SQL Server Column Store Indexes

Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price,
Srikumar Rangarajan, Aleksandras Surna, Qingqing Zhou
Microsoft

{palarson, ciprianc, ehans, artemoks, susanpr, srikumar, asurna, qizhou}@microsoft.com

ABSTRACT

The SQL Server 11 release (code named "Denali") introduces a new data warehouse query acceleration feature based on a new index type called a column store index. The new index type combined with new query operators processing batches of rows greatly improves data warehouse query performance: in some cases by hundreds of times and routinely a tenfold speedup for a broad range of decision support queries. Column store indexes are fully integrated with the rest of the system, including query processing and optimization. This paper gives an overview of the design and implementation of column store indexes including enhancements to query processing and query optimization to take full advantage of the new indexes. The resulting performance improvements are illustrated by a number of example queries.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Design

Keywords

Columnar index, column store, OLAP, data warehousing.

1. INTRODUCTION

Database systems traditionally store data row-wise, that is, values from different columns of a record are stored together. This data organization works well for transaction processing where requests typically touch only a few records. However, it is not well suited for data warehousing where queries typically scan many records but touch only a few columns. In this case, a column-wise organization where values from the same column in different records are stored together performs much better. It reduces the data processed by a query because the query reads only the columns that it needs and, furthermore, column-wise data can be compressed efficiently. Systems using column-wise storage are usually referred to as column stores.

SQL Server is a general-purpose database system that stores data in row format. To improve performance on data warehousing queries, SQL Server 11.0 (code named "Denali") adds column-wise storage and efficient column-wise processing to the system. This capability is exposed as a new index type: a column store index. That is, an index can now be stored either row-wise in a B-tree or column-wise in a column store index.

SIGMOD'10, June 12–16, 2011, Athens, Greece.

SQL Server column store indexes are "pure" column stores, not a hybrid, because they store all data for different columns on separate pages. This improves I/O scan performance and makes more efficient use of memory. SQL Server is the first major database product to support a pure column store index. Others have claimed that it is impossible to fully incorporate pure column store technology into an established database product with a broad market. We're happy to prove them wrong!

To improve performance of typical data warehousing queries, all a user needs to do is build a column store index on the fact tables in the data warehouse. It may also be beneficial to build column store indexes on extremely large dimension tables (say more than 10 million rows). After that, queries can be submitted unchanged and the optimizer automatically decides whether or not to use a column store index exactly as it does for other indexes.

We illustrate the potential performance gains by an example query against a 1TB TPC-DS [10] test data warehouse. In this database, the catalog_sales fact table contains 1.44 billion rows. The following statement created a column store index containing all 34 columns of the table:

```
CREATE COLUMNSTORE INDEX cstore on catalog_sales
  ( cs_sold_date_sk, cs_sold_time_sk, ···
    ···,cs_net_paid_inc_ship_tax, cs_net_profit)
```

We ran the following typical star-join query on a pre-release build of Denali, with and without the column store index on the fact table. All other tables were stored row-wise only. The test machine had 40 cores (hyperthreading was enabled), 256GB of memory, and a disk system capable of sustaining 10GB/sec.

```
select w_city, w_state, d_year,
       SUM(cs_sales_price) as cs_sales_price
from warehouse, catalog_sales, date_dim
where w_warehouse_sk = cs_warehouse_sk
  and cs_sold_date_sk = d_date_sk
  and w_state = 'SD'
  and d_year = 2002
group by w_city, w_state, d_year
order by d_year, w_state, w_city;
```

The query was run twice, first with a cold buffer pool and then with a warm buffer pool. With a warm buffer pool, the input data for the query all fit in main memory so no I/O was required.

Table 1: Observed CPU and elapsed times (in sec)

	Cold buffer pool		Warm buffer pool	
	CPU	Elapsed	CPU	Elapsed
Row store only	259	20	206	3.1
Column store	19.8	0.8	16.3	0.3
Improvement	13X	25X	13X	10X

The results are shown in Table 1. The column store index improves performance dramatically: the query consumes 13 times less CPU time and runs 25 times faster with a cold buffer pool and 10 times faster with a warm buffer pool. SQL Server column store technology gives subsecond response time for a star join query against a 1.44 billion row table on a commodity machine. This level of improvement is significant, especially considering that SQL Server has efficient and competitive query processing capabilities for data warehousing, having introduced star join query enhancements in SQL Server 2008.

The machine used has a high-throughput I/O system (10GB/sec) which favors the row store. On a machine with a weaker I/O system, the relative improvement in elapsed time would be even higher.

The rest of the paper provides more detail about column store indexes. Section 2 describes how they are stored including how they are compressed. Section 3 describes extensions to query processing and query optimization to fully exploit the new index type. Section 4 provides some experimental results and section 5 summarizes related work.

