Leveraging Compression in the Tableau Data Engine

Richard Wesley Pawel Terlecki
Tableau Software
837 North 34th Street, Suite 200
Seattle, WA 98103, USA
+01-206-633-3400
{hawkfish,pterlecki}@tableausoftware.com

ABSTRACT

Data sets are growing rapidly and there is an attendant need for tools that facilitate human analysis of them in a timely manner. To help meet this need, column-oriented databases (or "column stores") have come into wide use because of their low latency on analytic workloads. Column stores use a number of techniques to produce these dramatic performance techniques, including the ability to perform operations directly on compressed data.

In this paper, we describe how the Tableau Data Engine (an internally developed column store) leverages a number of compression techniques to improve query performance. The approach is simpler than existing systems for operating on compressed data and more unified, removing the necessity for custom data access mechanisms. The approach also uses some novel metadata extraction techniques to improve the choices made by the system's run-time optimizer.

1. INTRODUCTION

Recent years have seen an unprecedented growth in the amount of data available for analysis. This data often needs to be processed manually by a human who understands the semantics of the problem domain. Once the initial analysis has been completed, the results are often passed on to other interested parties. Visual analysis tools such as Tableau [1] enable the creation of such analyses via an intuitive drag-and-drop interface.

In the early days of data analytics, these results were generally static reports designed to communicate the end result of the analysis to decision makers. Increasingly, however, such analyses are becoming the starting point for further work. More recent incarnations of these tools have added an emphasis on interactive visualization and computer-mediated analytic narratives.

A low-latency analytic query engine best serves such interactive tools, but the underlying data source may have higher latency, either due to architectural issues or heavy workloads. The Tableau Data Engine (TDE) acts as a low-latency relational store for visual analysis in the Tableau system by storing and processing extracts of a data set under analysis.

SIGMOD'14, June 22–27, 2014, Snowbird, Utah, USA.
Copyright © 2014 ACM 978-1-4503-2376-5/14/06…$15.00.
http://dx.doi.org/10.1145/2588555.2595639

As a commercial desktop product, Tableau's lack of control over the deployment environment hardware makes scale-out of the TDE impossible in many use cases. While we have managed to incorporate some single-node parallelism into the execution core, the machines users employ for analytics are often laptops with limited energy, disk and CPU resources. This constraint has given us a bias towards gaining performance through algorithmic techniques. The focus of the current work is to describe several such algorithmic approaches centered around operating directly on compressed data.

One important component of many low-latency relational stores is the ability to operate directly on compressed data [12]. In this paper, we will describe how the TDE leverages various kinds of compression during query compilation and execution. The approaches we will describe include:

- New data formats that enable modifying the semantics of entire columns independent of the number of rows;
- Two methods of expressing decompression as a join operation, one of which we believe to be novel;
- A technique for extracting metadata during query execution for use by a tactical optimizer.

We will also combine these techniques with a flat file import operator to demonstrate how the system can generate good physical designs during the import process at low cost.

The organization of this paper is as follows. Section 2 covers some background material and related work. In Sect. 3 we describe a set of lightweight compression techniques that are used to quickly improve the physical design of the imported data set. Section 4 presents our optimizer-based decompression system. Section 5 contains our evaluation setup, including our flat file input system, and we give the results of our evaluation in Section 6. We describe our conclusions in Section 7, and Section 8 sketches avenues for future work.

2. BACKGROUND

In this section, we summarise some previous work on the use of column stores for analytic workloads [2,6,10,11] in order to provide the context for the rest of the paper.

2.1 Lightweight Compression

Lightweight compression is a staple of column store physical storage layers and has been described by [5,6] and others. Examples of lightweight compression include *run-length encoding* and *delta encoding*. These compression techniques typically work on data of fixed-width types and are often *symmetrical* in that compression and decompression have similar computational complexity. They are *lightweight* in that they require less computation time than reading from secondary storage, or even from main memory.

Lightweight compression algorithms are common in column stores because the heterogeneous data layout inherent to columnar layouts make such algorithms easy to apply. The data storage used by the algorithms typically includes actual values from the column, which makes it relatively simple to extract the values and use them directly in query processing instead of decompressing the data to its full size. Because of the relatively high performance of the compression algorithms, column stores can compress data during execution to save I/O bandwidth when spilling results to disk.

2.2 Extracts

Tableau is a visual analysis tool based on the Polaris system [1], which allows the user to create analytic queries and narratives using a simple drag-and-drop interface.

For input, Tableau connects natively to a wide range of databases, both relational and hierarchical. In addition to supporting a large number of commercial and open source databases, it also connects to *extracts* of data, which are subset of the original data set that may have been filtered, sampled or rolled up.

These extracts are used in various workflow scenarios, including off-line work, reducing the load on data warehouses, sharing of data with third parties, filtering/projecting subsets of the data, pre-aggregating the data and supplementing databases that either perform poorly or lack useful functionality such as `COUNTD` or `MEDIAN` aggregation.

2.3 Data Engine

Tableau extracts were originally created using the Firebird [14] open source relational engine, but performance considerations eventually led to its replacement with an in-house component, the Tableau Data Engine (TDE).

We first described the TDE in [10]. It is a read-only column store that has been optimized for use with the Tableau visualization environment. We chose to create our own component because of business requirements that could not be met by any existing commercial, academic or open source system, including: collated strings, single file databases, 32-bit hardware, calculation language semantics and Tableau’s NULL join semantics.

We now provide a brief overview of the salient parts of the TDE architecture to provide background for the main discussion.

2.3.1 Query Plan Generation

The TDE expresses a query plan as a block-iterated Volcano-style [8] operator tree with two styles of operators: flow operators process a block of rows at a time before passing the block on to the next operator; stop-and-go operators must read all the blocks of their input before their output is available. The query processor follows the optimization model of [3].

In the first *strategic* phase, the shape of the optimal plan is determined. A rule-based component derives properties for all tree nodes based on metadata and performs transformations, such as elimination of common sub-expressions, computation and filtering move-around, parallelism injection and expression simplification.

The second part of optimization, *tactical*, is delayed until run-time, where decisions can be made based on the actual data. This time property derivation happens on-the-go and can be more accurate. In particular, we track minimum/maximum value or cardinality or nullability. While the arrangement of operators is not affected, their implementation can be optimized based on specific input properties. For example, an aggregation operator can choose a hash algorithm based on the sizes and other attributes of the aggregation keys.

