On Improving User Response Times in Tableau

Pawel Terlecki
Tableau Software
pterlecki@tableau.com

Fei Xu
Tableau Software
fxu@tableau.com

Marianne Shaw
Tableau Software
mshaw@tableau.com

Valeri Kim
Tableau Software
vkim@tableau.com

Richard Wesley
Tableau Software
hawkfish@tableau.com

ABSTRACT
The rapid increase in data volumes and complexity of applied analytical tasks poses a big challenge for visualization solutions. It is important to keep the experience highly interactive, so that users stay engaged and can perform insightful data exploration.

Query processing usually dominates the cost of visualization generation. Therefore, in order to achieve acceptable response times, one needs to utilize backend capabilities to the fullest and apply techniques, such as caching or prefetching. In this paper we discuss key data processing components in Tableau: the query processor, query caches, Tableau Data Engine [1, 2] and Data Server. Furthermore, we cover recent performance improvements related to the number and quality of remote queries, broader reuse of cached data, and application of inter and intra query parallelism.

1. INTRODUCTION
Data visualization and big data processing have become key elements of modern analytic platforms. Lower hardware prices allow customers to purchase powerful computational clusters, collect high volumes of detailed data and perform sophisticated ad hoc analysis. Moreover, there is a growing demand for standardized solutions in related areas, such as: data federation, exploration of semi-structured and graph data sets, and data preparation or knowledge discovery. The increasing popularity of visual analytics is driving development in these fields and provides the research community with new data processing and usability challenges.

In the past years, Tableau has become a leader in providing easy-to-use data visualization tools. It currently connects to over 40 different data sources, from simple file-based sources, through SQL/MDX databases or web data sources, to Hadoop clusters. In a common intuitive interface, it allows for rich analysis of different areas of data, including filtering, custom calculations – potentially at different levels of detail, window and statistical functions and many others.

Tableau aims to provide a fast interactive environment for data exploration. A key challenge, both today and as we continue to add more sophisticated data analysis, is maintaining high responsiveness of the interface. To achieve that across all the supported data sources, one often needs to apply dedicated optimization techniques to alleviate efficiency and capability shortcomings of various architectures.

During data exploration, each user interaction with the application generates an adhoc query workload. This workload is difficult to predict and requires powerful analytic databases, e.g. column stores, to process requests with low latencies. Additionally, we leverage caching and query optimization techniques to provide high levels of interactivity.

Sharing the results of data exploration via Tableau Server in the intranet or Tableau Online, in the cloud, presents a different set of workload challenges. Users can combine visualizations created during data exploration into dashboards and publish them to Server. A dashboard consists of multiple visualizations; they are often linked by actions, such as filtering of one view based on selection in another.

The query workload generated by published dashboards is easier to predict than the adhoc workload generated during exploratory analysis. This is because queries are generated by interacting with schema components already defined in the dashboard. Moreover, caching is more efficient as it can be applied across multiple users accessing the same dashboards.

Although with different emphasis, providing a highly interactive experience in both aforementioned scenarios depends upon the efficient retrieval of new data from external sources and maximal utilization of previously obtained results. In this paper, we present recent performance improvements focused on increasing product interactivity, including:

- Query batch preparation and concurrent query execution to improve dashboard generation;
- Persisted and distributed intelligent cache to improve response times across users and their interactions;
- Parallel plans and fast scanning of RLE encoded columns in Tableau Data Engine;
- Managing data models and extracts in Data Server along with temporary table support.

The text is organized accordingly. Section 2 covers the general background. In Sect. 3 we describe how dashboards are generated focusing on query processing and caching. Section 4 gives an overview of the Tableau Data Engine and its recent performance features. In Sect. 5 we look at connection and data model management in the Data Server. Section 6 covers related work. Future plans are given in Sect. 7 and the paper is concluded in Sect. 8.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-2758-9/15/05…$15.00.
http://dx.doi.org/10.1145/2723372.2742799

Figure 1: Sample dashboard for the FAA Flights On-Time dataset

2. BACKGROUND

Tableau is an application for rapid interrogation of structured data. The application has a simple, intuitive drag-and-drop interface that generates queries against a wide spectrum of data sources. The application has a large number of built-in chart types. Users are offered a rich environment for analyzing their data, including defining new calculations, forecasting, filtering, and combining data from heterogeneous data sources [47].

User actions that add and remove items in a worksheet are represented with VizQL [40], a set of specialized algebraic operations. This representation is used to automatically generate queries against a specified data source. Queries are issued and the results are interpreted to produce the requested visualization. As the user interacts with elements of the visualization, additional queries may be issued to the underlying database.

The Tableau Data Engine (TDE) is a proprietary column store database that can be used to allow users to extract data from an existing database or text file. When a TDE extract is incorporated into a user's workbook, a user can continue to interrogate their data when a network connection to the database is not available. Additionally, the extract reduces the query load on the backing database. Users can easily switch from an extract back to a live connection, and extracts can be refreshed when appropriate.

Tableau Server allows users to easily share their visualizations with others. Users publish data sources or visualizations to Server and specify sharing permissions for them. These visualizations can be viewed and interacted with via another Tableau application, or simply through a Web browser. If visualizations are published with accompanying TDE extracts, a schedule can be created to automatically refresh the extracts, ensuring the data is always current.

3. QUERY PROCESSING FOR DASHBOARDS

Dashboards are a vital element of many data visualization platforms. Each covers a certain area of data and allows users to view it from different angles. Quality of experience depends on the degree of interactivity, a factor that distinguishes dashboards from old-fashioned static reports.

A dashboard is a collection of zones organized according to a certain layout. Different zone types include various charts, maps, filters, calculated text, images etc. One defines the behavior of individual zones first and then specifies dependencies between them. Complicated scenarios might involve tens or even hundreds of zones.

Let us consider a simple example of a dashboard (Fig. 1) defined for the popular FAA Flights On-time dataset [43]. It covers all the flights in the US in the past decade. The two upper maps show the number of flight origins and destination by state and, at the same time, allow specifying origins and destinations for the slave charts at the bottom. Each chart is annotated with average delays and flights per day. The bottom charts cover airlines operating the flights, destination airports, breakdown of cancellations and delays by weekdays, and distribution of arrival delays broken down by hours of a day. The right-hand side has filtering, total count of visible records and static legends.

Rendering a dashboard requires retrieving necessary data and post-processing it for visualization. Due to dependencies between zones, the entire process might take several iterations in which obtained responses are used to specify subsequent requests; the process is discussed in detail in Sect. 3.3.

Most zones require data from one or more data sources in order to be rendered. The requests are specified internally by queries with expressiveness matching the capabilities of the data sources. These internal queries eventually get turned into textual queries in appropriate dialects and sent to the corresponding data sources.

In the next section we discuss efficient data retrieval and generation of high-quality remote queries as both vastly contribute to the latency of dashboard interactions.