2. INDEX STORAGE

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

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

2.1 Column-Wise Index Storage

We now outline how a column store index is physically stored. Figure 1 illustrates the first step that converts rows to column segments. The set of rows to be stored is first divided into row groups, each group consisting of, say, one million rows. Each row group is encoded and compressed independently. The result is one compressed column segment for each column included. Figure 1 shows a table divided into three row groups where three of the four columns are included in the column store index. The result is nine compressed column segments, three segments for each of columns A, B, and C.

The column segments are then stored using existing SQL Server storage mechanisms as shown in Figure 2. Each column segment is stored as a separate blob (LOB). Segment blobs may be large, requiring multiple pages for storage, but this is automatically handled by the existing blob storage mechanisms. A segment directory keeps track of the location of each segment so that all segments of a given column can be easily located. The directory is stored in a new system table and visible through the catalog view sys.column_store_segments. The directory also contains

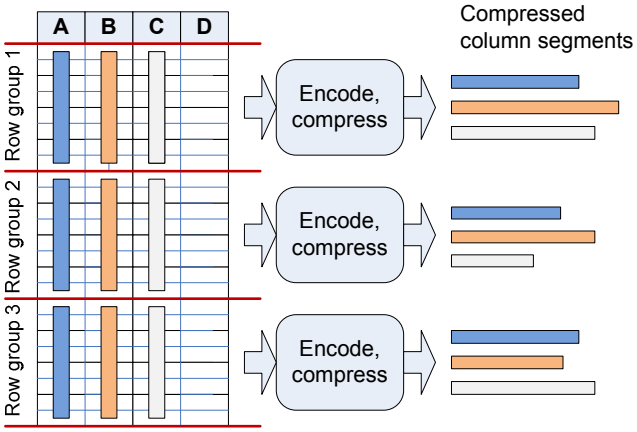

Figure 1: Converting rows to column segments

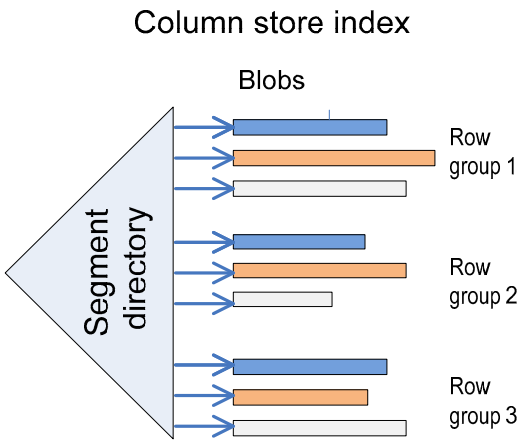

Figure 2: Storing column segments

additional metadata about each segment such as number of rows, size, how data is encoded, and min and max values.

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

2.2 Data Encoding and Compression

Data is stored in a compressed form to reduce storage space and I/O times. The format chosen allows column segments to be used without decompression in query processing. Compressing the columns in a segment consists of three steps.

1. Encode values in all columns.
2. Determine optimal row ordering.
3. Compress each column

2.2.1 Encoding

The encoding step transforms column values into a uniform type: a 32-bit or 64-bit integer. Two types of encoding are supported: a dictionary based encoding and a value based encoding.

The dictionary based encoding transforms a set of distinct values into a set of sequential integer numbers (data ids). The actual

values are stored in a data dictionary, essentially an array that is indexed by data ids. Each data dictionary is stored in a separate blob and kept track of in a new system table which is visible through the catalog view sys.column_store_dictionaries.

The value based encoding applies to integer and decimal data types. It transforms the domain (min/max range) of a set of distinct values into a smaller domain of integer numbers. The value based encoding has two components: exponent and base.

For decimal data types the smallest possible positive exponent is chosen so that all values in a column segment can be converted into integer numbers. For example, for values 0.5, 10.77, and 1.333, the exponent would be 3 (1000) and the converted integer numbers would be 500, 10770, and 1333 correspondingly.

For integer data types the smallest possible negative exponent is chosen so that the distance between the min and the max integer values in a column segment is reduced as much as possible without losing precision. For example, for values 500, 1700, and 1333000, the exponent would be -2 (1/100) and the converted integer numbers would be 5, 17, and 13330.

Once the exponent is chosen and applied, the base is set to the min integer number in the column segment. Each value in the column segment is then adjusted (rebased) by subtracting the base from the value. For the decimal example above, the base would be 500 and the final encoded values (data ids) would be 0, 10270, and 833. For the integer example, the base would be 5 and the final encoded values would be 0, 12, and 13325.