2.3.2 Compression

For historical and structural reasons, the TDE storage layer makes a distinction between *compression* and *encoding* of columns. Compression in this parlance is traditional dictionary compression with each column owning an associated dictionary that can contain either fixed width data (*array* compression) or variable width data (*heap* compression). The main data column is always fixed width and consists of either uncompressed scalars, indexes into the fixed width dictionary or offsets into the heap dictionary. This architecture allows the query optimizer to reason about compression and to optimize computation on compressed data by using *invisible joins* [5]

A second form of compression was included in the original TDE design, which operated only on fixed-width data. These forms of compression are called *encodings*. Encodings are an abstraction that externally appear as a paged array of fixed width values, but are stored internally in a more compressed format. Encodings are concealed from the rest of the system behind virtual interfaces that present a paged interface to an ordered stream of bytes. Encoding and decoding is implemented behind these interfaces during insertion and byte range requests. Encodings are semantically neutral in that they do not know the type of the underlying data, only its width. In the first release, the TDE only implemented run-length encoding.

All compression and encoding is performed independently at the column level, so there are no global tables to update.

2.3.3 Storage Constraints

One important usage requirement for a TDE database is that the user should be able to choose it in a file selection dialog, i.e. the database needs to be represented by a single file. While this restriction does not affect functionality of extracts, due to their read-only nature, it adds a significant I/O burden of copying the read-write internal format based on multiple column files into a a single file. Compression applied at the column level helps reduce the total size and, thus, the cost of making this unavoidable copy.

2.3.4 Hashing and Comparing

While the TDE's strategic optimizer determines the structure of the plan, the tactical optimizer makes run-time decisions based on the actual data being processed. One of its tasks is to choose algorithms for hashing and comparing used by several operators, including joins and aggregation. String comparison performance is greatly improved by having sorted string heaps whose tokens can be directly compared instead of comparing string contents. This performance benefit is especially important because unlike many column stores (which only offer simple binary collation) the TDE must implement locale-sensitive collations, which are even more time consuming to compute. Hashing of strings must also be performed in a locale-sensitive manner, and imposes a similar computational burden.

Even with unique string tokens, hashing performance is based on the width of the data being grouped or joined. A width of 1-2 bytes allows the TDE to use direct hashing with a small 64K-element lookup table. With a width of 3-4 bytes, a perfect hash function can be constructed, but wider data requires expensive collision detection. Minimising data width is, therefore, an important physical design goal for TDE columns.

The TDE has more leeway in type design than typical relational stores because Tableau itself does not model data types very precisely. In fact, Tableau only has Boolean, integer, real, date, timestamp and locale-sensitive string types. This means, for example, that the TDE can use any representation it likes for a column that Tableau considers an integer. This type design flexibility can in turn be used to improve hash performance.

2.3.5 Fetch Joins

The TDE can also use *fetch joins* [3] for many-to-one joins if there is a single join column and the row id of the inner table is an affine transformation of the column value. This type of join requires no intermediate lookup tables and is the fastest join available. Detecting when it can be applied improves join performance significantly. This situation happens most often in primary-key/foreign-key joins, and especially in decompression of scalar dimensions via invisible joins.

3. ENCODINGS

In the first TDE paper [10] we listed creating new encodings as a topic for future work. After a review of the literature, we settled on several simple encodings that are compatible with the existing, non-segmented storage model used by the TDE. Our goal was to reduce storage requirements for user extracts, but we discovered a number of unexpected benefits along the way, which we will now describe.

3.1 Encoding Formats

While the encodings being described here are well known in the literature, we would like to quickly explain the storage format of each one to facilitate later discussion. This header has been carefully designed so that some simple header manipulations can lead to semantically interesting column-level changes.

Each bit-packed stream starts with a header of the general form shown in Figure 1 followed by blocks of bit-packed values. The bit-packed values are treated as unsigned values by the encodings.

The first 8 header bytes cache the logical size of the stream to make stream length queries perform well and to handle situations where the physical size of the packed data is larger than the logical size. This happens frequently with bit-packing schemes, because bit fields must fit into an exact number of bytes.

The second 8 header bytes contain the offset to the bit-packed data. This allows the header to be resized without disturbing the bit packing.

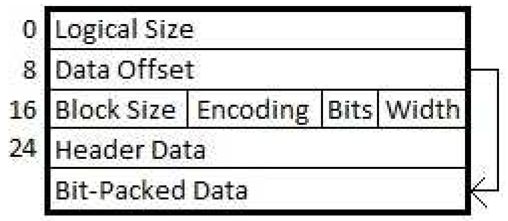

Figure 1. Bit-Packed Header Format.

The third 8 header bytes contain the decompression block size (i.e. the number of values in the block), the encoding algorithm, the width of the data stream elements and the number of packing bits. The block size is a multiple of 32 to ensure that the bit packing ends on a byte boundary. A decompression block may also contain header information such as a running total. Each physical stream only contains complete decompression blocks (hence the need for the logical size field). The number of values in a decompression block is typically the same as the block iteration size of the query execution system so that one decompression call is needed per iteration block.

The rest of the header contains encoding-specific data, which we will now describe.

3.1.1 Frame-of-Reference

The frame-of-reference header contains 8 bytes to hold the frame value. The bit-packed values are added to this frame value to produce the uncompressed values.

3.1.2 Delta

The delta-encoding header also contains 8 bytes to hold the minimum delta value. The bit field values are added to this delta value to obtain the next value. Each encoding block starts with the running total for that block so the data can be accessed randomly as well as sequentially.

3.1.3 Dictionary

The dictionary-encoding header starts with 8 bytes containing the number of dictionary entries, followed by enough space to contain 2^bits dictionary entries. This allows the dictionary to grow up to the 2^bits limit. Dictionary encoding is limited to 2^15 values to keep the dictionary in cache and make the compression cuckoo hash table implementation simple and fast.

3.1.4 Affine

Affine data streams are a simplified form of delta encoding where the bit width is *zero* or (equivalently) where the delta value is constant. Each value can be simply computed as *value = base + row * delta*.

The affine-encoding header contains 16 bytes to hold two signed integers for the base and delta values. Again, the header reserves 8 bytes for both values even if the actual values are narrower. Affine encoding does not require any bit-packed values and sets the bit count to 0.

Affine encoding is similar to the virtual column representation in MonetDB [3] except that the values are physically expanded a block at a time instead of being computed in line. The advantage of treating affine encoding uniformly with the other encodings is that its applicability can be detected during the compression stage e.g. when a column contains sequential values.

3.1.5 Run-Length

Run-length-encoding headers have a different format that consists of a header followed by runs represented by length/value pairs. The header contains the logical size of the stream and two bytes that contain the width of the two fields. These values are fixed for the entire stream.

