An Analytic Data Engine for Visualization in Tableau

Richard Wesley Matthew Eldridge Pawel Terlecki
Tableau Software
{hawkfish, eldridge, pterlecki}@tableausoftware.com

ABSTRACT
Efficient data processing is critical for interactive visualization of analytic data sets. Inspired by the large amount of recent research on column-oriented stores, we have developed a new specialized analytic data engine tightly-coupled with the Tableau data visualization system.

The Tableau Data Engine ships as an integral part of Tableau 6.0 and is intended for the desktop and server environments. This paper covers the main requirements of our project, system architecture and query-processing pipeline. We use real-life visualization scenarios to illustrate basic concepts and provide experimental evaluation.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *Query processing*, *Relational databases.*

General Terms
Algorithms, Performance, Design

Keywords
Column store, Query optimization, Data visualization, Tableau Data Engine, TDE

1. INTRODUCTION
Tableau is a graphical system for performing ad-hoc exploration and analysis of customer data sets. It is a commercial continuation of the Polaris research project [1] and over the past several years has become a powerful component of the BI stack in many organizations.

Using Tableau, information workers can prepare interactive visualizations through a desktop application, which can either connect to an online data source or work offline on its own copy of the data, and is able to switch seamlessly between the two versions. The first option ensures consistency across all connected users but it requires a single database server to handle a substantial analytic workload. In many organizations the server machine of choice has other interfering responsibilities, often operational, or is not suitable for such workloads.

SIGMOD '11, June 12–16, 2011, Athens, Greece.

The offline case is not less important. Users often wish to perform analyses on copies of their data when:

- The original data is unavailable (e.g. offline operation while travelling);
- The data may be stored in a high-latency database that is not well suited for analytic queries (e.g. text files);
- The analysis is to be presented as a self-contained report.

These requirements led to the Tableau *extract* feature, which allows users to retrieve a portion of the original data and perform further analysis offline. Consequently, the system needs to be equipped with an internal data engine. Originally, this feature was implemented using the Firebird open source relational database. Firebird has a number of advantages for use in a commercial desktop application: small footprint, an architecture designed for embedding, complete SQL-92 semantics and a large set of native data types. Unfortunately, it also has an old "System-R"-style architecture and suffers from the analytic performance issues common to such systems, such as transactional locking, excessive I/O and row level operations [8]. In the spring of 2009 our team was formed to find or build a suitable replacement.

Our most important requirement was to efficiently handle the types of analytic queries produced by Tableau. Since a primary use case is the unstructured exploration of new data sets, the desired system needed to avoid unexpected performance cliffs associated with premature optimization of the data store for a small set of queries. For the most part, workloads consist of typical aggregation queries, but Tableau supports complex multidimensional filtering expressed through explicit predicates or lookup tables. Inner and left equi joins need to be supported. In addition, users can define new columns using a typical set of relational row-level functions. These user-defined columns can often be inherently slow to evaluate and it may be desirable to instantiate them for performance. Last but not least, Tableau makes frequent domain queries in order to drive various pieces of user interface, e.g. filter controls.

A second source of constraints was the wide variety of data types present in the myriad database vendors that Tableau connects to. In addition to several types of numerics and dates, Tableau also supports per-column comparison semantics. Extracting data from such systems should not change the semantics of the queries (including any locale-specific semantics), thus, the new system needed to have an easily extensible type system, including the ability to extend the set of collated string types.

Another design constraint came from the target platform for Tableau Desktop. A significant fraction of the target audience was presumed to be working on 32-bit Windows™ laptops with

2-4GB of RAM. These same users were also expected to be working with data sets that might exceed their working memory, thus, a non-memory-resident system was required that could perform well on this limited hardware configuration.

Moreover, the desktop application is mostly delivered via downloads from the Internet, which meant that the new system needed to have a relatively small execution image (Firebird's download footprint was on the order of 1.5MB).

The Tableau system is also provided in a server configuration for collaborative sharing of visualizations. The server operating system was intended to be 64-bit Windows™ to support larger addressable memory. Existing Tableau server deployments require only a fraction of available storage of a single machine. Under this simplifying assumption, we expected the database to scale-up and scale-out.

In addition, there was also a set of "non-requirements". Most notably, the system did not need to have any sort of single row update capability because Tableau does not perform such "write back" operations. Tableau also presumes that integrity constraints are to be enforced by the source database itself. Stored procedures are not used.

After reviewing the available systems (both commercial and open source) we concluded that there was no existing system that could meet our needs:

- Complex filters are often best expressed using left joins to temporary tables and many systems have trouble executing such joins quickly (or at all in some cases);
- Typical analytic systems (and even some major "mixed workload" vendors) do not support column-level collation;
- Most existing systems are designed for server environments where hardware can be chosen to match the workload;
- Installation footprints are often quite large and not typically designed for embedding.