2.2.2 Optimal Row Ordering

Significant performance benefits accrue from operating directly on data compressed using run-length encoding (RLE), so it is important to get the best RLE compression possible. RLE gives the best compression when many identical values are clustered together in a column segment. Since the ordering of rows within a row group is unimportant, we can freely rearrange rows to obtain the best overall compression. For a schema containing only a single column, we get the best clustering simply by sorting the column as this will cluster identical values together. For schemas with two or more columns it is not that simple - rearranging rows based on one column can negatively affect clustering of identical values in other columns.

We use the Vertipaq™ algorithm to rearrange rows within a row group in an order that achieves maximal RLE compression. This patented compression technology is shared with SQL Server Analysis Services and PowerPivot.

2.2.3 Compression

Once the rows within a row group have been rearranged, each column segment is compressed independently using RLE compression or bit packing.

RLE (run-length encoding) compression stores data as a sequence of <value, count> pairs. The actual value is a 32-bit or 64-bit number containing either an encoded value or a value stored as is.

RLE compression thrives on long runs of identical values. If a column contains few long runs, RLE compression may even increase the space required. This occurs, for example, when all values are unique. Since values in a column segment get encoded into a smaller domain of integer numbers (data ids) in most cases, the actual range of encoded values will usually require fewer bits to represent each encoded value. Therefore, we also support a bit-pack compression and different bit-pack compression sizes.

2.3 I/O and Caching

A blob storing a column segment or dictionary may extend over multiple disk pages. When brought into memory, column segments and dictionaries are stored not in the page-oriented buffer pool but in a new cache designed for handling large objects. Each object in the cache is stored in consecutive storage, not scattered across discrete pages. This simplifies and speeds up scanning of columns because there are no "page breaks".

To improve I/O performance, read-ahead is applied both within and among segments. That is, when reading a blob storing a column segment, read-ahead is applied at the page level. A column may be stored as multiple segments so read-ahead is also applied at the segment level. Finally, read-ahead is also applied to data dictionaries.

For on-disk storage additional compression could be applied. When a column segment is written to disk, it could be further compressed by applying some standard streaming compression technique and automatically decompressed when being read into memory. Whether or not to apply additional compression is a tradeoff: it reduces both disk space and I/O requirements but increases CPU load.

3. QUERY PROCESSING AND OPTIMIZATION

3.1 Query Processing Enhancements

For queries that scan a large number of rows, using a column store index may reduce the amount of data read from disk by orders of magnitude. Such a large reduction in disk I/O very likely causes CPU resources to become the next bottleneck. To keep the system balanced it was thus necessary to significantly reduce CPU consumption for queries processing large numbers of rows.

Standard query processing in SQL Server is based on a row-at-a-time iterator model, that is, a query operator processes one row at a time. To reduce CPU time we introduced a new set of query operators that instead processes a batch of rows at a time. As has been observed before [8], batch-at-a-time processing significantly reduces the overhead for data movement between operators. The batch operators are optimized for data warehouse scenarios; they are not intended as replacements for row-at-a-time operators in OLTP workloads.

We chose not to implement a new engine specialized for data warehousing applications but instead opted to extend the existing SQL Server engine. This has several advantages.

1. Customers don't have to invest in a new engine and transfer data between engines. It is the same SQL Server system that they are already used to; it just has a new index type that greatly speeds up decision support queries. This is an important and tangible customer benefit.
2. It greatly reduces implementation costs. SQL Server is a mature product with lots of features that are automatically available, for example, query plan diagrams, query execution statistics, SQL profiling, SQL debugging, and so on.
3. A query plan can mix the two types of operators. The query optimizer has been extended to select the best operator type to use. It typically choses the new faster operators for the expensive parts of a query that process large numbers of rows.

4. Queries may dynamically switch at runtime from using batch operators to using row operators as necessary. For example, currently joins and aggregations that spill to disk automatically switch to row-at-a-time operators.
5. We get feature orthogonality. For example, the new operators support the majority of existing SQL Server data types, session level settings and so on. Any SQL query can take advantage of the faster processing offered by the new operators, not just stylized star-join queries.

The new batch iterator model is independent of the access methods supported by different data sources. Similarly to access methods for other data sources, access methods for column store indexes support filtering by predicates and bitmap filters. Whenever possible they perform operations directly on compressed data. The new batch operators are then typically used for the data intensive part of the computation, performing more complex filtering, projections, joins and aggregation. Row operators may sometimes be needed to finish the computation.

Certain access methods may expose additional optimizations such as delayed string materialization, and the new iterators are designed to transparently take advantage of these features whenever possible.

While processing batches of rows at a time can by itself achieve very significant reduction in processing time, it cannot achieve orders of magnitude reduction.