3.2 Dynamic Encoding

The MonetDB/X100 system described in [6,9] demonstrates that lightweight compression can be used during query execution because the compression routines are computationally cheap. We have not implemented the patched variants of these encodings because they require a segmented storage model. We needed another mechanism to solve the problem of inserting values that lie outside the range currently supported by the column encoding.

To accomplish this, we continually track statistics for a column as values are inserted. These statistics are simple to gather, consisting mostly of the value range and delta range. At any given point, we can quickly determine the best of the available choices of encoding for the column. We dynamically encode the columns one block at a time, using the block values for a column to update the column's statistics before inserting the data block into the column's encoding stream. If the column insert fails (e.g. due to representation limitations), we can consult the column's statistics and choose a new encoding. When all rows have been processed, we can also compare the current encoding with the optimal one and convert to this optimal format if desired. In practice, we found that the encoding stabilizes quite quickly: Encoding the TPC-H [TPCH] `lineitem` table at SF 1 made only two encoding changes and the rewrites still performed less disk IO than writing the unencoded column.

This technique is vulnerable to data sequences that can cause constant re-encoding, but we have not investigated situations that might trigger this problem. One approach would be to detect excessive reformatting and fall back to unencoded data until the end. At that point, we could consult the final statistics and decide whether it is worth encoding the column or leaving it unencoded as the I/O cost has already been paid.

3.3 FlowTable

Applying encodings require a full scan of each column, therefore, this functionality needs to be expressed in a plan by a stop-and-go operator. In the MonetDB/X100 project [6], encoding happens as part of the `Save` operator, which writes a table back into the store. Later on, such a table can be scanned and decoding will happen behind the scenes.

In order to leverage encodings in the TDE, we extended the existing `FlowTable` operator, whose task is to turn a stream of row blocks into a table. Note that encoding of each column is *independent*, therefore, the computations can be distributed across the available cores. This allows more processing power to be substituted for memory and I/O bandwidth.

3.4 Encoding Manipulations

Once a column has been encoded, there are some fast manipulations that can be performed on the header to change the type of the data, or create a sorted dictionary for a compressed column. These manipulations can be advantageous for downstream processing by reducing the size and complexity of the data. The speed of these transforms is a consequence of the formats used in the encoding headers as described above. The encoding statistics can also be mined to determine useful column level metadata. The `FlowTable` operator applies these manipulations as a post-processing step during its build step.

The unifying principle for all these manipulations is that lightweight compression makes it easy to transform the entire compressed data set in semantically meaningful ways. We will now describe these manipulations in more detail.

3.4.1 Type Narrowing

The headers for frame-of-reference, dictionary and affine encodings can all be modified to change the width of the data. For example, if the column is a 4-byte integer column under frame-of-reference encoding, we can use the bit width and the base value to detect when the values can be represented by a 2-byte integer. The header can then be edited to update the size, width and base fields. The offset to the bit field data does not need to change because the offset is stored in the header.

These operations can be accomplished in O(1) time for all three of these encodings except dictionary encoding, where the operation cost is proportional to the number of entries i.e. O(2^bits). Note that these run times are *independent* of the size of the column.

Delta encoding embeds the running totals in each block and run-length encoding contains values in each pair, so these encodings are not amenable to this type of header manipulation. It is possible, however, to decompose a run-length encoded column into a *value* stream and a *count* stream, perform the narrowing operation on the extracted value stream and then rebuild a run-length stream with the original counts and the new values.

If a narrower type can represent a column, then the downstream operators may be able to produce a better hash function that does not have collisions for aggregation or joins. It also means that later computations may produce smaller results, reducing the memory, disk and network footprint of the system.

3.4.2 Metadata Extraction

Encoding statistics can help cheaply derive properties of the underlying data and, thus, enable tactical optimizations. Delta-encoding, for example, can indicate whether a column is sorted. Detection of sorted columns can be used to improve the efficiency of downstream operations such as aggregation, joins and sorting.

Affine encodings can be checked to see if the delta is 1. If so, the column is not only sorted, but also *dense* and *unique*, which may enable *fetch joins* downstream. Filtering the inner table of a join can mask the applicability of a fetch join, because the filter will remove an existing *dense* attribute of a column (because it may no longer be valid). If `FlowTable` is used to build the inner join table after the filter has been applied, its encoding component can detect the situation where the filter leaves a contiguous sub-range of the data and reassert the *dense* property, allowing a fetch join to be generated. This situation is common with date columns, which are often compressed and filtered to a range and then joined back into the main query.

The encoding statistics can also be analysed to determine the cardinality of the column domain, the maximum and minimum value of the column and – because the TDE uses sentinel values for NULL – whether the column contains NULLs. These properties can be used by downstream operators to make tactical optimizations, or reported back to Tableau. Tableau can in turn use this metadata to drive choices in the UI such as whether to represent domain values using colors, shapes or other mark types.

3.4.3 Encoding Becomes Compression

Accelerators [3] for string heaps help make small string heaps *distinct*, but an even better outcome is to have a sorted heap, which means that its tokens are directly *comparable*. The accelerator also reduces the number of tokens in use, and if it succeeds in keeping the token count low enough, this reduction will result in dictionary *encoding* of the tokens.

In this situation, the dictionary encoding entries are the set of distinct tokens for the strings. Since the number of strings is small, we can sort them in a relatively short period of time. The new tokens take up the same amount of space as the original tokens and can be written back out to the dictionary-encoding header, with the result that the column now has both comparable and distinct tokens. Combined with type narrowing, this allows us to optimize the representation of an intermediate computed string column in time proportional to the domain size and avoid touching the actual rows of the column – which can be arbitrarily many.

A dictionary-encoded scalar column can be converted into a dictionary-compressed column by copying the encoding dictionary (which is just an array of scalars) into a compression dictionary. The original encoding dictionary is then replaced with the compression tokens (again, narrowing them if desired) and the column is now a dictionary-compressed column with minimal width. This can be valuable for scalar dimensions such as dates, which have relatively few values, but expensive calculations (such as extracting the month). Converting the column into a dictionary-compressed column enables *invisible joins* on the data if the containing table is written out as part of an import process.

A similar transformation can be performed with frame-of-reference encoding, with the caveat that the compression dictionary may contain values that are not actually in the column. The frame value and the bit width determine the outer envelope of integer values that are present in the column. If the underlying type has the same bit ordering semantics as the signed integers used by the encoding process, then a *sorted* scalar dictionary can be generated from the base value and the number of bits in the representation. The header can then be modified to contain the unsigned tokens as indexes into this scalar dictionary. Because the frame range defined an outer bound, we have no guarantee that all dictionary values are actually present in the column, but this technique looks promising for compressing date and timestamp columns, and may be the topic of future work.