3.1 Single Query Processing

The internal queries formulated by components in Tableau closely follow the concepts of the application. In general, the queries

express aggregate-select-project scenarios, with potential subqueries for computed columns of different levels of detail and more sophisticated filters, such as top-n.

A query gets executed against a certain view on the data of a single data source. Users can specify views as single tables (e.g. denormalized SQL relations, OData sources, and text files), multi-table joins (often star or snowflake schemas), parameterized custom SQL queries, stored procedures or cubes.

Before a query can be sent to a relevant backend, it undergoes a compilation process consisting of structural simplification and implementation. First, a query is translated into a tree of elementary operations to facilitate subsequent transformations. Numerous optimizations are applied to the tree, including join culling, predicate simplification and externalization of large enumerations with temporary secondary structures. The query compiler incorporates information about cardinalities, domains, and overall capabilities of the data source, such as support for subqueries, temporary table creation and indexing, or insertion over selection.

A simplified query is subsequently translated into a textual representation that matches the dialect of the underlying data source. While most supported data sources speak a variant of SQL or MDX, each has their own exceptions to the standard and non-compliances. Moreover, out of the wide spectrum of scalar and aggregate functions available in the system, the native implementations might vary a lot due to type promotion and casting, function efficiency or even feasibility. As a result, some operations may need to be locally applied in the post-processing stage.

Tableau communicates with remote data sources by means of connections. Most often a connection maps to a database server connection maintained over a network stack. In some cases it might be entirely handled on the side of the application, e.g. parsing of text or Excel files. Communication with the drivers is encapsulated in external processes to isolate failures in the third-party software and potentially greedy resource allocation from the rest of the system. Retrieved results are streamed back in a tabular format.

3.2 Query Caching

Generating visualizations is expensive, therefore, Tableau utilizes several levels of caching. In this section we focus on the product's use of query-related caches, as data retrieval is the most time-consuming stage of the entire process. Other examples include caching entire visualizations, image and map tiles, etc.

As a user interacts with a dashboard, it is likely that identical or similar visualizations will be requested at each refresh of the view. In a multi-user scenario, it is even more common to get identical or similar requests, since different users are working with the same shared dashboards. An extreme example of this is seen in Tableau Public, which allows for publishing visualizations and data, and referencing them in personal articles or blogs. The user-generated traffic is saturated by initial load requests, as many viewers just read content with the initial state of a dashboard and make further interactions rarely.

Tableau incorporates two levels of query caching: intelligent and literal. The intelligent cache maps the internal query structure to a key that is associated with the query results [48]. When a new query is to be executed, a cache key is generated and the intelligent cache is searched for a match. When looking for matches, we attempt to prove that results of the stored query subsume the requested data [19, 20]. While currently we accept the first match, in the future release, we plan to choose the entry that requires the least post-processing. Also, even though the matching logic is designed to be fast, we are planning to maintain an index over the cache to minimize the lookup time.

The applicability of the intelligent cache is limited by proving capabilities and efficiency, e.g. analyzing implications of predicates, potentially large or formulated in different equivalent ways, and by post-processing capabilities. The latter includes roll-up, filtering, calculation projection, and column restriction. It is worth noting that certain operations cannot be performed locally, in particular, if they involve native function calls or collation conflicts.

The intelligent cache can be treated as a database view-matching component. It keeps the application highly responsive as long as covering data is available and can be post-processed. The additional post-processing usually does not require much time as we retrieve data in small, pre-filtered and pre-aggregated volumes. For many queries, the number of tuples returned is of smaller size than the number of un-aggregated, unfiltered rows in the database. Local post-processing occurs on this relatively small set of aggregated data. The query processor might choose to adjust queries before sending, in order to make the results more useful for future reuse.

The literal query cache contains low-level queries that are not directly related to visualization generation; it is keyed on the query text. It is used to match internal queries that end up having the same textual representation but where a match could not be proven upfront without performing complete query compilation. Predicate simplification based on domains or join culling are some examples of this scenario.

Cache entries in both the literal and intelligent cache are purged based upon a combination of entry age, usage, and the expense of re-evaluating the query. Entries are also purged when a connection to a data source is closed or refreshed.

Let us revisit the sample dashboard in Fig. 1 in the context of caching. Note that the queries for the domains of filters on the right need to be sent only once. Further interactions might change the selection but not the domains. Furthermore, data for other charts got cached with all the filtering values selected. If a user deselects some of the values on the right, the intelligent cache will be able to filter out the necessary rows for the results for other charts, as long as the filtering columns are included.

In Tableau Desktop query caches get persisted to enable fast response times across different sessions with the application. Tableau Server does not persist the caches but it utilizes a distributed layer based on REDIS [41] or Cassandra [42] depending on the configuration. This allows sharing data across nodes in the cluster and keeping data warm regardless of which node handles particular requests. For efficiency, recent entries are also stored in memory on the nodes processing particular queries. In general, we cache all the query results unless computation time is comparable with a cache lookup time or the results are excessively large

3.3 Query Batch Processing

From a user's perspective, the ultimate goal is to immediately load an entire dashboard or refresh it after a given interaction. Therefore, the data retrieval task is not formulated with respect to

individual queries but in terms of minimizing the latency of processing all of them.

Due to dependencies between zones, rendering of a dashboard might require several iterations to complete. Consider the dashboard in Fig. 2, which shows the Flights/Day for a Market, Carrier, and Airline Name for the FAA Flights data from Figure 1. The Carrier zone is filtered to the top 5 carriers, based upon number of flights, that have more than 1,400 Flights/Day. The dashboard has two interactive filter actions defined: (1) selecting a field in the Market zone will filter the results in the Carrier and Airline Name zones, and (2) selecting a carrier in the Carrier zone will filter the Airline Name zone. In Figure 2, the user has already selected a Market (LAX-SFO) and a Carrier (AA).

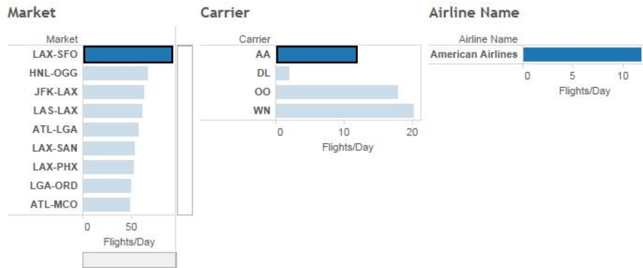

Figure 2: A dashboard with three zones, linked by two interactive filter actions. Selecting items in either the Market or Carrier zones filters the viz results.

If the user selects HNL-OGG in Market, the interactive filter actions will cause the Airline Name and Carrier zones to be updated with a filter on HNL-OGG. One side-effect of these updated results is that the previous user-selection (AA) in the Carrier zone is eliminated, as AA is not a carrier for the HNL-OGG market. Subsequently, the interactive filter action will cause a query without a filter on Carrier to be generated to update the Airline Name zone.

Zones that participate in an iteration generate queries that together comprise a query batch of this iteration. As the iteration and, thus, the corresponding batches are processed independently, here we examine the optimization of a single query batch.