Several additional improvements were also implemented:

1. The new iterators are fully optimized for the latest generation of CPUs like Intel Nehalem or AMD Opteron architecture. In particular, the algorithms were designed to take advantage of increased memory throughput and better utilize 64-bit processing and multicore CPUs.
2. The bitmap filter implementation was modified to have several specific data layouts based on data distribution.
3. Runtime resource management was improved by making operators share available memory with each other in a more flexible manner.

For queries that process large numbers of rows, the net result of all these improvements is order of magnitude performance improvements. Row-at-a-time based operators are still preferred for short OLTP queries. The query optimizer automatically chooses the proper operators at query compilation time; no application changes are required.

3.2 Query Optimization

In order to leverage the storage engine and query execution capabilities described above, several changes were required in the query optimizer. Unlike regular indexes, column store indexes do not (efficiently) support point queries and range scans, do not offer any sort order and, since sorting is not required to build them, they do not offer any statistics. All these properties were "de facto" assumptions in the row store world. On the other hand, column store indexes offer high data compression, highly effective usage of modern CPUs, and much reduced I/O. While point queries will still heavily favor row stores, queries requiring expensive scans will benefit greatly from column store indexes. The query optimizer can get very accurate information about the actual on-disk size of columns and uses this information to estimate the amount of IO required.

Column store indexes are further complemented by the new batch operators. The set of batch operators is limited in the initial release and it is up to the query optimizer to ensure batch execution is used where possible and beneficial. Mixing batch and row operators is possible but converting data from one format to the other has its own costs. Rather than creating a complex cost based mechanism to decide when to convert between the two modes the initial implementation uses a more conservative approach that limits the number of conversions.

We introduced the concept of a *batch segment* in a query plan. A batch segment consists of a sequence of batch operators that execute using the same set of threads. An operator can only extend the segment to a single child. In our implementation a join always extends its segment to the probe side. A new BatchHashTableBuild operator builds the hash table for a batch hash join. It is a batch operator that can accept input either in batch or row mode.

In order to make generation of these plan shapes possible, the query optimizer introduced a new physical property used to distinguish between batch and row output of an operator. Using this property, it is now easy to make sure we do not have unnecessary transitions between row and batch operators. All batch operators will request that their children provide batches and all row operators will request their children to provide rows. Transitioning between the two modes is achieved by the ability of some operators to output either rows or batches.

As stated above, batch hash join does not build its hash table; it is built by the BatchHashTableBuild operator. BatchHashTableBuild also has the ability to build a bitmap (Bloom filter) on the hash key; this bitmap can be used to filter out rows early from the probe side input. These bitmaps are even more important when multiple batch hash joins are stacked. Bitmap filters are typically very effective in filtering out rows early. In the same way as regular selection predicates they can be pushed down all the way into the storage engine (access method).

We also introduced new methods for handling multiple joins. The SQL Server optimizer tries to collapse inner joins into a single n-ary join operator. Our plan generation for column store indexes uses the n-ary join as the starting point because the n-ary join gives us the benefit of being able to inspect the entire join graph at once rather than having to deal with just a subset of the tables involved. Our algorithm consists of several steps.

We first go over the expressions joined together and analyze the join predicates involved, trying to identify which join keys are unique. Using the uniqueness information we identify fact tables as the tables that do not have any unique key involved in a join.

Once this is done, starting from the smallest fact table, we expand the join graph to cover as many dimension tables as possible by only traversing many-to-one relationships. This will build a snow-flake around the fact table. We then continue with the other fact tables in increasing size order. This step ends when there are no expressions from the original set left.

If multiple fact tables were identified initially, we will have multiple snowflakes and these snowflakes need to be joined together. Beginning with the snowflake with the largest fact table, we recursively add neighboring snowflakes as dimensions of the snowflake built so far. At this point we have a single snowflake expression and can start generating the final plan shape.

First we try to identify which joins are worthy candidates for building bitmaps. Once the bitmap expressions are identified we start building a right deep join tree with the dimensions on the left side and the fact table on the right side of the rightmost join. Keep in mind that each dimension can in turn be another snowflake and the algorithm has to expand them recursively. At each join, certain conditions are checked to ensure batch execution compatibility. If the fact table does not meet the criteria for batch execution then the tree generated cannot use batch hash joins. If the fact table meets the criteria, then each join is analyzed and all the joins that are batch-able are placed at the bottom and the remaining joins are left at the top.

In practice, this algorithm reliably builds maximal-size batch hash join pipelines for star join sub-expressions within a query, for star joins centered on a fact table with a column store index.

The optimizer chooses the best plan based on estimated costs. The optimizer's cost model of course had to be augmented to include the new batch operators and column store indexes.

