix/warehouse/

[6] Alfons Kemper et. al., Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer. *IEEE Data Engineering Bulletin* 36(2): 41-47 (2013)

[7] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, Qingqing Zhou: SQL server column store indexes. *SIGMOD 2011*: 1177-1184

[8] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, Mike Zwilling: High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB* 5(4): 298-309 (2011)

[9] 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: Enhancements to SQL server column stores. *SIGMOD 2013*: 1159-1168

[10] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta: The Bw-Tree: A B-tree for new hardware platforms. *ICDE 2013:* 302-313

[11] Memsql database, http://www.memsql.com/product/

[12] Microsoft, In-Memory OLTP (In-Memory Optimization), http://msdn.microsoft.com/en-us/library/dn133186.aspx

[13] Microsoft, Columnstore Indexes Described, http://msdn.microsoft.com/en-us/library/gg492088.aspx

[14] Oracle, Oracle Database In-Memory, http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.html

[15] Pivotal Greenplum Database, http://pivotal.io/big-data/pivotal-greenplum-database

[16] Vijayshankar Raman et. Al., DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB* 6(11): 1080-1091 (2013)

[17] SAP HANA, http://hana.sap.com/abouthana.html

[18] Teradata 14 Hybrid Columnar, http://www.teradata.com/Resources/White-Papers/Teradata-14-Hybrid-Columnar