We have not implemented this encoding to scalar dictionary compression in the `FlowTable` operator because the existence of non-string dictionaries is something that the query compiler needs to be aware of. This technique can, however, be employed by the more heavyweight operators (like `AlterColumn`) used during the TDE's global optimization phase to reduce the run time of that operation. `AlterColumn` can also apply the run-length decomposition technique described in Sect.3.4.1 to generate dictionary compression columns from the value stream, greatly reducing the optimization cost. This results in a scalar dictionary compressed column with a run-length encoded token stream.

4. DECOMPRESSION JOINS

Strategic query optimizers are typically oblivious about details of the storage layer, such as data compression. That ensures clear boundaries of the component but may also limit application of optimizations dependent on low-level properties of data. For this reason certain storage concepts are at times modeled for the optimizer to use.

Plan costing is one good example. Row sizes or average costs of reads or writes of data stored in different layouts are made available to allow for more precise estimations. Also, non-standard storage concepts can be expressed by special types of columns, such as the sparse column set in Microsoft SQL Server used to model the interpreted storage [13].

Below, we express decompression of data values in a manner that permits the query optimizer to reason about it using established query optimization techniques. In this model, compressed columns are expanded using joins against special kinds of tables.

4.1 Dictionary Tables

4.1.1 Invisible Joins

Dictionary *compressed* columns (that is, columns with a secondary heap, as distinct from the dictionary *encoded* columns of Sect. 3.1.3) can be introduced to the query plan using an operator called `DictionaryTable`. This table operator has a column of the same type as the original, but the column *data* has been replaced with the set of unique tokens in heap order. For variable width data (e.g. strings) this column is the only one in the table and it has a copy of the original column's heap. For fixed width data, the token column does not have a heap but the table itself has a second column, which is simply a copy of the original column's fixed-width heap.

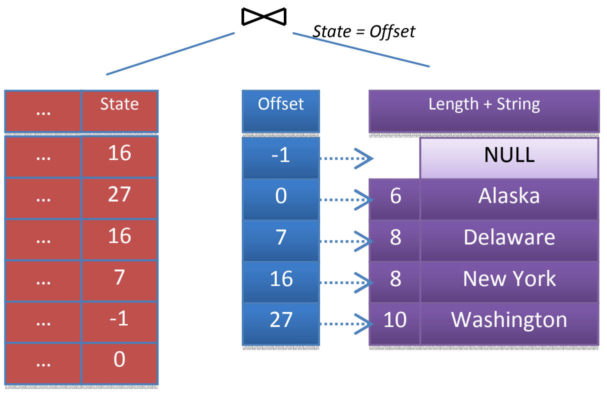

Figure 2. Invisible join for a string column.

Expansion of this column can now be expressed as a foreign key join between the main table and the token column in the `DictionaryTable` (see Fig. 2). Further, the strategic optimizer can rearrange the plan by pushing filters and computations on the column values down to the inner side of this foreign key join. In consequence, computations on the compressed data get expressed quite naturally as part of a traditional query plan without having to widen the inter-operator interfaces.

4.1.2 *Tactical Optimization*

Using the encoding manipulations above, there is now an opportunity for the run-time (or *tactical*) optimizer to take advantage of compression. The TDE `Join` operator takes a stop-and-go operator as the inner relation, so once the plan contains flow operators (like `Select` and `Project`) on the inner side of these expansion joins, the flow needs to be materialized, usually by employing the `FlowTable` operator. `FlowTable` now extracts metadata during its processing, including the kinds of metadata used by the tactical optimizer to decide upon the join algorithm.

Consider the common situation where a date column has been dictionary compressed and a range predicate has been applied. Assuming the date column heap has been sorted, the range predicate will produce a dense range of token values from the `DictionaryTable`, which `FlowTable` can now detect. The `Join` operator can in turn use this information to choose an efficient fetch join instead of some form of hashing that requires an additional table lookup per row.

Next, consider the situation of a string column containing URL requests and a calculation to extract the file extension of the request. This will produce a relatively small number of strings on the inner side of the join, but the string function library is probably unable to estimate the resulting domain ahead of time. The computation therefore produces a column with wide tokens and an unsorted heap. `FlowTable` can now sort this small string table quickly and minimize the width of the token data. If the query then aggregates on this computation (e.g. counting the number of requests for each file type) the aggregation operator will be able to use a faster hashing algorithm thanks to the narrower representation.

4.2 Index Tables

4.2.1 *Rank Joins*

A similar "special table" technique can be used to expose run-length encoded columns to the strategic optimizer. The table is called an `IndexTable` and consists of three columns: The *value*, the *count* and the *start*. The first two are extracted directly from the column data and the start values are computed as the running total of the count values.

This `IndexTable` can be joined to the main table as before with `DictionaryTable`, but in this case, the join is not an equi-join. Instead, the join condition is a range predicate:

Index.start <= Outer.rank < Index.start + Index.count

Once again, now that we have expressed the decompression as a join, we can push single column arguments down to the inner table (see Fig. 3). Predicates and computations will now be evaluated on the compressed data, with significant performance gains.

Because the join is on the rank column, we have implemented a new type of join operator called `IndexedScan` that translates the range specifications directly into disk accesses. `IndexedScan` will access the outer table in the order given by

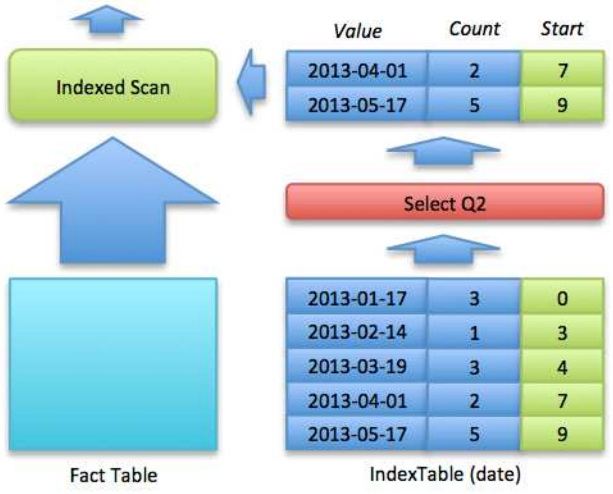

Figure 3. Predicate push-down on a Rank Join.

the inner table. This allows us to express range skipping simply as a join in the query plan. As we also use `FlowTable` to produce the inner table, we can apply the same metadata extraction and type minimisation techniques to any computed columns and present this enhanced metadata to the client of the `IndexedScan`.