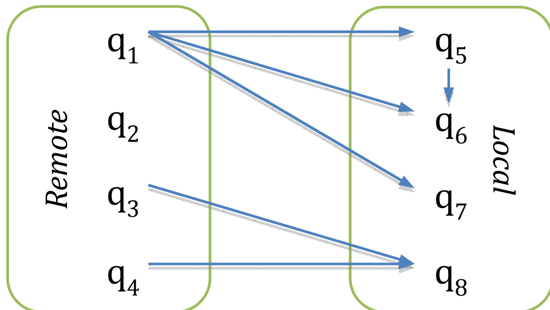

Figure 3: Sample cache hit opportunities graph with partition of queries into local and remote

Consider a query batch $B=[q_1, .. , q_n]$ and assume that the target data sources are all idle and have all the resources available. In addition, to express cache hit opportunities, consider a directed graph G with the queries as nodes and edges pointing from q_i to q_j iff the result of q_j can be computed from the results of q_i (Fig. 3). The latter is determined by the matching logic of the intelligent query cache.

One way of executing these queries is to send them one by one in a certain order, and each time wait for the query to complete. This way each query has all the resources for itself and its processing time is the shortest. Note that the query execution order matters and topological orders in G are most beneficial, as the queries with incoming edges will be answered from the cache. This strategy roughly corresponds to processing zones serially.

Serial execution of queries minimizes their individual processing times but does not utilize available resources to the fullest. Therefore, executing queries concurrently might reduce the overall rendering time of a dashboard. In order to maximize cache usage we process the batch in two phases. First, we analyze it and partition the nodes of G into two sets. One set contains queries that need to be sent to the remote back-ends; they correspond to the source nodes, i.e. the nodes without incoming edges. The second set contains queries that are cache hits that can be processed locally. In the second phase, remote queries are submitted for execution concurrently and the local ones are processed as soon as any of their predecessors in G finishes.

3.4 Query fusion

Grouping queries in batches creates an opportunity to utilize multi-query optimization techniques. One basic optimization we apply across queries before executing a query batch is combining groups of queries defined over the same relation and potentially different with respect to their top-level projection lists. Strictly speaking, we replace a group of queries of the form $[\boldsymbol{\pi}_{P1}(R), .. , \boldsymbol{\pi}_{Pn}(R)]$ with a single query $\boldsymbol{\pi}_{P}(R)$, where R is the common relation, $P_1, .., P_n$ are respective projection lists and $P= \cup_{i=1,..,n} P_i$. We refer to this transformation as *query fusion*.

One obvious advantage of this approach is a reduction of the number of queries in a batch and, thus, the overall overhead of dispatching queries in corresponding data sources and necessary communication. Since it is quite common for different zones of a dashboard to share the same filters but request different columns, the reduction might be substantial.

More importantly processing of a fused query is often much more efficient than processing of the individual queries that were fused, as the underlying relation needs to be computed only once. Although it is possible that a physical layout of the data can lead to a performance regression, this optimization brings significant gains in our visualization benchmarks.

3.5 Concurrent Execution of Queries

In order to process dashboards in parallel, Tableau needs to concurrently execute queries against external data sources. Since the capabilities, stability and efficiency of the many supported back-ends often vary dramatically, the communication and execution layers need to be universal. This section explains the design choices and their consequences for different data source architectures.

As noted in Sect. 3.1, Tableau uses the abstraction of a connection to communicate with external data sources. There are basically two strategies to submit queries concurrently: by using a single or multiple connections. The first route is rarely supported by databases and usually comes with functional or performance limitations. SQL Server with MARS enabled is one example here. On the contrary, maintaining separate connections to allow for concurrent access from independent clients belongs to a basic repertoire of most database servers. Their responsibility is to govern resources appropriately across the corresponding workloads and make sure none gets starved.

Tableau manages a certain number of active connections to each data source to implement concurrent execution of remote queries. The process of opening a connection, retrieving configuration information and metadata are costly, therefore, connections are pooled and kept around even if idle.

In addition, connection pooling plays an important role in preserving and reusing temporary structures stored in remote sessions. For example, temporary tables created for large filters or multi-dimensional sets during query processing are likely to be useful while formulating queries within the same query batch or within batches originating from interactions with the same dashboard. An age-wise eviction policy is used in case of local memory pressure or to release remote resources unused for longer periods of time.

Queries submitted by different components of the system are multiplexed across connections regardless of their remote state. That means that popular temporary structures will be duplicated in several connections. While this adds a certain overhead to the backend, it also makes the connections potentially more useful for subsequent queries. Some architectures allow to alleviate this overhead and share a single remote session across multiple connections, however, this comes at a price of more complicated locking on the server side to resolve concurrent changes across the participating connections.

While Tableau generates a highly parallel workload, the overall efficiency of the system depends on the processing ability of the back-ends. The latter is determined by several aspects of the database architecture and the underlying hardware. First of all, the sooner a database server receives a workload the better it can schedule execution of the participating queries. Generally, databases do not perform cross-query optimization with respect to query plans. This would require substantial pre-processing and affect short-running queries. Nonetheless, certain optimizations might be applied in lower layers. One example is the shared scans feature [45] present in several systems, including SQL Server. It allows the storage layer to pipe pages of a single table scan to multiple concurrently handled execution plans.

Resource allocation on the remote side plays a pivotal role in handling a concurrent workload efficiently. While some queries are already being executed, an arrival of a new query might cause the previously assigned resources to be reclaimed. Furthermore, even though clients might be sending a high number of concurrent queries, the database is likely to throttle them based on available resources or a hard-coded threshold. One might need to tune the system to appropriately manage resources of the entire cluster or their allocation across connections. Also, it is worth noting that some systems impose limitations on the overall number of connections or the number of concurrently executed queries.

Moreover, the way a database allocates CPU in the single query execution substantially affects performance. Many architectures use a single thread per query. That means that a serial execution of a query batch would leave a tremendous amount of processing power idle. On the contrary, massively parallel clusters or non-clusters solutions that support parallel plans, such as SQL Server, the TDE (see 3.2) and many others, will attempt to allocate many machines/CPUs to reduce the individual latencies. This makes the resource allocation harder, as it needs to take into account overheads of parallelism, data migration, etc. Also, scalability limitations, such as restricting query execution or its certain stages, e.g. aggregation, to a single node, allow for higher performance of a more concurrent workload.

Our experiments show that using multiple connections to handle concurrent workloads boosts performance, often dramatically, across the architectures supported by Tableau. Obviously, the positive effect is observable if idle resources are available and can be utilized. Note that some resources can be shared easily, such as CPU, and some not, such as memory. In addition, any bottleneck in the system can slow down the entire pipeline. For example, in certain databases, session-local DDL operations for temporary structures take a high-level lock. We have also observed higher contention on disc controllers to affect the overall performance.