Accordingly, we set out to build our own analytic query engine.

A review of the literature led us to a block-iterated column store design modeled on the MonetDB/X100 project [2] and subsequent work. Our new specialized data engine is further referred to as Tableau Data Engine (TDE). It can be configured to operate in either a single user desktop application environment or a shared server environment. The latter uses a shared-nothing architecture with inter-query parallelism. The focus of this paper is efficient data processing support for Tableau Desktop.

As mentioned before, our project was strongly influenced by the extensive research around MonetDB [2][3]. We also included some concepts on operations on compressed data from C-Store [4]. In addition, we investigated other commercial column stores. In particular, InfoBright [11] partitions its data into 64KB portions, called data packs. Rich statistical information associated with each pack enables a quick exclusion of data that are irrelevant for a given query. The Gemini add-on to Microsoft Excel is an embedded database with a column-oriented storage. As in our solution, compressed data is kept in a workbook. However, data processing capabilities in Gemini are currently limited by the available memory. Slow calculations can be persisted as columns on-demand, which is similar to extract optimization in Tableau (see Sect. 5.1).

Section 2 provides a running visualization example. The TDE architecture is covered in Sect. 3. Data compression and query optimization are given in Sect. 4 and 5, respectively. Section 6 presents experiments. We describe additional visualization support, such as query cancellation, in Sect. 7. The paper is concluded in Sect. 8.

2. Visualization Scenario

Tableau provides a drag and drop interface for interactive data exploration. It uses the VizQL language [1] to describe an analytical query and associated graphical layout. Queries are caused by user interactions and consequently have an ad hoc nature. The results received from the database are further post-processed and rendered to obtain a specified visualization.

Figure 1. Visualization of profit and sales correlation over time for different regions and products.

Let us consider a simple data relation with two measures Sales and Profit, and four dimensions: Time, State, Product and Supplier. We want to investigate the correlation of both measures over time for each Product and State. This can be accomplished by a tabular view where each column corresponds to a different Product and each row to a different State. Each cell of the grid contains the corresponding correlation chart (Fig. 1).

In order to make the visualization less detailed one may want to roll up on Time and State. Since no explicit hierarchies are defined on those dimensions, we define new higher dimension levels using auxiliary functions:

- Year = year_func(Time) ∈ {2000,..,2010}, defined as a 4-character prefix of Time; the latter is assumed to be encoded as a string;
- Region = region_func(State) ∈ {NORTH, WEST, SOUTH, EAST}, which is a manually-provided partition of State and can be expressed as a CASE-WHEN statement with equality conditions.

Tableau uses an abstract query representation to perform initial transformations and optimizations. Depending on the type of a target data source, an appropriate database query is generated.

Assuming that the data are stored in a denormalized relational table T, both measures and dimensions map to its columns. Also, the new dimension levels can be expressed as computed columns in the table's schema. The required data can be retrieved by the following SQL query:

```
SELECT    SUM(Profit), SUM(Sales),
          Product, Region, Year
FROM      T
GROUP BY  Product, Region, Year
```

Note that the data is already grouped to simplify the additional post-processing on the client.

We continue this example with respect to query processing in the data engine presented below.

3. SYSTEM OVERVIEW

For the purposes of exposition, it is convenient to view the Tableau Data Engine (TDE) as being comprised of several layers:

- A storage model;
- An execution engine;
- A query parser and optimizer;
- A communication interface;
- The Tableau VizQL compiler.

This section will give a brief description of each of these pieces; later sections will focus on the details of the compiler and execution engine.

3.1 Storage Model

The TDE has a typical three-level logical object namespace of schemas, tables and columns. For simplicity, this namespace is stored as a multi-level "directory" structure. Each table is then a directory that contains column files, each schema is a directory containing tables and a database is a top-level directory containing the schemas.

For most tables and columns, metadata is stored in special tables in the reserved SYS schema. Metadata in the SYS schema (and other special schemas like TEMP) is kept in yaml key-value files next to the column, one metadata file per object (column, table, schema). Object names can then be decoupled from the underlying file structure.

Column files are of two kinds: a fixed width array of values and an optional "dictionary" file. When a dictionary is present, the value array contains dictionary *tokens* instead of actual values.

The "directory" structure is abstracted to enable implementations other than the simple file system version described above. The most important implementation is a read-only implementation that packages a database as a single file for user convenience

3.2 Execution Engine

The TDE execution engine follows traditional database patterns. It supports a collection of operators and function primitives. Each operator implements a certain data processing algorithm and consumes rows on its optional inputs to produce output rows. A query plan is a tree of operators. It is executed by iterating over all the rows of the root of the tree. For the sake of performance, we employ block processing with a fixed block size and optional selection vector to mark valid rows [2].