4.2.2 *Ordered Retrieval*

`IndexedScan` also allows us to implement ordered retrieval of out-of-order run-length values by sorting the index on the value column. This access pattern enables us to use ordered (or sandwiched [Sandwich]) aggregation on columns that are not the primary sort key. This technique must be used with care because if the runs are too small, performance will be degraded, as we will show in our results.

4.3 Other Optimization Concerns

Besides the benefits of new optimizations that one can apply thanks to the nature of some encodings, the fact that data is compressed might have detrimental impact on the plan performance if other rewrites are not applied carefully.

`FlowTable` is commonly used as the right hand (or inner) side of a join operator. Hash joins usually exhibit random access patterns against this inner table, and not all of our encoding types have good random access performance. In particular, seeking backwards in our run-length encoded columns requires a sequential scan from the start of the data stream. Therefore, during strategic optimization we restrict encoding choices for the `FlowTable` nodes on the inner side.

Furthermore, the quality of encodings is sensitive to the order of raw data. Therefore, operators that disturb order, such as sorting or exchange, might affect how well data will be encoded down the stream and potentially increase I/O.

To give a more concrete example, let us consider a simple plan in which we read a column of dates from disk and apply a filter to it. Moreover, during extraction the column got dictionary-compressed and the resulting column *Date* in the dictionary got delta-encoded. Note that the tokens follow the original order of the dates and thus, the achieved encoding is efficient.

Since we are dealing with an invisible join scenario between the denormalized table and the dictionary, the filter can be pushed down to the dictionary. The values from the dictionary column *Date* need to be decoded first to evaluate the filtering predicate. Then, the results have to be materialized using `FlowTable` to build an inner side of a hash-join. The final encoding is likely to be of similar quality, because the filter removed a subset of data but did not change the order of values. However, if one injects exchange operators to parallelize the filter, the order of blocks of values gets disturbed and the resulting encoding might be much worse and lead to a physically larger column.

We identify situations of this kind and force the exchange operator to use order-preserving routing, i.e. number the blocks and output them in order [8]. Our benchmarks showed a relatively low, 10-15% overhead, associated with this additional constraint.

5. EXPERIMENTAL DESIGN

We performed two sets of experiments to assess the performance of our new features. The first set of experiments involves using a high-performance flat-file parsing operator to drive our new dynamic encoder, and measuring compression and metadata extraction performance. The second set of experiments measure the performance of pushed down filter predicates with indexed scans on an artificial run-length encoded data set.

We do not measure the performance of the dictionary expansion as that was evaluated in the original TDE paper [10].

5.1 TextScan

As part of our experimental setup, we used a text-parsing operator called `TextScan` to produce a stream of uncompressed columns with little metadata. `TextScan` is a flow type operator, which reads from a memory mapped byte stream and produces blocks of typed data. It attempts to perform type and column name inference if the schema is not provided, which can further reduce the need for user intervention, but it can also be given a schema if one is available. The development of this operator produced some interesting experimental results of its own, which we now describe.

5.1.1 Initial Approach

The first implementation of `TextScan` was designed to evaluate the text cracking process described in [7]. The operator was originally given a text file and a list of columns to parse. The unparsed columns would just be cracked into separate text files for later parsing. The file is assumed to be UTF-8.

The first step in the parsing process is to determine the field and record boundaries. A sample set of rows is tokenized using a given record separator (which defaults to end-of-line). Simple statistical analysis is used to determine the field separator.

Once the field boundaries have been determined, the columns must be typed. A sample block of rows is selected and typed by comparing the results of parsers for each data type to see which produced the fewest errors. The winning type parser is then used for the eventual scan of the entire text file. The parsers are then applied to the first row and if there were no errors, it was presumed that the flat file did not contain a header row and all values were treated as data. If there were errors, then the values were taken to be the column names. The schema and header row information can also be specified as inputs to the parsing system.

5.1.2 Parallel Parsing

Because these column parsers were producing independent output from a shared read-only state, it was a simple matter to run them in parallel on each block of rows. We were surprised to find that the performance *degraded* by at least an order of magnitude under parallel execution. Profiling the code showed that the problem was that the native parsers attached to each type object in the TDE's extensible type system were using the C++ standard library to parse the fields. The standard library is locale sensitive and each stream parse first needed to obtain and lock a singleton locale object. The lock contention for this object completely negated any gains from parallelism.

5.1.3 Scalar Cracking

To avoid this problem, we wrote buffer-oriented parsers for all the different types in the system. These parsers are tightly written C code and rely on no external state. With these in place, we found that parsing the scalar columns in the TPC-H `lineitem` table at SF-30 [4] on a four-core machine was comparable to the disk read bandwidth. This suggested that scalar parsing did not need to be deferred because it could be performed while waiting for the disk.

5.1.4 String Cracking

This discovery naturally led us to ask whether string parsing could be done at the same speed. To create a baseline, we first added a "compression" style where split strings were simply written to a text file with quotation marks and end-of-line separators. This is approximately the same amount of I/O as used for writing a string heap element consisting of a 4-byte header followed by the character data. The corresponding data streams consisted of a series of ascending offset tokens whose deltas were equal to the length of the string itself. If these tokens were fed into a delta encoder, the resulting data stream could be highly compressed. For example, the `l_comment` column has a maximum string width of 140 characters, so an 8-bit delta could be used, adding about 1 byte per row to the data.

Creating a string column in the TDE makes optional use of a heap accelerator object, which maintains a hash table of all strings that have been seen so far. This allows us to minimize the size of the heap for columns with small (< 2^31) numbers of strings and also ensures that such string columns have distinct tokens. Profiling of the import process shows that maintaining this hash table appears to be an execution "hot spot" when the number of strings is small, but performance of the entire import process does not appear to be affected thanks to the gains in reduced disk IO. The accelerator gives up on hashing once the number of heap elements passes the 2^31 element threshold.

5.2 Text Data

We conducted a number of experiments to judge the performance and other properties of the combined `TextScan` / `FlowTable` system. For these experiments, we used the output of the TPC-H `dbgen` tool [4], at both SF-1 and SF-30. We also used a 25GB text version of a 67M row internal testing database of containing ten years of FAA on time flight data ("Flights").

The only typical import operations that were not performed were sorting on a preferred attribute (e.g. `l_orderdate`) and applying dictionary compression to date scalars. Sorting is expensive but can sometimes help filtering and aggregation performance. Dictionary compression of dates can improve the performance of certain date calculations (e.g. month extraction) by performing the calculation on the date domain and joining the results to the main query via an invisible join. Both of these operations could be performed as a further design optimization if the workload suggests it.