4. Tableau Data Engine

The Tableau Data Engine (TDE) is a read-only column store implemented by Tableau. It has been described in [1] and [2]. Most features described in the above papers have been shipped before Tableau 9.0, except for the new performance improvements covered in Sect. 4.2 and 4.3. The TDE is used both in the client environment with Tableau Desktop and in the server environment with Tableau Server. This section first gives an overview of the TDE, and then discusses these improvements.

4.1 TDE Overview

The TDE is designed to support the Extract scenario: extracting a fraction of or the entire dataset from the lively connected data source for offline analysis. Popular use cases include reducing the workload to a live database, getting better performance for analysis, making a snapshot and without worrying about data change, working on an airplane, etc. This scenario imposes a few requirements that are not fully satisfied by existing open-source and commercial systems: support fast execution of complex analytical queries, column level collated strings, 32-bit hardware with limited resource, single database file, and small installation footprint. The single database file is an important convenience feature for users to move, share, and publish the data. The TDE is also used in the Tableau Server environment, where it is deployed as a cluster of nodes through either shared-nothing or shared-everything architecture. The TDE's design is largely influenced by the extensive research work around MonetDB [3]. And it has a few advantages over existing systems:

1. The TDE supports all popular platforms, such as Windows, Mac and Linux. It also supports both 32bit and 64 bit systems.
2. As an analytical engine, the TDE supports column level collated strings.
3. The TDE models column decompression using regular logical operators. Special logical operators are also introduced to utilize the compression to speed up the query execution. This provides a uniform view for all the operations in the TDE.
4. The TDE has a very small installation package. It is easy to download and install.
5. The TDE is able to run as a client application on cheap hardware and limited resource. It can also be deployed as a shared-nothing or shared-everything cluster in the server environment.
6. The TDE is specially tuned for interactive analysis of complicated analytical queries. Features such as fact table culling directly address this kind of a workload

7. The TDE is able to compact a database into a single file.

This section gives an overview of the TDE. We focus on the following aspects of its architecture:

- the storage layer
- the compiler and optimizer
- the execution engine.

We also discuss how TDE is deployed in both the client and server environment.

4.1.1 Storage Layer

Like most database systems, the TDE has a three-layer namespace for logical objects in a database: schema, table and column. It has a simple on-disk storage layout, which makes packing the entire database into a single file easy. The three-layer namespace matches on-disk storage layout: each database is a top-level directory that contains schemas. Each schema is the next level directory that contains tables. Each table is a directory that contains columns. Each column is a set of related files. The metadata is stored in the reserved SYS schema. This directory is packaged into a single file once created.

The TDE implements column-level compression. Each column stores uncompressed fixed-width data, or compressed fixed length or variable-length data. The TDE uses a dictionary-based compression. When data is compressed, the fixed tokens are stored in the original column. Each compressed column also owns an associated dictionary for the original fixed length (array compression) or variable length (heap compression) values. The TDE also employs lightweight compression storage format, such as run-length or delta encodings, for storing fixed-width data. This form of compression is called *encoding* in the TDE. The dictionary-compression is visible to the outside of the storage layer while encoding is a storage format that is typically invisible outside this layer.

Unlike most analytical databases, the TDE supports column-level collated strings. This is important for keeping behavior in the live and Extract scenario in Tableau consistent, as most live databases do support column-level collated strings.

4.1.2 Compiler and Optimizer

The TDE uses a logical tree style language called Tableau Query Language (TQL). It supports logical operators present in most databases, such as TableScan, Select, Project, Join, Aggregate, Order, and TopN. It has a classic query compiler that accepts a TQL query as text and translates it into some logical operator tree structure. The compiler does parsing, syntax checking, binding and semantic analysis. In addition, the compiler also performs classic rewrites of the tree, for example, expressing SELECT DISTINCT as a GROUP BY query. It also rewrites the tree to model the decompression as a normal join and introduces special logical operators to speed up query execution by utilizing the compression. This will be further discussed in Sect. 3.3.

The TDE optimizer is a rule-based optimizer. It derives properties, such as column dependencies, equivalence sets, uniqueness, sorting properties and utilizes them to perform a series of optimizations, such as filter and project push-down/pull-up, removal of unnecessary joins, removal of unnecessary orderings, common sub-expression elimination etc. The TDE optimizer is specially optimized for interactive analysis of complicated analytical queries. For example, removal of the fact table from a join is critical for performance of domain queries, frequently sent by Tableau.

4.1.3 Execution Engine

The TDE execution engine is based on the Volcano execution framework. It has implementations for a set of operators such as filter, join, etc. Each query is compiled into an execution tree by the compiler and optimizer. The query tree is executed by iterating over all the rows of the root of the tree. Operators are of two types: streaming, and stop-and-go. The first type of operators can immediately generate output rows while consuming input rows. The second type of operator needs to consume the entire input before generating any output.

4.1.4 TDE Deployment in the Client and Server Environments

The TDE is a part of any Tableau Desktop deployment and gets executed as a separate process in its process tree. When the TDE is used in the server environment, it is deployed either as a shared-nothing architecture or shared-everything architecture. Each node in the cluster is a separate TDE program. In the shared-everything architecture, storage is shared across all the nodes. A load balancer dispatches queries to different nodes in the TDE cluster.

In both cases Tableau, treats the TDE like any other supported database. It pre-processes query batches, compiles queries in TQL and executes them against the engine.

4.2 Parallel Query Execution

The TDE is able to run different queries in parallel on a single host. For a single query, however, the first version of the TDE did not use multiple parallel threads. In Tableau 8.1, we introduced a limited implementation that targeted expensive calculations in filters and projections. In the current release, we redesigned the parallel execution in the TDE to use parallel scans, which provides a much greater degree of parallelism. This section discusses the implementation of parallel execution.

4.2.1 Parallel Execution Runtime Operators

Like most database systems, the TDE execution engine uses the Exchange operator [4] to handle the parallel part of the query plan. The TDE has an implementation of the Exchange operator that is able to take N inputs and produce M outputs. It has a capability to repartition the data and can preserve the order of the input if needed.

Besides the Exchange operator, the TDE implements parallelism by means of additional operators: SharedTable and FractionTable. SharedTable is used to share access to a table across multiple threads and handles synchronization. FractionTable enables the TDE to read the table in parallel, since each fraction can be read by a separate thread.

4.2.2 Parallel Plan Generation Algorithm

Like VectorWise [9], the TDE optimizer takes a serial plan, and transforms it into a parallel plan by determining the degree of parallelism, and inserting Exchange operators [4] into the tree. In Tableau 9.0, we limited the usage of the Exchange operator to only support N inputs and one output. This means that the parallel plans we consider do not support repartitioning. The only data partitioning happens in TableScan. We also do not utilize the order preserving capability in the Exchange operator. In the

coming releases, we will explore how repartitioning and order-preservation can benefit the performance of Tableau's workloads.