The parser generates query operators, which come in two basic flavors: *streaming* (Data Flow) and *stop-and-go* (Table). The first ones can process input blocks independently, e.g. a projection that simply defines new columns in a block. On the other hand, stop-and-go operators need to consume all the input rows and materialize the intermediate result before any rows can be output. An aggregation of unordered data is an example.

In addition to the query operators, there are a number of command operators that implement DDL statements and miscellaneous server operations unrelated to query processing.

3.3 Query Parser and Optimizer

Tableau performs an initial analysis of a query to apply general optimizations valid across many target data sources (see Sect. 5). Inferences are made using an abstract tree representation of a query. We based our approach on selected concepts from the relational algebra.

Most engines use declarative languages and require appropriate translation of the internal query representation. To make similar translations straightforward on both ends of the wire for the TDE, we developed a Tableau Query Language (TQL). It preserves the semantics and tree structure of an abstract query built on the Tableau side.

The query parser accepts text commands in TQL and converts them into an in-memory tree representation. The initial tree further transformed by the optimizer and converted to an executable query plan.

3.4 Communication Interface

The Tableau Data Engine runs as a separate process communicating over standard sockets. The TDE side of the protocol just reads queries and other commands from the channel and routes them to a multi-threaded session manager. The results of the commands are then written to the channel.

In addition to the main communication channel, sessions can be addressed through a secondary control channel. This channel supports user interaction with running queries by reporting progress and allowing the user to cancel long-running queries in a responsive manner.

Tableau has an internal API that can be used to send and receive queries from a wide variety of data providers. In addition to a standard OLE DB/ODBC wrapper, the API can be used to wrap native implementations, such as the InterBase API used by Firebird. For the TDE we wrote another implementation of this API, which translates the operations into wire commands. The

implementation also connects query execution to the user interface to support progress feedback and query cancellation.

3.5 VizQL Compiler

Visualizations in Tableau are expressed via the VizQL specification language [1]. The VizQL compiler is a Tableau subsystem that accepts visualization specifications and generates relevant database queries for target languages, such as MDX and SQL. The existing Tableau SQL compiler was generalized to support TQL as a new relational dialect.

4. COMPRESSION

One of the most important benefits of a column store is the ability to compress data and then operate on the data in its compressed form [7]. Operations on compressed data can improve the performance of a typical analytic query by a factor of two [4]. The TDE implements two compression strategies: dictionary compression and run-length encoding. Dictionary compression is visible in query processing, while the storage engine performs RLE implicitly.

4.1 Tokens

Columns are simply arrays of a fixed width type. Their content is accessed through a data stream interface that allows processing data in portions fitting in memory. Most scalars (such as integers, doubles and dates) can be directly expressed in this array format. Such columns are referred to as uncompressed.

Compressed columns come in two forms. *Heap* compression is used for variable width types such as strings and *array* compression is used for fixed width types such as dates. The data portion of the column consists of *dictionary tokens* that reference members of the dictionary.

Because the data stream of a column is required to be an array, variable width types can only be stored in heap-compressed columns. Under heap compression, the tokens are offsets into the dictionary and the data is stored in the form <*length*> <*data*>. The set of offset is not dense, but they simplify the system by eliminating an index-to-offset indirection.

Fixed width types may also be compressed using array compression. In this format, the dictionary is an array of values and the tokens are indexes into this array. The set of array tokens is dense, which is a useful property for some operations. Array compressed columns can take up significantly less space than their uncompressed equivalent: for example, the TPC-H lineitem table has a date column that has only about 2500 distinct values. The TDE's date type is 4 bytes wide, but by compressing the column it can be represented by 2 byte tokens, saving 50% on the storage requirements.

Tokens are just integers, so they can be compared as integers. If the dictionary entries are all unique then the tokens are said to be *distinct*. Distinct tokens can be compared for equality and hashed consistently without consulting the dictionary. If the dictionary is also sorted, then the tokens are said to be *comparable*. Comparable tokens can be used for sorting and ordered comparisons.

Because the TDE works on fixed data sets, tables can be maximally compressed without having to declare the token width ahead of time – or even whether compression is wanted (e.g. columns declared *enum* in [3]). In addition to relieving the desktop user from the burden of being a DBA, this kind of maximal compression improves memory bandwidth and creates opportunities for better hashing.