4. EXPERIMENTAL RESULTS

This section presents early experimental results on compression rates for six databases and performance improvement for four example queries. All results were obtained on a pre-release build of SQL Server 11.0 (code named "Denali").

4.1 Data Compression

Compression ratios for artificially generated data are uninteresting at best and misleading at worst; what matters are the results on real data. Table 2 below shows the size of an uncompressed fact table, the size of a column store index containing all columns of the table, and the compression ratio for six real data sets. The "Cosmetics" data set is from an orders fact table from a cosmetics manufacturer. The "SQM" data set is from an internal Microsoft data warehouse that tracks SQL Server usage patterns. The "Xbox" data set tracks use of Xbox Live. "MSSales" contains Microsoft sales data. The "Web Analytics" data set contains web clickstream information. The "Telecom" data set contains call detail records from a telecommunications data warehouse.

Table 2: Column store compression on real data sets

Data Set	Uncompressed table size (MB)	Column store index size (MB)	Com-pression Ratio
Cosmetics	1,302	88.5	14.7
SQM	1,431	166	8.6
Xbox	1,045	202	5.2
MSSales	642,000	126,000	5.1
Web Analytics	2,560	553	4.6
Telecom	2,905	727	4.0

On average, column store compression is about 1.8 times more effective than SQL Server PAGE compression, the highest form of compression implemented for the SQL Server row store structures (heaps and B-trees), which was introduced in SQL Server 2008. In other words, a column store index is about 1/1.8 = 0.56 times the size of a PAGE compressed B-tree or heap containing the same data.

4.2 Example queries

In this section we illustrate the performance improvements that can be achieved by using column store indexes. Four queries against a TPC-DS [10] database at scale factor 100 are included.

A TPC-DS database at scale factor 100 is intended to require about 100GB for the base tables. For SQL Server the space requirements were 92.3 GB for data, 15.3GB for secondary (row store) indexes and 36.6GB for column store indexes covering all columns on every table. Our example queries used the five tables listed below. Each table had a B-tree clustering index and a column store index that included all columns. The two larger (fact) tables had one secondary index each but not the three smaller (dimension) tables. We found that dropping the column store indexes on the three small tables had virtually no effect on the observed times.

Catalog_sales (144M rows)

- Clustering index: cs_sold_date_sk_cluidx (cs_sold_date_sk)
- Secondary index: cs_item_sk_cs_order_number_idx (cs_item_sk, cs_order_number)
- Column store index: catalog_sales_cstore(all columns)

Catalog_returns (14.4M rows)

- Clustering index: cr_returned_date_sk_cluidx (cr_returned_date_sk)
- Secondary index: cr_item_sk_cr_order_number_idx (cr_item_sk, cr_order_number)
- Column store index: catalog_returns_cstore(all columns).

Customer_address (1M rows)

- Clustering index: pk_ca_address_sk(ca_address_sk)
- Column store index: customer_address_cstore (all columns)

Item (204,000 rows)

- Clustering index: pk_i_item_sk(i_item_sk)
- Column stored index: Item_cstore (all columns)

Date_dim (73049 rows):

- Clustering index: pk_d_date_sk(d_date_sk)
- Column store index: date_dim_cstore (all columns)

The experiments were run on a small commodity server with the following characteristics: Nehalem EP Xeon L5520 with two four-core processors for a total of eight cores running at 2.27GHz, hyper-threading off, 24G of memory, four 146G SAS drives in RAID0 configuration, each with read throughput in the range 100 to 125 MB/sec.

We ran the four queries in two ways: a) restricting the optimizer to use only row store indexes and b) allowing, but not forcing, the optimizer to use column store indexes. Each query was run twice, in isolation, first with a cold buffer pool and second with a warm buffer pool. The database is large enough that all the required data did not fit in memory. Tables 3 and 4 below show the observed elapsed times and total CPU times (in seconds) and the improvement when using column store indexes.

We will discuss the individual queries in more detail below but they can be briefly characterized as follows. Query 1 is a straightforward star-join query with one fact table and two dimension tables. Query 2 exemplifies a drill-down query with a very restrictive predicate where using a column store index provides little or no benefit. Query 3 is a narrow single-table

query with several expensive expressions. Query 4 is a complex query with a common table expression and a subquery.

As shown in tables 3 and 4, all but Q2 show an improvement of over 10X in elapsed time with a warm buffer pool. With a cold buffer pool the improvements in elapsed time are somewhat less but still very substantial.