For brevity, we first describe the parallel plan generation for the case of a query without any joins. The algorithm to parallelize such a serial query follows the bottom-up scheme:

1. Leaf nodes are TableScan operators. At the TableScan operator the optimizer looks at the metadata, and makes a decision to partition the table into N fractions, where N is at least one.
2. If the parent is a flow operator such as Select or Project, the parent inherits the degree of parallelism from the child.
3. If the parent is a stop-and-go operator, such as Aggregate, Order or TopN, the optimizer inserts an Exchange operator between the child and the parent.
4. If the root has a degree of parallelism that is larger than one, the optimizer inserts an Exchange operator to close the parallelism.

Figure 3 shows some parallel plan examples generated by the algorithm.

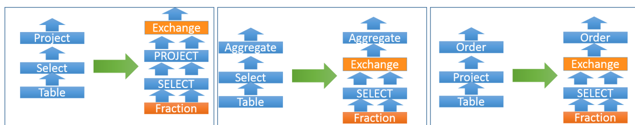

Figure 3: Parallel Plan Examples

We notice that because the Exchange operator has only one output, once the optimizer decides to put an Exchange operator to close the parallelism, the entire tree above the Exchange operator runs in serial. This is a limitation that we plan to explore and alleviate in the next release.

Determining the degree of parallelism in TableScan relies on metadata, such as data volume stored in a table. The TDE also has a cost profile for different supported elementary functions. The cost constants are obtained by empirical measuring. Certain operations, such as string manipulations, are much more expensive than others, even though the engine employs vectorization in expression evaluation. The cost profile is used to determine how expensive an expression could be. This further affects the decision of the parallelization.

Since joins in Tableau are usually between the fact table and multiple dimension tables, the TDE uses a left-deep tree to represent the joins, where the leftmost leaf table is the fact table. The TDE's execution engine processes the join by building a hash table for the right-side input, and probing the left-side input for matches.

Extending the parallel plan generation algorithms to handle join is straightforward and can be described as follows:

1. The left sub-tree of the join participates in the main parallelism
2. The right sub-tree forms a separate and independent parallel unit, and the resulting table is shared between threads.
3. A single hash table is built from the shared table and then shared for every left-hand block to probe.

Figure 4 shows a parallel plan example with join.

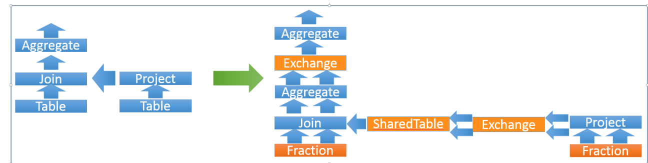

Figure 4: Parallel Plan with Join

4.2.3 *Optimizing Aggregations in the Parallel Plan*

Aggregations are very common in TQL queries. The simple parallel plan generation algorithm puts an Exchange operator to close a parallelized block and then applies the aggregation function serially on top of the output of the Exchange operator.

This approach can be improved by using local/global aggregation. The basic idea is simple: we apply the aggregate on each partition in parallel, let the Exchange operator to merge those partially aggregated results, and again apply the aggregate on top of the output of the Exchange operator. The local/global aggregation approach reduces the total size of data that goes into the Exchange operator. The same approach can also be applied to the TopN operator. Figure 5 shows how the parallel plan looks like when local/global aggregation is applied.

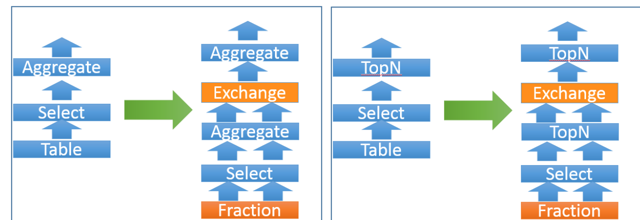

Figure 5: Parallel Plan with Local/Global Aggregation

Even if we apply local/global aggregation to the query tree, aggregation is still a serialization point. We can do even better by removing the Exchange operator and the global aggregation completely from the tree and keep the plan in parallel. This requires a more careful partitioning of the data. If we can guarantee that all the records of each unique group appear in one and exactly one partition, the global aggregation is redundant. In the remaining part of this section, we explore how to make this possible.

Typically, database systems use three types of partitioning: hash partitioning, range partitioning, and random partitioning. Random partitioning means that there is no constraint on which record goes to which partition. The system just sends records to partitions in an arbitrary fashion. The TDE is able to perform random partitioning for any table. Most tables are sorted according to one or more columns, and for those tables, the TDE is also able to perform range partitioning on the major sort. Performing range partitioning sometimes is important because interesting orders [13] on columns involved in the partitioning may be helpful in optimizing the query.

In this section, we discuss the conditions when the global aggregate and the associated Exchange operator can be removed, under the constraint that the TDE is either able to perform a random partitioning, or a range partitioning on a sorted table. A sufficient condition for removing the global aggregation is: if there exists a subset of GROUP BY columns such that a permutation of these columns is a prefix of the sorted column list,

a range partition is able to be delivered for removing the global aggregation and the associated Exchange operator. We briefly prove this through the following lemmas:

Lemma 1: A sorted table on an ordered set of columns {c_1, .. , c_n} can be range partitioned on a prefix of the ordered set.

Proof. If a table is sorted on an ordered set, it is also sorted on any prefix of the ordered set. Therefore, we are able to range partition on any prefix. ■

Lemma 2: If a table is range partitioned on a subset of group by columns, the partition guarantees that each unique group appears in one and only one of the partitions.

Proof. If a table is range partitioned on a subset of group by columns, the partition guarantees each unique group with respect to the subset appears in one and only one of these partitions. Since any group according to the full column set is a subset of some group according to the subset, all the records of such a group must be in one and only one of the partitions. ■

Lemma 3: If some permutation of a subset of the group by columns is a prefix of a sorted table's ordered column set, there exists at least one range partitioning scheme that allows aggregates to be computed in parallel.

Proof. Lemma 1 says that we can range partition on a prefix of the ordered set. Lemma 2 says that range partitioning on a subset of group by columns enables parallel execution for aggregates. By combining the two lemmas, Lemma 3 is proved. ■

The TDE today implements range partitioning by allowing the Aggregate operator to push down the partitioning requirements. TableScan can then utilize this information along with the metadata that indicates whether the table is sorted on one or more columns, to make a decision whether a range partitioning is appropriate. Multiple layers of Aggregates may accumulate more than one partition requirements along the tree from the root to the leaf. Inferring the relationship and finding a best partition is an interesting topic. In Tableau 9.0, we did not explore this problem space. Instead, the TableScan only gets the partition requirements from the nearest Aggregate operator.

Data skew and low cardinality are other concerns related to range partitioning. Namely, if the data is skewed or if the partition key has very low cardinality (e.g. partitioning on gender), range partitioning may be slower than the local/global aggregation approach. Therefore, range partitioning in the TDE is applied conservatively today.