4.2 Domain Tables

The relationship between a compressed column and its dictionary resembles foreign/primary key relationship, and this observation is central to how the TDE handles compression. Decompression is expressed in queries as a join between the main table's tokens and a virtual *domain table* representing the column's data dictionary. One interesting aspect of this approach is that it makes decompression a high level operation, which can be reasoned about in a natural way by the query optimizer

The TDE query optimizer can reorder predicates and computations across joins, reducing the amount of computation performed. For computations and predicates that only reference a compressed column, this means that computations can be performed on the column's domain instead of on every row of the table. For example, a computation to extract the year of a date in the lineitem table can be pushed down to the ~2500 date values in the domain table and only executed that many times. A filter on that year can also be applied before the join, further reducing the size of the join hash table.

4.3 Invisible Joins

Many predicates used in analytic queries consist of simple filters comparing a data value to a compile time constant. In the case where the column is compressed, we made use of the "invisible join" technique of Abadi et al. [4] to compare tokens instead of values. This necessitates translation of constants to the domain of the column to which they are being compared.

To enable this translation, TQL compiler makes a pass over the expression tree and attempts to attach a domain name to each constant. When the constant node is evaluated for the first time, the domain column can be looked up in the input name space and the value looked up in the column dictionary. If it is found, then the constant column is replaced with one that contains the constant's token and shares the dictionary with the domain column.

When the function node that references the constant and its domain column is evaluated, it attempts to use a version of the function that uses tokens instead of values. In the case of a comparison function (such as *equals*) this will lead to comparing tokens instead of compressed values.

Other functions may not support this optimization, but in that case sharing the dictionary does no harm (dictionaries are not copied, just referenced). For example, the *find(string,string)* function may coincidentally have a second constant argument that is in the domain of the first column, but since *find* does not support use of tokens, the string implementation will be used.

4.4 Lookups

While invisible joins are invaluable for improving the performance of simple predicates, Tableau typically expresses complex filters and other functions using joins. This led us to an extension of invisible joins called the *lookup* function.

4.4.1 Multidimensional Functions

Tableau allows users to generate complex multidimensional filters by selecting points in a visualization (such as a scatter plot) and either including or excluding them. The default implementation of the filter as a large sum-of-products expression tree is unwieldy both in the ability of a database to optimize it and in some cases is simply too large for the server's query buffer (e.g. Firebird has a 64KB query buffer). Such filters are more naturally represented as a lookup table containing the list of key sets to be included or excluded and a constant output column containing "true". Include filters are then expressed as inner joins and exclude filters are expressed as left outer joins followed by a predicate asserting that the output Boolean column is null.

Filters are a special case of a more general multidimensional function implemented via a lookup table. Tableau generates such functions in the form of the *group* calculation. A group is a mapping that collapses several values of a column into a single value. Typical use cases of this feature include higher level dimensional modeling and data cleaning. For example, a data set may contain sales data by state but the analysis requires the aggregation to be by sales region, which is not modeled. Or the data may contain states with variant names and the group can be used to combine the variant spellings. The user can define a grouping on the state column, which simply maps each state variant to its group.

The VizQL compiler attempts to implement such functions by creating temporary tables and joining them to the main relation. While the TDE allows the creation of such temporary tables for joining, the joins are often keyed on multiple string columns, which can lead to inefficient joins because the database is unaware that the inner table columns have domains that are subsets of the outer table. Solving the problem was the motivation for our new *Lookup* operator.

4.4.2 Hashing for Joins

The TDE supports a number of hashing algorithms for implementing equi-joins. The basic hash algorithm computes a hash, probes the inner table and then checks for collisions. If a collision is detected, it is added to a separate collision list for that hash value as in Zukowski et al. [5] that is checked sequentially whenever the hash value is encountered.

Collision checking can result in random access to the inner table, which can lead to serious performance degradation. Avoiding collisions is therefore quite desirable and the TDE implements a number of faster hashing algorithms that avoid collision detection and which can be used preferentially if their preconditions are met.

The width of a column for the purposes of hashing is the number of bits needed to represent a value in the column's data stream. For uncompressed columns, this is just the number of bits in the representation itself. For compressed columns with distinct tokens, the width is the size of the tokens, which in a fully compressed column is reduced to the minimum number of bytes needed to represent all the dictionary entries.

TDE hash values are 32 bit quantities and if the input columns have a total bit width of 32 bits or less, a perfect hash function can be constructed by concatenating the data bytes, which avoids the need for collision detection. The result is then run through a reversible mixer with good avalanche properties to make the bits more amenable for hash table lookups.

If the total number of bits being hashed is 16 or less, then an even simpler system of *radix* hashing may be used [2]. The bytes are concatenated with no mixing and used to index a 64K translation table.