Execution speed was only measured for the two large tables (both labeled "Large Tables" in the Figures.) Both of these files are larger than the disk cache of the test machine, so there were no cache "warm up" issues. The most important difference between the two files is that Flights does not have a large random string column like `l_comment`, but this is more typical of the data sets actually analysed by our customers.

The smaller tables from TPC-H SF-1 were used to demonstrate the efficacy of the metadata extraction process. They are labeled "SF-1 Tables" in the combined Figures.

5.3 Run Length Data

To evaluate the performance of indexed scans, we created artificial tables containing two run-length encoded columns (called *primary* and *secondary*). The columns consist of uniformly distributed random values in the range [0,100), and the tables were sorted ascending on both columns. We prepared two tables, one with 1 million rows and one with 1 billion rows and ran aggregation queries of varying selectivity against this data. Because of the size of the data sets, both columns were run-length encoded and the entire data sets easily fit in main memory.

6. EXPERIMENTAL EVALUATION

6.1 Parsing Performance

We ran the import process for both SF-30 `lineitem` and Flights 5 times on an Intel® Xeon® E5620 single chip machine running Windows 7 and averaged the run times. We measured times for:

- Disk bandwidth (summing all the bytes of the text file)
- Tokenizing the data (finding field boundaries)
- Splitting the file into column files, but not parsing
- Parsing scalars only (numbers and dates)
- Parsing all columns.

Where applicable, we also ran the tests with heap acceleration on and off, as well as with encodings on and off. The results are displayed in Figure 2.

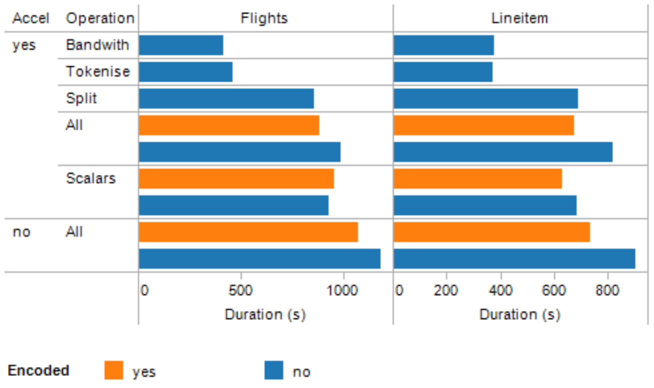

Figure 4. Split Time versus Compression Time.

By comparing the encoded and un-encoded results for the "All" and "Scalars" scenarios in Fig. 4, we can see that in all situations were it was applicable, the system performance with encoding turned on was either comparable to or superior to the performance without encoding. This result holds whether or not heap acceleration is in use.

Moreover, by comparing the adjacent "Split" and "All" bars, we can see that the system performance with both encoding and acceleration was comparable to simply splitting the flat file into separate text columns for later parsing, which shows that there was no benefit to delayed parsing on these data sets. The scalar only parsing also splits the strings for later parsing, which similarly appears to provide no additional benefit here.

Encoding and acceleration provide other important benefits beyond eliminating deferred parsing costs, which we will now quantify.

6.2 Storage

Our original TPC-H SF-1 database from [10] was about 660MB. Applying the new encodings to the columns reduces the size of the database by about 140MB.

We did not have a version 1 database for TPC-H SF-30, but we show the logical and physical sizes of the `lineitem` table for all combinations of encoding and heap acceleration in Fig. 5. The total disk savings from the original 26GB flat file is 22GB (84%) and the savings from the logical size (i.e. the un-encoded size) is 7.5GB (63%).

The version 1 Flights database with only run-length encoding and dictionary compression was 4.1GB. Figure 5 also shows the logical and physical sizes of this table. The total disk savings from the original 25GB flat file is 21GB (84%) and the savings from the logical size is 15GB (85%).

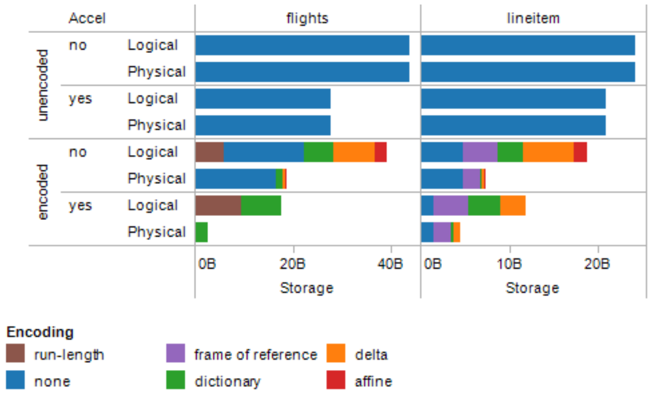

Figure 5. Compression Savings.

The top section of Fig. 5 shows the effects of heap acceleration without encoding. The effects are more pronounced for Flights because all of its string columns have relatively small domains. By contrast, `lineitem` consists mostly of `l_comment`, which is too large for the accelerator to compress effectively. The accelerator is designed to be small and fast for common usage, but is not designed to scale and it is doubtful that the 33% disk savings that would result would be worth the extra IO caused by heap collision comparisons.

Moreover, Fig. 5 breaks down the contribution of each type of encoding to the total savings. This shows that artificiality of the TPC-H data provides a number of opportunities for affine encoding. One notable example is the `c_customername` column, which consists of a set of unique strings all with the same length. Each string takes up the same amount of space in the heap so the tokens are equally spaced, which the system notices and encodes accordingly. For systems that store fixed width `CHAR(N)` style strings, this could be an important source of space savings when such strings are unique, because affine encoding has constant storage requirements.

The benefits of reduced storage footprint extend throughout the entire storage hierarchy. At one end, compression effectively increases memory bandwidth [6] by trading off CPU resources for memory latency. At the other end, smaller storage requirements reduce network transfer latency for data set upload and download.

6.3 Heap Sorting

Another operation we were able to perform during the parsing stage at no discernable performance cost was the sorting of string heaps when the column was dictionary encoded. Since TDE string heap tokens are not dense (being offsets instead of indexes), they typically end up being dictionary encoded if the domain is small. Figure 6 shows the extent that dictionary encoding can be leveraged to improve the generation of sorted string heaps for SF 1 tables as well as the SF-30 lineitem and Flights tables.

Note that with no encoding, there were a total of five sorted heaps in the table set (the blue bars in the figure), mostly due to the TPC-H data generation algorithm or other accidents. With encoding on, however, all string heaps are sorted except one (`l_comment`), which has a large domain with low duplication.