Table 3: Observed query times (sec) with warm buffer pool

Warm	Row store only		Column store		Improvement	
Query	Elapsed	CPU	Elapsed	CPU	Elapsed	CPU
Q1	4.9	36.4	0.3	1.9	16.4X	19.3X
Q2	0.2	1.4	0.3	1.2	0.8X	1.2X
Q3	21.0	166.9	1.8	13.4	11.9X	12.5X
Q4	49.2	101.6	4.9	30.0	10.1X	3.4X

Table 4: Observed query times (sec) with cold buffer pool

Cold	Row store only		Column store		Improvement	
Query	Elapsed	CPU	Elapsed	CPU	Elapsed	CPU
Q1	19.7	35.6	1.6	2.0	12.3X	18.2X
Q2	1.7	0.9	1.3	1.3	1.3X	0.7X
Q3	55.1	168.4	8.5	13.7	6.5X	12.3X
Q4	55.5	102.3	7.2	20.5	7.7X	3.4X

4.2.1 Query one

This query is a typical OLAP query: a simple star-join query over three tables with catalog_sales as the fact table and two dimension tables, date_dim and item. The only selection predicate is on date_dim.

```
select i_brand, count(*)
from catalog_sales, date_dim, item
where cs_sold_date_sk = d_date_sk and
      cs_item_sk = i_item_sk and d_year > 2001
group by i_brand
```

The query plan selected by the optimizer when allowed to use column store indexes is shown in Figure 3. This plan was 16.4X faster with a warm buffer pool and 12.3X faster with a cold buffer pool than the best plan with row store indexes only. The query processed over 144M tuples in less than a third of a second.

The optimizer chose to use column store indexes for all three tables. Execution begins by scanning the two dimension tables and building hash tables. Next multiple threads scan different segments of catalog_sales column store index and probe the hash tables in parallel. Batched partial aggregation is then performed on each stream of joined tuples after which the tuples are redistributed on i_brand to be able to finish the aggregation. Final aggregation is completed in parallel on each stream and finally the result tuples are gathered into a single output stream.

The plan also illustrates how batch and row processing can be mixed in the same plan. The bulk of the processing is done in batch mode. Only the final processing, from repartition on, is done in row mode but by then only 5416 rows remain.

Evaluation of certain predicates can be pushed all the way down into the storage engine. This includes selections with bitmaps (Bloom filters). The predicate d_year > 2001 is pushed all the way down and reduces the stream from date_dim_cstore from 73K to 35.8K. While the hash table on date_dim is built, a bitmap on the join column d_date_sk is constructed. The input from catalog_sales_cstore is immediately filtered using the bitmap reducing it from 144M to 29.1M rows. The data is reduced in two ways: a) some column segments can be immediately eliminated based on segment metadata alone and b) in the remaining segments, rows with no match in the bitmap are eliminated.

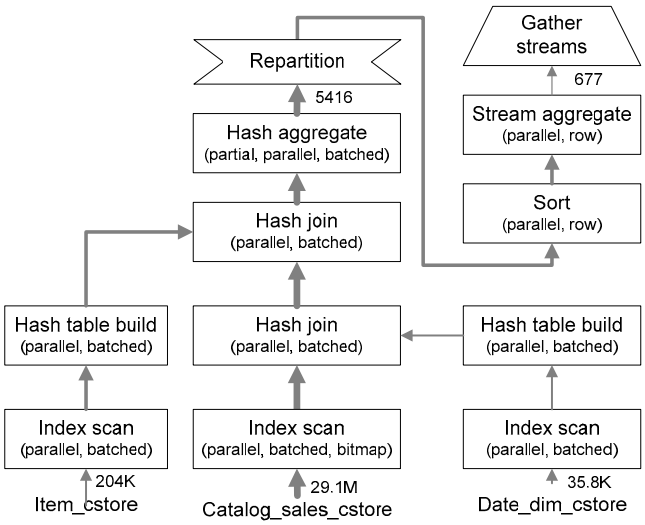

Figure 3: Q1 execution plan when using column store indexes

4.2.2 Query two

The next query is similar to Q1 but with a very restrictive selection predicate. It is an example of a drill-down query that an analyst might issue to get more detailed information. It illustrates the fact that the optimizer may choose a plan that includes both row store and column store indexes.

```
select i_brand, count(*)
from catalog_sales, date_dim, item
where cs_sold_date_sk = d_date_sk and
    cs_item_sk = i_item_sk and d_year = 2001 and
    d_moy = 11 and d_weekend = 'Y'
group by i_brand
```

With a warm buffer pool, the row store only plan is slightly faster but, with a cold buffer pool, it is slightly slower.