Finally, if there is only one integer join column that is both *dense* and *ordered ascending* in the inner table, then the mapping from hash function to inner row id is essentially an identity (or at worst an affine transformation.) In such a *Fetch* join [2], no translation table indirection is required, which not only improves performance through reducing (or eliminating) computation but also avoids consuming valuable cache resources to contain the translation table.

4.4.3 Lookup

Given the constraints of the hashing system, it is desirable to reduce the width of the join columns in a filter or lookup table as much as possible. The simplest reduction would be to use the fact that each pair of join columns shares a domain. When the column is compressed (as is always the case for strings) the inner join column's dictionary is a subset of the outer join column's and we can replace the inner column with a copy that uses the outer column's dictionary and tokens instead of its own. This allows the hash algorithms to use tokens instead of the larger string values that they represent, which in turn can enable the system to use a more efficient hash algorithm for the join.

To take advantage of this optimization, the query compiler needs to be aware of the semantics of the join. TQL allows the direct expression of such functions by defining the *lookup* pseudo-function. Lookup takes a table, a list of column bindings to define the arguments and the name of a result column to produce as output. An optional "else" value can also be specified in the case of no match.

A special operator that is derived from the Join operator implements Lookup. It post-processes the construction of the inner table by attempting to rebuild the inner join columns to match the tokenization of the outer join columns. If it fails, it simply leaves the column alone for the default join hashing to handle.

4.5 Run Length Encoding

Unlike many column stores [7] that employ multiple forms of compression, the Tableau Data Engine does not attempt to operate on all these forms of compression. Instead of trying to operate on a range of compressed data formats (run-length-encoding, delta etc.), we have elected for simplicity to operate on the dictionary directly in the common case where a single column is involved in a computation. Further compression can then be applied to dictionary tokens at the stream level, which avoids forcing the query compiler to reason about the locality inherent in many other compression schemes. The case where two different compressed columns are used in the same expression was not optimized because unless the columns are correlated in some way that is reflected in their respective compression techniques, it did not seem that there was any benefit to doing so.

There is still a benefit, however, in lightweight compression for the purposes of reducing disk I/O, especially on the wide range of

systems that are the targeted operating environment. Accordingly, the TDE will also attempt to run-length encode the data streams for columns that have been sorted or have very low cardinality. In addition to reducing disk I/O this form of compression offers the possibility of skipping large blocks of rows during scan. This optimization will be the subject of future work.

This form of decompression is implemented at the stream level rather than at the compiler level. A column seeks to the next block of rows in its data value stream and the stream decompresses the data into an internal buffer. In this sense the system resembles the disk compression utilities popular in the late 1990s, which would compress data at the driver level.

5. QUERY OPTIMIZATION

Tableau is compatible with a wide spectrum of data sources, including text files, Excel, popular DBMSs, OLAP systems and the TDE server introduced in this paper. In order to generate more efficient queries, the initial optimization happens on the Tableau side of the wire.

The extent of Tableau optimizations is restricted by the data source input language, which in most important cases is declarative, e.g. SQL or MDX. Further more sophisticated optimization and target query processors perform actual plan generation.

Building on the optimization techniques used in the MonetDB database [3], we partitioned optimization of the query into what are called *tactical* and *strategic* optimizations. Strategic optimizations are query plan level choices such as operator reordering. Tactical optimizations are performance choices made during execution based on the actual data flowing through the system at a point in time, such as which hash function to use.

The TDE uses this partitioning to reduce the space of query optimization options. In fact, one ought to be able to compute most things ahead of time, but in practice, we have found that it simplifies system design by having orthogonal concerns implemented separately.

5.1 Tableau-side Optimization

Certain optimization techniques can be applied across many supported engines. Tableau uses an abstract query representation based on concepts from relational algebra to perform relevant transformations. A resulting tree is further translated to an engine-specific query.

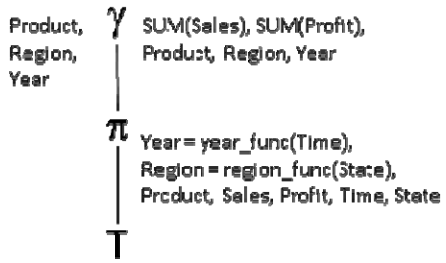

Figure 2. Query structure for the visualization from Fig. 1.

Figure 2 shows a relational representation of the query considered in Sect. 2. This simple scenario involves scanning the table T, computing additional columns and aggregating measures. Slow computations, such as string operations or complex case-statements, may significantly affect query performance. A potential improvement can be gained by pre-computing the expression values for all feasible combinations of referenced columns. The result is further stored in a separate lookup table joined with the main table. In our case, one can evaluate region_func(State) for all possible states and later join with T on State.