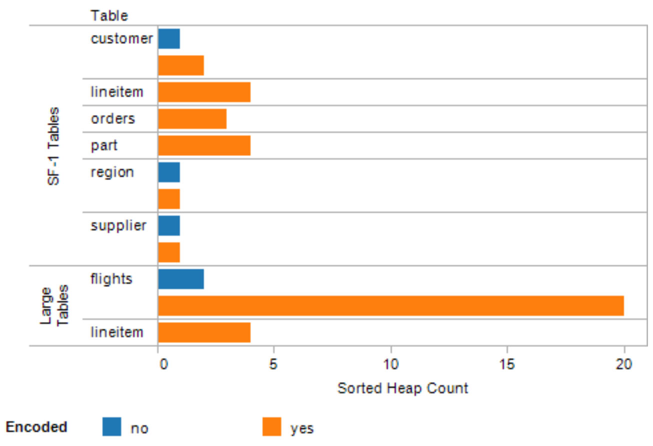

Figure 6. Number of Sorted Heaps.

Sorted heaps improve query performance by converting string comparisons to integer comparisons. The results shown in here demonstrate that this benefit can be realized for many string columns at no significant latency cost during the loading process.

6.4 Metadata Extraction

The encoding statistics enable the extraction of a number of metadata properties. Figure 7 shows the number of extracted metadata properties for the full set of tables, both with and without encoding active, and broken down into the set of all SF-1 tables and the two large tables. Heap acceleration was turned on for these tests.

Many of these properties were not even detected with encoding off and the few that were detected owe their detection to fortuitous circumstances such as the string data being inserted in order or as a side effect of the accelerator's statistics (e.g. domain cardinality.)

As we have shown in Sect. 6.1, this metadata was extracted by the system with no latency costs. The extracted metadata can then be used by both the TDE to improve query performance and by the Tableau visual system to enhance the analytic experience.

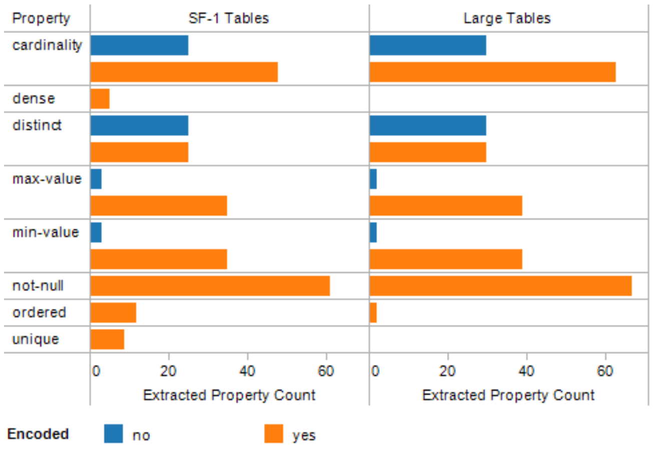

Figure 7. Metadata Detected.

6.5 Minimal Representations

The use of minimum width representations for scalars and tokens is another important optimization. When values have minimal widths, the system can choose better hashing algorithms for joins and aggregation. In Fig. 8, we can see that about three quarters of the string columns had their token width reduced from the default width of 8 bytes, often down to one byte. This can mean the difference between using an imperfect hash function with collision detection and using a perfect hash, or even a fast direct hash during joins and aggregation.

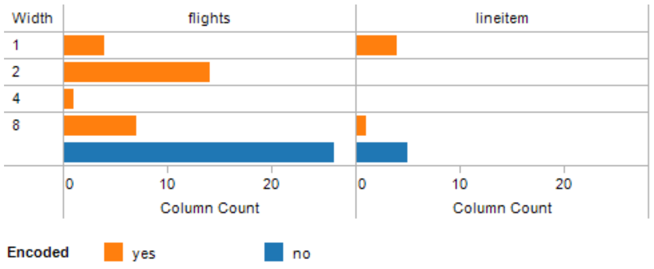

Figure 8. String Token Width Reduction.

A similar transformation can be performed on integer columns. Integers are parsed with a default width of 8 bytes, but often contain numbers from a much smaller domain. In Fig. 9, we can again see that about three quarters of the integer columns had their width reduced, often down to one byte, indicating that the values are in a very small range near zero.

These representation transformations were achieved without any additional import latency costs over simply splitting the file.

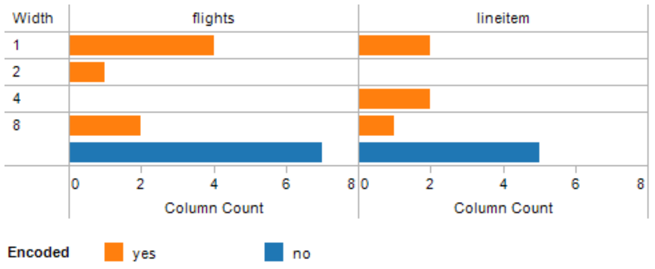

Figure 9. Integer Width Reduction.

6.6 Filtering

To test the performance of indexed scanning, we ran the following query over both run-length encoded tables:

```
SELECT Index, MAX(Other)
FROM table
WHERE Index > (100-selectivity)
GROUP BY Index
```

`Index` is one of the two integer sort columns (`primary` and `secondary`) whose run-length encoding index we are using and `Other` is the one we are not filtering.

To evaluate the indexed table operator, we tested the performance of three plans:

1. Scan => Filter => Aggregate
2. Index => Filter => IndexedScan => Aggregate
3. Index => Filter => Sort => IndexedScan => OrdAggr

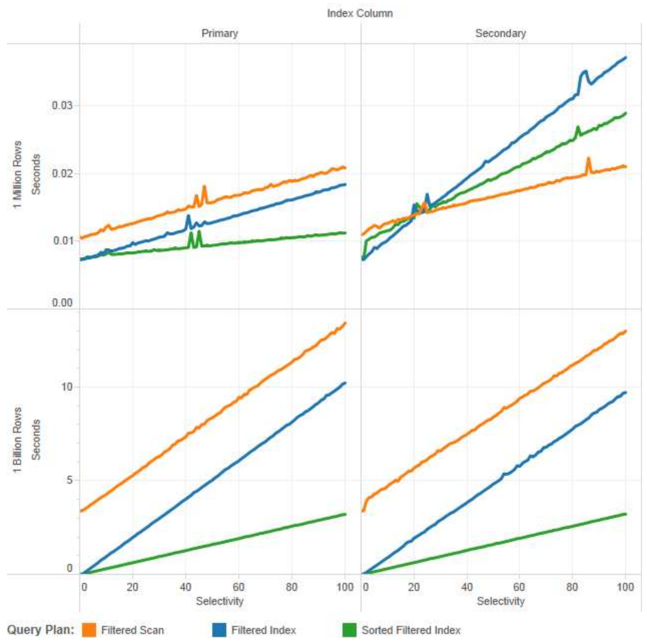

Figure 10. Indexed Filter Performance.