The optimizer selects all indexes based on estimated cost. When allowed to use column store indexes, the optimizer chose a mix of column store and row store indexes. Execution begins by scanning date_dim_cstore which produces only nine rows because of the highly restrictive predicate. The result is then joined with catalog_sales using a nested-loop join against the clustering index which is a row store index. This produces an output of 1.4M rows which is then preaggregated on cs_item_sk reducing it to 102K rows. The result is hash-joined with the item table using the column store index item_cstore and final aggregation is done using hash aggregation.

4.2.3 Query three

Q3 (on the next page) is an aggregation query over a single table where the aggregation contain somewhat more complex (and

expensive) expressions. Batched expression evaluation is much more efficient than evaluation one row at a time.

```
select cs_warehouse_sk,
 sum(cs_sales_price*(1-cs_ext_discount_amt) as s1,
 sum(cs_sales_price*(1-cs_ext_discount_amt)*
          (1 + cs_ext_tax)) as s2,
 avg(cs_quantity) as avg_qty,
 avg(cs_coupon_amt) as avg_coupon
from catalog_sales
where cs_sold_date_sk > 2450815+500 and
 cs_sold_date_sk < 2452654-500 and cs_quantity >=1
group by cs_warehouse_sk
order by cs_warehouse_sk;
```

With a warm buffer pool, the column store plan reduces the elapsed time from 21.0 sec to 1.8 sec and the CPU time from 166.9 sec to 13.4 sec. We will outline the main causes of this reduction in a moment.

The query plan using the column store index is shown in Figure 4. The selection predicates are all pushed into the storage engine so no filtering operator is required. The "compute scalar" operator computes the arithmetic expression used in the aggregates. Each parallel stream is then preaggregated, after which the streams are gathered for the final aggregation which is done in row mode.

The row-store only plan is exactly the same except it begins with a range scan of the clustering index. The predicate on cs_sold_-date_sk defines the range so only part of the index is scanned.

So if the plans are the same, what explains the huge difference in elapsed time and CPU time? The reduced elapsed time is caused partly by reduced CPU time and partly by reduced I/O time. The reduced I/O time is caused by two factors: scanning only a few columns and quick elimination of many segments based on metadata alone. The reduced CPU time is caused by the fact that the batched operators and evaluation of expressions are much more efficient. In batched expression evaluation, significant savings accrue from switching between the query execution component and the expression evaluation component once per batch instead of once per row. The average cost of arithmetic operations is reduced to around 10~20 cycles per operation.

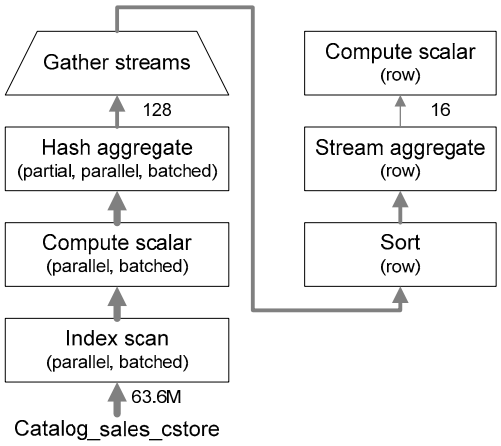

Figure 4: Q3 execution plan when using column store index

Query four is shown in Figure 5 below. It is a complex query with a common table expression and a subquery in the where-clause. It illustrates the fact that the full power of SQL Server's query processing is available; arbitrarily complex queries can make use of column store indexes.

The use of column store indexes reduced the elapsed time from 49.2 sec to 4.9 sec (with warm buffer pool) and CPU time from 101.6 sec to 30.0 sec.

The execution plan is too large to include here but the overall structure is relatively simple. The outer query joining customer_total_return, customer_address, and customer is computed entirely in batch mode using column store indexes. The subquery is then computed, with the joins done in batch mode but aggregation done in row mode. Next, the result of the subquery is joined with the result of the outer query. Up to this point, everything has been done in parallel. Finally, the different streams are gathered and the top operator evaluated. The plan processes a total of 87.9M rows and about 62% is done in batch mode. With around 1/3 of the rows processed in row mode, we can get at most 1/(1/3)=3 times improvement if the plan shape remains similar. We obtained 3.4 X CPU improvements but the improvement in