Although queries are created ad hoc, calculated fields are defined by users and are likely to be referenced in many queries. Expensive computations may be evaluated once and materialized as a new column of a fact table. It is a potentially costly DDL operation of a run time of the order of the size of the fact table. Therefore, materialization of user-defined computations is not performed automatically but can be triggered by the user as extract optimization.

Note that optimizations can be applied as long as a target database supports necessary features, e.g. creation of temporary tables.

5.2 Strategic Optimization

The TDE is equipped with a query rewriter responsible for transforming a parsed query tree into a form supported by the execution module. Besides optimizations, it implements complex operations, such as quantile or count distinct computation, by means of more basic operators, so that compilation of a final execution plan is straightforward.

We use a relational data model, where an extract is represented by a set of tables. Facts and dimensions are stored in a single table in a denormalized form. In addition, Tableau may create other auxiliary tables, e.g. to look up computations results.

TQL queries generated by Tableau process the data from the fact table performing desirable computations, filtering, aggregations, etc. Compressed dimension columns are treated as their values were present in the table and implicitly handled during plan generation. In fact, tokens may be sufficient for certain operator, such as grouping or sorting. In our example, the values of Product are not necessary to perform the aggregation and they need to be retrieved only at the end to prepare the final result set. Otherwise, decompression is implemented by introducing foreign key joins to appropriate internal dictionary tables.

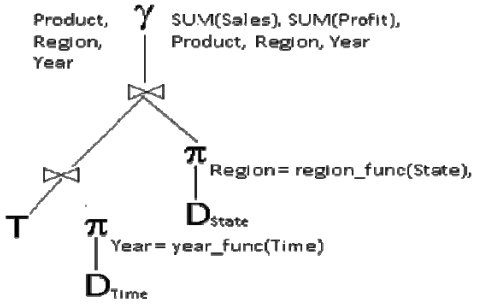

Figure 3. Query structure for the visualization from Fig. 1 after expanding compressed columns to joins and pushing down calculations.

Significant efficiency gain can be obtained by appropriate placement of filters and computations in a plan. The way compressed dimensions are expanded into joins allowed us to leverage classic methods known from the literature [10].

In fact, our reordering heuristics are designed for the most common case of left-deep trees with a fact table as the left-most leaf. We assume that a subtree referencing a fact table always has higher cardinality than subtrees referencing only dictionary tables. Joins are assumed to have the same fractional selectivity. Similarly, selections are believed to have the same fractional selectivity and are more selective than joins. Finally, all computations have the same positive costs.

As a consequence, selections are pushed down as close to relevant tables as possible. Computations involving a single compressed column are performed on corresponding dictionary tables. Otherwise, they are pulled-up in the tree as far as possible, since subsequent joins are most likely to reduce cardinality.

Figure 3 shows the original query after expanding compressed columns Time and State to joins that fetch their actual values. These columns are involved the calculations defining new columns Year and Region. In fact, these calculations could be pushed down to domain tables D_{Time} and D_{State}.

Some columns are required only up to some point in a tree, e.g. their values are used for data partitioning or computation. Since carrying them over to the root would defeat the purpose of column-oriented processing, appropriate restriction projections are placed in the plan. In the considered example, one needs only the Product, Sales, Profit, Time, State columns from the fact table. Also, the Time and State token columns can be restricted right above a respective join.

5.3 Tactical Optimization

When a query operator is invoked for the first time, it needs to settle the metadata description and set up the data storage for its output columns before it can begin operation. This process is called *column finalization*. During this process the complete metadata of the column is defined, data columns are allocated and any dictionaries that can be shared are identified.

For leaf tables, the process is straightforward: the data streams for each column are read from the data store and attached to the columns of the operator.

Operators that compute values, such as *Project* and *Aggregate*, need to perform *late binding* of the functions used in their computation expression trees based on the newly available type information. For constant expressions, this may include translating the constant into the dictionary of the domain that the constant is associated with as described in section 3.3 – a decision that cannot be made without having the actual dictionary involved.

Operators that can work with compressed data need to know if the tokens have needed properties like being distinct or comparable. These properties may depend on the actual results of calculations, which cannot be known at plan generation time. For example, strings generated by *left(limeitem.l_comment, 3)* may have low enough cardinality that the column's heap can tell that the strings are all unique (e.g. its internal hash table did not overflow.)

Join and *Aggregate* nodes also need to decide upon a *grouping strategy*. If the join fields or aggregation list are not empty, then various hashing strategies need to be considered based on detailed consideration of the grouping column metadata. In the case of the *Lookup* operator, this choice cannot be made until runtime because the coercion of compressed column data to a single dictionary representation is not known until run time when all the data has been loaded and computed.