The first plan is a control, which fulfills the query using the existing system. The second plan applies the filter to the index, but relies on hash aggregation. The third plan also *sorts* the index, before scanning to allow the use of ordered aggregation.

We ran all three plans against both tables for filter selectivities between 0 and 100. Each plan was run 12 times, the two extremes were discarded and the remaining times were averaged:

As can be seen in Fig. 10, the plan that filters the run-length encoding index outperforms the other two plans by about a factor of two when the filtering is on the primary sort key.

We can also see that this plan gives the best performance for filtering the *secondary* sort column on the larger (1B row) table by nearly a factor of three. This is because we can apply a faster ordered aggregation to the secondary sort column, even though the table was not originally ordered on this column.

The only case where the sorted and filtered index plan does not outperform the other two plans is when filtering the secondary sort column on the smaller (1M row) table. In this case, the run lengths of the secondary encoding are only about 100 rows long, and the system ends up processing many more small blocks, which degrades performance past the point where the ordered aggregation can compensate.

7. CONCLUSIONS

In this paper, we described and evaluated various mechanisms for operating on compressed data in the Tableau Data Engine. These mechanisms can be used to significantly improve the performance of both data import and query execution operations.

The first mechanism consists of a novel set of techniques for post processing an encoded column without modifying the main body of the column. As part of the encoding process, we also extract metadata that can be used for later tactical optimizations during query execution. We have demonstrated that these techniques can be used to significantly reduce storage requirements and accumulate valuable metadata during thess loading of flat files without degrading performance.

The second mechanism introduces a new pseudo-table derived from a run-length encoded column and a corresponding join operator that can be used to express decompression as a join in a query plan. We have demonstrated that traditional optimization techniques of predicate pushdown can be used to improve the performance of a single-threaded filter and aggregate query by a factor of two when filtering primary sort keys. This result can be further extended to secondary sort keys if the secondary run lengths are larger than the block iteration size.

As part of our test process, we introduced a new flat file import operator for fast reading of large, well-designed formatted flat files with latency comparable to that required for only tokenising and splitting the text files. This initial import design should allow fairly responsive exploration of the data set without incurring any loading penalty beyond what was already required.

8. FUTURE WORK

Previous uses of parallel execution in the TDE have centered on multiple queries, sorting and data flow operators such as `Exchange` [8]. Distributing the processing of independent columns across multiple cores in the `TextScan` and `FlowTable` operators is new form of parallel processing in our execution engine. Our results suggest that there may be other places in column stores where work on independent columns can be easily and effectively parallelized with minimal synchronisation overhead. In the future, we intend to consider more uses for parallel computation that operate on independent columns.

One possibly interesting use case for `IndexTable` occurs when it is applied to a sorted date column. A common analytic calculation on dates is to roll them up to a higher level (e.g. rolling a date up to the start of the month or a date time upto the start of the hour). If this roll-up calculation is performed on the `IndexTable` , the computed result can then be aggregated on the rolled up date using `MIN(start)` and `SUM(count)`, which converts the original index on the raw date to one on the rolled up date. In the future, we hope to investigate using this technique for implementing parallel ordered aggregation on rolled up dates (or any other order-preserving calculation) by partitioning the index range and running the scan for each partition on a separate core.

The cost of rewriting a run-length encoding may be worth paying if the number of blocks is small compared to the full set of data in the column, but we have not investigated or quantified the use of this technique.

Because of its read-only, single file database format, the TDE is restricted by design from taking full advantage of the cracking approach. In spite of this, we would like to find ways to extend it to be able to reference external flat files and rebuild the database when the file changes. This would require a repackaging cost, but the user is most likely willing to incur this cost to have up-to-date data. Work along these lines may help us to find ways to integrate other aspects of database cracking techniques, possibly under user control so as to minimise unexpected IO costs.

9. ACKNOWLEDGMENTS

Our thanks to the rest of the TDE team (Chris Stolte, Pat Hanrahan, Matthew Eldridge, Cristian Vlasceanu, Marianne Shaw and Dana Groff.) We are also grateful to the members of the CWI group for valuable conversations and groundbreaking research that inspired much of the present work.

10. REFERENCES

[1] Stolte, C., Tang, D., and Hanrahan, P. 2008. Polaris: a system for query, analysis, and visualization of multidimensional databases. *Commun. ACM* 51, 11 (Nov. 2008), 75-84.

[2] Boncz, P., Żukowski, M., and Nes, N. MonetDB/X100: Hyper-Pipelining Query Execution. In *International Conference on Innovative Data Systems Research (CIDR)*, Jan. 2005, 225-237.

[3] Boncz, P. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Doctoral Thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[4] `http://www.tpc.org/`

[5] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., and Zdonik, S. 2005. C-store: a column-oriented DBMS. In *Proceedings of the 31st international Conference on Very Large Data Bases* (Trondheim, Norway, August 30 - September 02, 2005). Very Large Data Bases. VLDB Endowment, 553-564.

[6] Żukowski, M. *Balancing. Vectorized Query Execution with. Bandwidth-Optimized Storage*. Doctoral Thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, September 2009.

[7] Idreos, S., Alagiannis, I., Johnson, R., and Ailamaki, A. Here are my Data Files. Here are my Queries. Where are my Results? In *International Conference on Innovative Data Systems Research (CIDR)*, Jan. 2011.

[8] Graefe, G. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, June 1990.

[9] Żukowski, M., Heman, S., Nes, N. and Boncz, P. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, June. 1990.

[10] Wesley, R., Eldridge, M. and Terlecki, P. An analytic data engine for visualization in Tableau. In *Proceedings of the 2011 international conference on Management of data (SIGMOD)*, June 2011.

[11] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases (VLDB)*, September 2010, 330-339.

[12] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, (SIGMOD '06). ACM, New York, NY, USA, 671-682. DOI=10.1145/1142473.1142548 http://doi.acm.org/10.1145/1142473.1142548

[13] Srini Acharya, Peter Carlin, Cesar Galindo-Legaria, Krzysztof Kozielczyk, Pawel Terlecki, and Peter Zabback. 2008. Relational support for flexible schema scenarios. In *Proc. VLDB Endow.*, 1, 2 (August 2008), 1289-1300s

[14] `http://www.firebirdsql.org/`