4.2.4 Interaction between Parallelization and Other Query Optimizations

The TDE query optimizer performs a series of optimizations. Some of them are affected by the introduction of parallelism to the plans. For example, the optimizer derives sorting properties that are used in certain rewrites. One of the usages is to determine whether a streaming implementation can be used for an aggregate function. Strictly speaking, if the data is grouped according to the group by columns, streaming aggregates can be applied. The TDE only tracks sorting properties but sorting is a sufficient (but not a necessary) condition to satisfy the grouping requirements.

The Exchange operator disturbs the sorting properties. Choosing between a streaming aggregate while the query is running in serial, and the parallel plan while we need to use a normal aggregate (currently based on hashing only in the TDE) is a cost-based decision. One can also consider variations of the parallel plans with resorting or order-preserving Exchange but both strategies performed badly on our testing workloads. In general, parallelization introduces more alternatives into the search space and the current rule-based query optimizer does not explore all the additional ones.

4.3 Leverage Encoding for Query Execution

The TDE optimizer is able to utilize the encoding information to speed up query execution. Such techniques have been discussed in [2]. The implementation has now become part of the Tableau 9.0 release. This section gives a quick overview of one specific technique and is largely a summary of Sect. 5.2 from [2].

For a run length encoded column, the optimizer can generate an IndexTable, which consists of three columns: *value*, *count* and *start*. The IndexTable can then be joined back to the main table on a range predicate:

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

While this join by itself is not interesting, combining with the operator pushdown allows the optimizer to push a filter condition on the run length encoded column to the IndexTable. As a result, it produces a much smaller output that normally only contains a few rows. This join then significantly reduces the output of the TableScan. Furthermore, since this is a special join, we implement the join that translates the range specifications directly into disk accesses. This approach allows us to express range skipping simply as a join in the query plan. Parallel execution is implemented by distributing the result from the IndexTable across multiple threads. These threads then scan different ranges of the same input table.

Given the parallel plan introduced in Tableau 9.0, the specific approach described above does not always make the query execution faster. Although it reduces the total amount of data to be read from the disk, it may also reduce the degree of parallelism. Furthermore, this approach can introduce data skew among different threads in the parallel execution. We are looking into how to better choose the best plan among different alternatives. We are also looking for different ways to utilize the encoding for faster query execution.

4.4 Shadow Extract for Text and Excel Files

Tableau is able to connect to a wide range of data sources, including text and Microsoft Excel files. Both text and Excel files are highly popular file formats for storing data.

In the past Tableau used Microsoft Jet/Ace drivers to query text and Excel files. This approach had a number of drawbacks, such as lack of portability to other operating systems and a 4GB parsing limit. Furthermore, running analytical queries over these data sources was inherently slow because the system had to parse the file for every query.

Shadow extracts have been introduced to speed up the query execution and overcome the Jet limitations. When a text or excel file is connected, Tableau extracts the data from the file, and stores them in temporary tables in the TDE. Subsequently, all queries are executed by the TDE instead of parsing the entire file each time. This greatly improves the query execution time, however, we need to pay a one-time cost of creating the temporary database. Last but not least, the system can persist extracts in workbooks to avoid recreating temporary tables at every load.

In order to effectively extract data from text and Excel files, and overcome the Jet/Ace limitations, Tableau uses an in-house parser for parsing text files and LibXL [6] for parsing Excel files. These parsers are both more efficient, do not have the 4GB limitation, and are cross-platform. The text parser accepts a schema file as additional input if one is available. Otherwise, it attempts to discover the metadata by performing type and column name inference.

5. Improving Interactivity of Tableau Server

Tableau Server enables users to share their visualizations in an organization. Tableau 9.0 has improved the interactivity of Tableau Server significantly. In this section, we first discuss the limitations of using Tableau Server for sharing. We then present the Data Server component, which was introduced to address these problems. The rest of this section discusses the performance improvements in Tableau Server for interactivity.

5.1 Sharing Visualizations in Tableau Server

Tableau users share their data visualizations by publishing them to Tableau Server as collections of visualizations, called workbooks. Users interact with these published visualizations via a Web browser. The visualizations may rely on either live connections to data sources or data extracts. Extracts can be automatically refreshed by Server to prevent stale data.

Except for their connections to live data sources, Tableau workbooks are self-contained. Customized calculations and fields are defined within the workbook. TDE extracts are contained in the workbook.

Bundling all data source definitions and extracts within a workbook makes sharing a workbook simple, but prevents other workbooks from sharing the contained calculations and extracts. Users wishing to use the same calculations and fields defined in a published workbook must manually copy the definitions into their own workbooks. If a calculation needs to be modified, all workbooks containing the calculation must also be updated.

Similarly, TDE extracts must be generated and included in each workbook that references it. If hundreds of workbooks all use the same large extract, considerable disk resources are consumed by redundant data. Refreshing the workbooks' extracts daily to prevent stale data incurs a significant and redundant load on the underlying database.

5.2 Tableau Data Server Overview

The Tableau Data Server is a part Tableau Server that reduces the overhead of sharing calculations and extracts across workbooks. Data Server also allows filters to be applied to a published data source to restrict individual users' access to the data. For example, an individual salesperson may only be able to see customers in their region, while their manager can see customers in all regions.

Users publish data sources that can be leveraged, without duplication, by multiple workbooks to Data Server. By publishing a data source to Data Server, a complex calculation in a data source can be defined once and used everywhere. Categorical bins and multi-dimensional sets of thousands of constants can be manually defined once. Modifications to a published data source affect all visualizations that refer to it.

TDE extracts can be published with a data source. Instead of 100 workbooks with distinct copies of the same extract, a single extract is created. Refreshing a single extract daily -- rather than all copies of it – significantly reduces the query load on the underlying database.

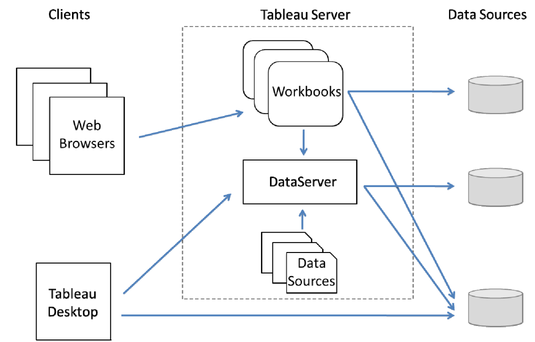

Figure 6: Data Server in the Tableau eco-system

Figure 6 depicts Data Server in the larger Tableau eco-system. Clients can directly connect to databases or connect to data sources published to Data Server, which acts as a proxy between clients and the underlying database.

When a client connects to a published data source, it receives metadata (e.g. schema) about the published data source from Data Server. The client populates its data window with this information. As the user drags fields on to the visualization, queries are dispatched from the client to Data Server.

Data Server parses the query into an internal representation, optimizes it and generates the query for the specific underlying database. Databases have different capabilities and the query optimization and generation process takes these capabilities into account. The generated query is evaluated against the underlying database and the retrieved results are returned to the client for display.

5.3 Improving Interactivity of Published Data Sources

Data Server as any other proxy adds on a certain overhead. However, other than imposing data permissions, there is conceptually no reason why proxied interactions with underlying data sources would be different from the ones against equivalent direct connections.