Order and *TopN* nodes have to perform late binding of their sort functions based on the data type of the column, which may depend on the actual data involved. For example, a string column may be computed by an expression and coincidentally produce ordered dictionary tokens (not because the data is sorted, but because new values are inserted into the dictionary in order.) This column can then be sorted on its tokens rather than on its string values.

6. VISUALIZATION SUPPORT

6.1 Domain Metadata

The TQL language, allowing Tableau to query directly for column domains, supports the domain table abstraction described earlier. In addition, the metadata for each column may contain the cardinality of the domain and other useful metadata such as the minimum and maximum values. These latter can be used by the user interface to choose an appropriate level of detail for a date hierarchy.

6.2 Progress and Cancel

Despite our best efforts, there will be queries that require noticeable processing time. In such cases, the user experience benefits from feedback on how a query is progressing and the ability to cancel long running queries. Cancellation is especially important in an interactive exploratory environment such as Tableau where a user may be dragging fields out in an attempt to build a visualization and may not be interested in intermediate results produced by an incomplete VizQL specification.

The TDE provides sideband progress reporting and cancel control for all queries. Progress is reported using the driver node estimator system described in [9] and includes upper and lower bound estimates. Cancel requests are handled by a separate thread and passed on to all *Scan* nodes and any other nodes that perform large amounts of internal processing (e.g. *Order*).

7. EXPERIMENTAL EVALUATION

The primary reason for building a new data engine was inefficiency of row-oriented databases against an analytic workload. We demonstrate that the TDE gives a significant performance improvement over the previously used Firebird database and comparable performance to MonetDB [3].

All experiments were conducted on a single DELL machine with Intel Xeon E5520 with two 2.27GHz physical cores, 12Gb of RAM and running 64-bit Windows 7.

7.1 TPC-H Results

Figure 3 shows a performance comparison of the TDE against Firebird and MonetDB conducted against a subset of the TPC-H benchmark [6]. The test database was generated at SF=1 for all three servers using vendor-supplied translations of the TPC-H queries and build scripts. Each test was run 5 times in the same hardware with the data set warm in the disk cache and the results

were averaged. Output was redirected to a file for all three databases.

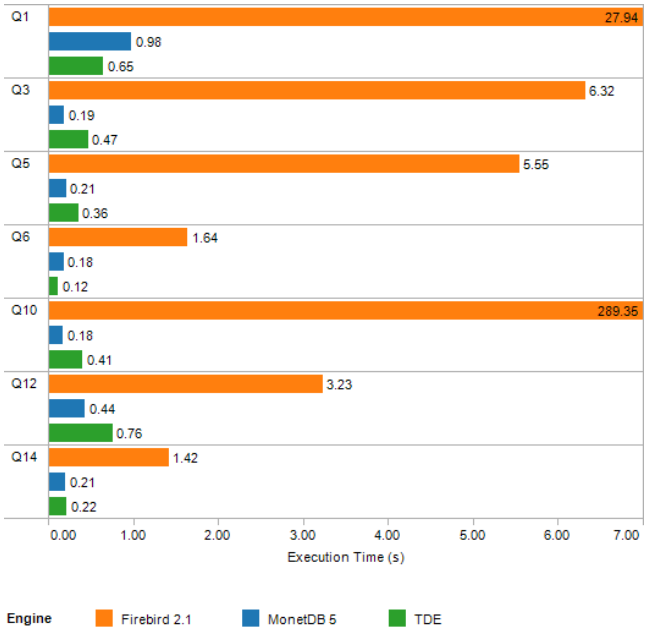

Figure 3. Execution times for selected queries from TPC-H for Firebird, MonetDB and the TDE.

Replacing Firebird with the TDE brings improvement of 1-3 orders of magnitude. At the same time, due to generally similar design principles, the TDE shows performance comparable with MonetDB.

Extract files created for Firebird, MonetDB and the TDE had sizes of 1.9 Gb, 1.06Gb and 641Mb respectively. The Firebird page size was set to 8Kb. The significant differences result from page allocation strategies in Firebird and compression in the TDE.

7.2 Flights Results

Tableau generates specific classes of queries. We prepared an in-house benchmark, called Flights, that represents a data exploration session. The data set has a 70M row fact table containing 10 years of FAA on-time flight statistics for the US.