4.2.4 Query four

```
with customer_total_return as
 (select cr_returning_customer_sk as ctr_customer_sk, ca_state as ctr_state,
        sum(cr_return_amt_inc_tax) as ctr_total_return
 from catalog_returns, date_dim, customer_address
 where cr_returned_date_sk = d_date_sk and d_year >=2000
   and cr_returning_addr_sk = ca_address_sk
 group by cr_returning_customer_sk, ca_state )

select top 5  c_customer_id, c_salutation, c_first_name, c_last_name,
   ca_street_number, ca_street_name, ca_street_type, ca_suite_number,ca_city,
   ca_county, ca_state, ca_zip, ca_country, ca_gmt_offset, ca_location_type,
   ctr_total_return
from customer_total_return ctr1, customer_address, customer
where ctr1.ctr_total_return > (select avg(ctr_total_return)*1.2
                               from customer_total_return ctr2
                               where ctr1.ctr_state = ctr2.ctr_state)
  and ca_address_sk = c_current_addr_sk and ca_state = 'GA'
  and ctr1.ctr_customer_sk = c_customer_sk
order by c_customer_id, c_salutation, c_first_name, c_last_name, ca_street_number,
  ca_street_name,ca_street_type, ca_suite_number, ca_city, ca_county,
  ca_state, ca_zip, ca_country, ca_gmt_offset, ca_location_type, ctr_total_return;
```

Figure 5: Query 4

elapsed time is much higher. This is because the row-mode-only execution plans causes some intermediate results to spill to disk while the batch mode plan avoids this.

5. WORK IN PROGRESS

For reasons of scope and schedule, direct update and load of tables with column store indexes is not supported in the Denali release of SQL Server, but we still accommodate the need to modify and load data. In Denali, you can add data to tables in a number of ways. If the table is small enough, you can drop its column store index, perform updates, and then rebuild the index. Column store indexes fully support range partitioning. So for larger tables, you can use partitioning to load a staging table, index it with a column store index, and switch it in as the newest partition. This can handle many loads per day, allowing you to keep data current. We plan to support direct update in a future release.

In SQL Server, the primary organization of a table can be either a heap or a B-tree. Again, for reasons of scope and schedule, using a column store index as the primary organization is not supported in the initial release; they can only be used for secondary indexes. We plan to lift this restriction in a future release.

6. RELATED WORK

Storing data column-wise is a rather old idea. The concept of decomposing records into smaller subrecords and storing them in separate files goes back to the early 1970s. Hoffer and Severance [7] published a paper as early as 1975 on the optimal decomposition into subrecords. Batory [4] investigated how to compute queries agains such files in a paper from 1979. In a paper from 1985, Copeland and Khoshafian [5] studied fully decomposed storage where each column is stored in a separate file, that is, a column store.

Research on column stores then lay largely dormant for almost twenty years. The 2005 paper on C-Store by Stonebraker et al [9] and subsequent papers [1][2][3] [6] revived interest in column stores.

A considerable number of prototype and commercial systems relying on column-wise storage have been developed. References [4][5] include references to several prototype systems built during the 1970s.

Several commercial systems are available today. They are targeted for data warehousing and most are pure column stores, that is, data is stored only in column format. The earliest such systems are Sybase IQ [19] and MonetDB [16], which have been available for over a decade. Newer players include Vertica [20], Exasol [12], Paraccel [17], InfoBright [14] and SAND [18].

SQL Server is the first general-purpose database system to fully integrate column-wise storage and processing into the system. Ingres VectorWise [15] is a pure column store and engine embedded within Ingres but it does not appear to interoperate with the row-oriented Ingres engine, that is, a query cannot access data both in the VectorWise column store and the standard Ingres row store. Greenplum [13] and Aster Data [11] offer systems targeted for data warehousing that began as row stores but have now added column store capabilities. However, we have found no information on how deeply column-wise processing has been integrated into their engines.

7. ACKNOWLEDGMENTS

Many other people have contributed to the success of this project. We would especially like to thank Amir Netz for his many ideas and for challenging us, Cristian Petculescu for answering endless questions, Hanuma Kodavalla for initiating and nurturing the project, and Jose Blakeley and Mike Zwilling for their advice and continuing support of the project.

8. REFERENCES

[1] Abadi, D.J., Madden, S.R., and Ferreira, M.: Integrating compression and execution in column-oriented database systems. SIGMOD, 2006, 671-682.

[2] Abadi, D.J., Myers, D.S., DeWitt, D.J., and Madden, S.R.: Materialization strategies in a column-oriented DBMS. ICDE, 2007, 466-475.

[3] Abadi, D.J., Madden, S.R., and Hachem, N.: Column-stores vs. row-stores: how different are they really? SIGMOD, 2008, 981-992.

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

[5] Copeland, G.P., Khoshafian, S.N.: A Decomposition Storage Model. In Proc. SIGMOD, 1985, 268-279.

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

[7] Jeffrey A. Hoffer, Dennis G. Severance: The Use of Cluster Analysis in Physical Data Base Design. VLDB 1975: 69-86

[8] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. ICDE, 2001, 567-574.

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

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

[11] Aster Data, http://www.asterdata.com

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

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

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

[15] Ingres VectorWise, http://www.ingres.com/products/vectorwise

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

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

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

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

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