In the past, in certain cases the queries from Data Server could be less optimal. This is because Data Server was using a separate optimization pipeline from the one in Tableau Desktop (Sect. 3.1). In Tableau 9.0, these pipelines got unified, which allows both applications to take advantage of the same rich optimizations. Another shortcoming was related to managing a remote state. Namely, when a client directly connects to a database, Tableau creates temporary data structures on the database to improve interactivity. For example, a filter on a large cardinality database field may be stored as a temporary table on the database. Instead of issuing a query with a very long and complicated filter to the database, the temporary table is used in the query.

We have introduced temporary tables to the Data Server to improve the user experience for published data sources. The temporary data structures provide two different performance improvements: (1) reduced network traffic between the client and the Data Server if a temporary data structure is used repeatedly in

subsequent queries, and (2) improved query execution times on the database.

When a client connects to a published data source on Data Server, the Data Server establishes a connection to the underlying database and determines if it supports the creation of temporary tables. This information is conveyed back to the client with the data source metadata.

As fields are dragged to the visualization, the client issues a series of requests to Data Server to create and populate a temporary table, causing the Data Server to construct an in-memory temporary table. Subsequent queries from the client to the Data Server may reference this temporary table. On the Data Server, the temporary table's definition is incorporated during query optimization, compilation, and evaluation. In some cases, the query may be evaluated without interacting with the underlying database.

If the generated query must be evaluated by the underlying database, the Data Server will create a temporary table on the database. The generated query is modified to use the temporary table on the database and executed. If the Data Server fails to create a temporary table on the database, the query is rewritten to produce a query that can be evaluated without it.

5.4 State Management

Temporary table state is maintained in two different places in Data Server: in memory and on the underlying database. In both cases, this state is maintained while the client connection to Data Server remains active; it is reclaimed when the connection is closed or expired due to inactivity.

To alleviate the in-memory cost of temporary tables, temporary table definitions are shared across client connections. These definitions are updated as clients create and drop temporary tables. The definitions are removed when all references to them are removed.

If desired, in-memory temporary tables on Data Server can be disabled. While network traffic between the client and Data Server will increase, users will still benefit from improved query execution times on the underlying database.

6. RELATED WORK

The TDE is a column store for analytical queries. C-store [7] is a column store that has been commercialized as Vertica. MonetDB [3, 8] from CWI is another column store that has wide impact. Vectorwise [9] is the commercialized MonetDB/X100 [3] project that has an Ingres front-end and many additional improvements. Also, SAP Hana [11] is a column store that supports both OLTP and analytical queries.

A parallel database has a long history of research and commercial development since early 1990s. It is beyond the scope of this section to give an overview of this area, for details see [10]. The TDE's execution model is based on the Volcano system [4]. In particular, it uses the Exchange operator to perform parallel execution. Vectorwise's parallel optimizer [12] provides some good data structure to track the requests from the parents to the children and the delivery from the children to the parents. The TDE's parallelizer has largely borrowed the notations from Vectorwise's optimizer. Property derivation has a long history. System R [13] pioneered this technique by tracking the ordering information for intermediate query results. Partition property derivation was first discussed in [5].

Eliminating redundant operations, such as joins has been widely studied [14, 15, 16, 17, 18]. In particular, elimination of redundant joins in [14, 15] is based on the concept of tableaux and is only able to treat a sub-class of relational algebras. [16] enhanced the tableaux approach by using so-called tagged selection tableaux. Removing redundant joins in queries involving views has been studied in [17]. The approach uses functional dependencies and source level transformations of SQL to remove redundant joins. Join culling is widely applied in Tableau and the TDE to improve query execution latency for better interactive experience.

Query matching plays an important role in Tableau's caching infrastructure. It has been widely studied in the database literature, mostly in the context of database views. Giving an overview of the research for database views is beyond the scope of this section. A good survey is provided in [13]. Query containment and query equivalence are two important concepts in the research literature. They enable comparison between different queries. The theory of query containment and equivalence has been widely studied [18, 21, 22, 23, 24, 25]. GMAP [26, 27] pioneered the view matching approach in the database system. In [28], the authors provide a similar algorithm for view matching. A transformation rule based approach for view matching in the SQL Server's optimizer is discussed in [29]. This approach introduces the filter-tree index to dramatically speed up the matching process.

Multiple query processing has been widely studied in the database literature, mostly in the context of multiple query optimization [30, 31, 32, 33, 34, 35]. Much of the work [30, 31, 34] has focused on finding common subexpressions and materializing the results temporarily for sharing among these queries. Researchers also studied multiple query processing beyond the common subexpression detection. For example, Tan and Lu [36, 37] investigated how to schedule query fragments to better share the data in memory and across multiprocessors. Pipelining the data between processes to share the same data is studied in [38]. Furthermore, scheduling the queries carefully using a middleware to improve sharing between queries has been studied in [39].

7. FUTURE PLANS

Tableau constantly adds new data sources and types of analysis, which often requires implementation of dedicated performance features. In particular, we are planning to add end-to-end support for federated and semi-structured data sources, where appropriate optimization techniques, efficient extraction and management in Data Server play important roles.

Regarding the query processing pipeline we plan to apply more multi-query processing techniques to take advantage of sophisticated commonalities between queries in a batch. That fits well with incorporating more classic optimizations to improve our plans and generated remote queries. Furthermore, in order to alleviate frequently substantial local post-processing, we are going to explore multi-level caching similar to [44].

Substantial sizes of federated datasets and rapidly growing popularity of our SaaS platform put more pressure on the Tableau Data Engine to process larger extracts. Therefore, we are considering using data partitioning in a distributed architecture.

Last but not least, both data exploration and dashboard generation could become more responsive if requested data has been accurately predicted and prefetched. Materialization of secondary

structures and prediction approaches such as DICE [46], are good examples in this field.

8. SUMMARY

In this paper we have discussed several performance improvements that contribute to high interactivity of visual data analysis. They are available in the new release of the product, Tableau 9.0.

Most of all, in dashboard generation, we have suggested query batch processing with an initial batch optimization phase, query fusion and concurrent query submission. Using two levels of query caches, intelligent and literal, that can be persisted on the client or distributed in the server environment, further magnifies the responsiveness. Furthermore, performance of the TDE has been improved thanks to addition of parallel plans and index scans for RLE-encoded columns. Last but not least, temporary table support has been added to Tableau Data Server to improve performance of published data sources.

9. REFERENCES

[1] Richard Wesley, Matthew Eldridge, and Pawel T. Terlecki. 2011. An analytic data engine for visualization in tableau. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (SIGMOD '11). ACM, New York, NY, USA, 1185-1194. DOI= http://doi.acm.org/10.1145/1989323.1989449

[2] Richard Michael Grantham Wesley and Pawel Terlecki. 2014. Leveraging compression in the tableau data engine. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (SIGMOD '14). ACM, New York, NY, USA, 563-573. DOI=http://doi.acm.org/10.1145/2588555.2595639

[3] 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.