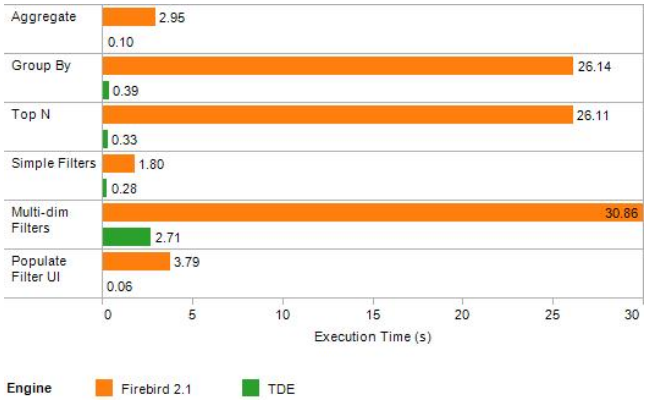

Figure 4. Execution times for selected queries from TPC-H for Firebird and the TDE.

Figure 4 shows performance comparison for Firebird and the TDE for a 0.5M row fact table. This significant data set reduction was required to let Firebird successfully process the queries. For brevity, we partitioned the testing workload into six categories and reported sums of individual query times. Again we warmed the caches and averaged the results over 5 runs.

We observe a similar performance impact of 1-2 orders of magnitude and impact of compression on database sizes. Extract files for Firebird and the TDE had 221Mb and 45Mb, respectively.

7.3 Integration Testing

The TDE was developed at Tableau over the course of 18 months and first deployed commercially in the fall of 2010. In addition to the benchmark tests just described, the TDE has been through a full release test cycle including internal testing, automated test suites and an external beta testing using real customer data. The Tableau automated test suites include about 1000 correctness tests that are applied on a daily basis to all supported versions of the databases supported by the product including the TDE.

8. CONCLUSIONS

In this paper, we introduced the Tableau Data Engine, a specialized column store modeled after MonetDB. The new engine is an integral part of the Tableau 6.0 release. It provides significantly faster data processing for Tableau Desktop. In addition, it is a default extract engine for Tableau Server and, in our opinion, it remains a convenient alternative for a wide group of customers who do not own a dedicated analytic database server.

The TDE meets our functional and non-functional requirements for an extract engine and brings 1-3 orders of magnitude improvement over the previously used Firebird database. This enables true interactivity in exploration of large data sets with hundreds of millions of rows.

Importantly, developing an own analytic data engine removed a strong dependency of our product on a third party database. We control both ends of the wire, therefore, we can relatively easily introduce new analytic functionality to Tableau with an efficient implementation in the TDE. Also, particular performance issues can be addressed directly in the engine.

Compression plays an important role in efficient processing of column-oriented data. In the future we would like to support some other forms of stream compression such as delta encoding (for columns with local ordering) and bit field encoding (for columns with very small member counts.)

Furthermore, we intend to collect more statistical information on data and employ some traditional optimization strategies during plan compilation, such as heuristic join reordering or incremental execution.

As far as the throughput is concerned, the TDE can run multiple independent queries at the same time allocating them to separate processor cores. While a sufficient strategy for the Tableau Server, the desktop application suffers from highly inefficient usage of system resources for serially submitted queries. We plan to address cases of slow computations and aggregations with intra-query parallelism.

Last but not least, the TDE server may require extensions with respect to resource governing, administering and monitoring. Also, we plan to perform extensive scalability and availability study.

9. ACKNOWLEDGMENTS

Our thanks to the rest of the TDE team (Ken Ross, Chris Stolte and Pat Hanrahan.) We are also grateful to Stefan Manegold and other members of the CWI group for advice and inspiration.

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., Zukowski, 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] Abadi, D. J., Madden, S. R., and Hachem, N. 2008. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international Conference on Management of Data* (Vancouver, Canada, June 09 - 12, 2008). SIGMOD '08. ACM, New York, NY, 967-980.

[5] Zukowski, M., Héman, S., and Boncz, P. 2006. Architecture-conscious hashing. In *Proceedings of the 2nd international Workshop on Data Management on New Hardware* (Chicago, Illinois, June 25 - 25, 2006). DaMoN '06. ACM, New York, NY, 6.

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

[7] 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.

[8] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases* (VLDB '07). VLDB Endowment 1150-1160.

[9] Chaudhuri, S., Narasayya, V., and Ramamurthy, R. 2004. Estimating progress of execution for SQL queries. In *Proceedings of the 2004 ACM SIGMOD international Conference on Management of Data* (Paris, France, June 13 - 18, 2004). SIGMOD '04. ACM, New York, NY, 803-814.

[10] Hellerstein, Joseph M. 1994. Practical predicate placement. In proceedings of SIGMOD. ACM, New York, NY

[11] Slezak D., Eastwood V..2009. Data warehouse technology by infobright. In proceedings of SIGMOD. ACM, New York, NY