[4] G. Graefe, "Volcano: An extensible and parallel query evaluation system," IEEE Transactions on Knowledge and Data Engineering, 120-135, 1994.

[5] J. Zhou, P. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In ICDE, 2010.

[6] LibXL http://www.libxl.com/

[7] 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.

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

[9] Zukowski, Marcin, and Peter A. Boncz. "Vectorwise: Beyond column stores."*IEEE Data Engineering Bulletin* 35.1 (2012): 21-27.

[10] Shivnath Babu and Herodotos Herodotou (2013), "Massively Parallel Databases and MapReduce Systems", Foundations and Trends® in Databases: Vol. 5: No. 1, pp 1-104. http://dx.doi.org/10.1561/1900000036

[11] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. IEEE Data Engineering Bulletin, 35(1):28–33, 2012.

[12] Anikiej K. Multi-core Parallelization of Vectorized Queries [dissertation]. University of Warsaw and VU University of Amsterdam, 2010.

[13] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In Proceedings of SIGMOD Conference, 1979.

[14] M. Majster-Cederbaum. Elimination of redundant operations in relational queries with general selection operators. Computing, 34(4):303-323, 1984.

[15] A. V. Aho, C. Beeri, and J. D. Ullman. The theory of joins in relational databases. ACM Trans. on Database Systems, 4(3): 297-314, 1979.

[16] A. V. Aho, Y. Sagiv, and J. D. Ullman. Efficient optimization of a class of relational expression. ACM Trans. on Database Systems, 4(4):435-454, 1979.

[17] Nikolaus Ott, Klaus Horländer, Removing redundant join operations in queries involving views, Information Systems, Volume 10, Issue 3, 1985, Pages 279-288

[18] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operator. Journal of the ACM, 27(4):633-655, 1980.

[19] Halevy, Alon Y. "Answering queries using views: A survey." The VLDB Journal 10.4 (2001): 270-294.

[20] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. 2003. Containment of Aggregate Queries. In *Proceedings of the 9th International Conference on Database Theory* (ICDT '03), Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani (Eds.). Springer-Verlag, London, UK, UK, 111-125.

[21] Chandra A.K., Merlin P.M. Optimal implementation of conjunctive queries in relational databases. In: Proc. Ninth AnnualACMSymposium on Theory of Computing.pp 77–90, 1977

[22] Zhang X., Ozsoyoglu M.Z. On efficient reasoning with implication constraints. In: Proc. of DOOD. pp 236–252, 1993

[23] Chaudhuri S., Vardi M. Optimizing real conjunctive queries. In: Proc. of PODS. pp 59–70, Washington D.C., USA, 1993

[24] Chaudhuri S., Vardi M. On the complexity of equivalence between recursive and nonrecursive datalog programs. In: Proc. of PODS. pp 55–66, Minneapolis, Minn., USA, 1994

[25] Kolaitis P., Martin D., Thakur M. On the complexity of the containment problem for conjunctive queries with built-in predicates. In: Proc. of PODS. pp 197–204, Seattle,Wash., USA, 1998

[26] Tsatalos O.G., Solomon M.H., Ioannidis Y.E. The GMAP: a versatile tool for physical data independence. In: Proc. of VLDB. pp 367–378, Santiago, Chile, 1994

[27] Tsatalos O.G., Solomon M.H., Ioannidis Y.E. The GMAP: a versatile tool for physical data independence. VLDB J. (2):101–118, 1996

[28] Chaudhuri, S., Krishnamurthy, R., Potamianos, S., & Shim, K. (1995, March). Optimizing queries with materialized views. In 2013 IEEE 29th International Conference on Data Engineering (ICDE) (pp. 190-190). IEEE Computer Society.

[29] Goldstein J., Larson P.A. Optimizing queries using materialized views: a practical, scalable solution. In: Proc. of SIGMOD. pp 331–342, 2001

[30] JarkeM. Common subexpression isolation in multiple query optimization. Query Processing in Database Systems, KimW, Reiner DS, Batory DS (eds.). Springer: Berlin, 1985

[31] Park J, Segev A. Using common subexpressions to optimize multiple queries. Proceedings of the 4th International Conference on Data Engineering. IEEE Computer Society: Washington, DC, 1988; 311–319.

[32] Sellis T. Multiple query optimization. ACM Transactions on Database Systems 1988; 13(1):23–52.

[33] Cosar A, Lim E, Srivastava J. Multiple query optimization with depth-first branch-and-bound and dynamic query ordering. CIKM 93, Proceedings of the Second International Conference on Information and Knowledge Management. ACM, 1993; 433–438.

[34] Chen F, Dunham M. Common subexpression processing in multiple-query processing. IEEE Transactions on Knowledge and Data Engineering 1988; 10(3):493–499.

[35] Roy P et al. Efficient and extensible algorithms for multi query optimization. Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. ACM Press: New York, 2000; 249–260.

[36] Tan K, Lu H. Workload scheduling for multiple query processing. Information Processing Letters 1995; 55(5):251–257.

[37] Tan K, Lu H. Scheduling multiple queries in symmetric multiprocessors. Information Sciences 1996; 95(1/2):125–153.

[38] Dalvi N et al. Pipelining in multi-query optimization. J. Comput. Syst. Sci. 2003; 66(4):728–762.

[39] O'Gorman, Kevin, Amr El Abbadi, and Divyakant Agrawal. "Multiple query optimization in middleware using query teamwork." Software: Practice and Experience 35.4 (2005): 361-391.

[40] 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.

[41] http://redis.io/

[42] Lakshman, Avinash, and Prashant Malik. "Cassandra: a decentralized structured storage system." ACM SIGOPS Operating Systems Review 44.2 (2010): 35-40.

[43] https://www.faa.gov/data_research/

[44] Milena G. Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo A.P. Gonçalves. 2009. An architecture for recycling intermediates in a column-store. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD '09), Carsten Binnig and Benoit Dageville (Eds.). ACM, New York, NY, USA, 309-320

[45] Parag Agrawal , Daniel Kifer , Christopher Olston, Scheduling shared scans of large data files, Proceedings of the VLDB Endowment, v.1 n.1, August 2008

[46] Prasanth Jayachandran, Karthik Tunga, Niranjan Kamat, Arnab Nandi. Combining User Interaction, Speculative Query Execution and Sampling in the DICE System. PVLDB 7(13): 1697-1700 (2014)

[47] Kristi Morton, Ross Bunker, Jock Mackinlay, Robert Morton, and Chris Stolte. 2012. Dynamic workload driven data integration in tableau. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12). ACM, New York, NY, USA, 807-816. http://doi.acm.org/10.1145/2213836.2213961

[48] Shaul Dar , Michael J. Franklin , Björn Þór Jónsson , Divesh Srivastava , Michael Tan. Semantic Data Caching and Replacement, *Proceedings of the 22th International Conference on Very Large Data Bases*, p.330-341, September 03-06, 1996