T. Grust et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2019),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2019 313

Domain Query Optimization: Adapting the General-Purpose Database System Hyper for Tableau Workloads

Adrian Vogelsgesang[1] Tobias Muehlbauer[2] Viktor Leis[3] Thomas Neumann[4] Alfons Kemper[5]

Abstract: The Hyper database system was started as an academic project at Technical University Munich. In 2016, the commercial spin-off of the academic Hyper database system was acquired by Tableau, a leader in the analytics and business intelligence (BI) platforms market. As a human-in-the-loop BI platform, Tableau products machine-generate query workloads with characteristics that differ from human-written queries and queries represented in industry-standard database system benchmarks. In this work, we contribute optimizations we developed for one important class of queries typically generated by Tableau products: retrieving (aggregates of) the domain of a column. We devise methods for leveraging the compression of the database column in order to efficiently retrieve the duplicate-free value set, i.e., the domain. Our extensive performance evaluation of a synthetic benchmark and over 60 thousand real-world workbooks from Tableau Public shows that our optimization enables query latencies for domain queries that allow self-service ad-hoc data exploration.

1 Introduction

The Hyper database system was started as an academic project at Technical University Munich. Hyper is developed as a general-purpose database system for transactional as well as analytical workloads, operating simultaneously on one state, embracing the SQL standard and ACID transactional guarantees, while delivering highest performance in both workload classes. In 2016, the commercial spin-off of the academic Hyper database system was acquired by Tableau, a leader in the analytics and business intelligence (BI) platforms market. The machine-generated query workload by Tableau's products greatly differs from the queries in publicly available industry-standard benchmarks that Hyper was initially optimized for [Vo18]. Not only let modern BI tools easily express very complex queries via an intuitive drag-and-drop interface, users of such tools also expect to be able to quickly analyze data of all sizes and of all types. Standardized benchmarks like TPC-H, TPC-C, TPC-DS, the CH-Benchmark, or the SQLite test suite are relevant, but do not represent the

[1] Tableau Software, avogelsgesang@tableau.com
[2] Tableau Software, tmuehlbauer@tableau.com
[3] Tableau Software, vleis@tableau.com
[4] Tableau Software, tneumann@tableau.com
[5] Tableau Software, akemper@tableau.com

doi:10.18420/btw2019-19

observed long tail in both, query and data set complexity. In addition, there is a growing demand to analyze the freshest state of the data as it becomes available and in interactive ways, where each result is the starting point of further questions. As such, it is infeasible to pre-compute results. To meet the low-latency query response time requirements of modern interactive BI, Hyper needed to be adapted to specific classes of queries commonly generated by BI tools like Tableau.

A common class of queries generated by modern BI tools are *domain queries*. In this context, domain refers to the set of distinct values that a column in a relation may contain. The domain of a column or an aggregate of the domain is retrieved in various contexts, e.g., to define filters or to properly size the axes in a multi-dimensional chart. What makes domain queries challenging is the fact that most real-world data sets are not properly normalized or are the result of a query that joined together different but related pieces of information into a single relation. Denormalized schemata inherently lead to duplicate values in columns and cannot efficiently be re-normalized on the fly.

In Hyper's storage layer, relations are horizontally partitioned into fixed-size chunks. Each chunk is again vertically partitioned into columns. Chunks can either be hot, i.e., containing tuples frequently modified by transactions, or cold, i.e., mostly scanned or accessed by point queries. Cold chunks are compressed using light-weight compression techniques and are called Data Blocks [La16]. To guarantee low-latency access to the freshest state of the data, de-duplication of columnar values is only feasible on a per-chunk level, since global de-duplication is inherently non-linear in the size of the data set. In this paper, we introduce optimizations for domain queries in the Hyper system that leverage de-duplication in Hyper's Data Blocks storage format. In particular, we contribute:

- A description of how domain queries can be detected.
- A way to efficiently execute domain queries leveraging the Data Blocks storage format.
- An extensive evaluation of domain query optimization in Hyper on synthetic and real-world data sets, including an analysis of 60,000 workbooks from Tableau Public.

2 Background

2.1 Challenges in the Context of Tableau

Tableau generates SQL queries for the Hyper database systems triggered by user input from and scheduled background tasks.

In this paper, we show how we addressed the two key aspects of the observed SQL workload:

- **Denormalized storage**: Many underlying data models in Tableau uses are flat, denormalized tables. This is due to several factors, including for performance reasons, similar to Blink [Ra08].
- **Machine-generated queries**: In contrast to most standard benchmarks for database systems, the queries issued by Tableau are machine-generated. For a database system, this has both up- and downsides: On the positive side, all queries are following a similar structure. The database can be optimized to detect some of the common patterns in the queries and provide optimized implementations for those queries. On the other hand, the machine-generated queries tend to be more complex than hand-written queries.

2.2 The Data Blocks Storage Format

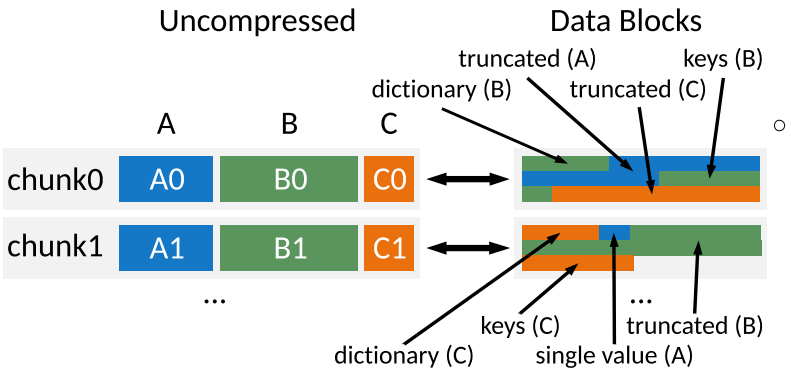

Fig. 1: The DataBlocks storage format

In this paper, we present how to apply domain query optimization to Hyper's primary storage format, Data Blocks. Nevertheless, this optimization is applicable to other data storage formats. The Data Blocks format was originally introduced by Lang et al. [La16]. Figure 1 schematically depicts how a relation with three columns (A, B, C) is represented in the Data Blocks format.

The table is split into multiple chunks of a fixed maximum size, e.g., 2^{17} tuples. Each Data Block is self-contained and stores one or more attribute chunks in a byte-addressable compressed format. The attributes within a Data Block are stored in a columnar fashion. Columns are compressed using lightweight compression methods such as dictionary encoding or single-value encoding. Optimal compression methods are chosen per column within a block based mostly on the expected compression ratio. Note that the same column might be compressed through different compression methods within different blocks. In the figure, e.g., the first chunk of column A is stored using the compression *truncated* while the second chunk is stored as a single value. In addition, each block contains a small materialized aggregate (SMA) [Mo98] storing the minimum and maximum value contained in the block for each column.

The goal of Data Blocks is to conserve memory while allowing a high OLTP and OLAP performance. By maintaining a flat structure without pointers, Data Blocks are also suitable for eviction to secondary storage [FKN12]. A Data Block contains all tuple data, but no metadata, such as schema information.

Compression is employed to optimize processing speed by better exploiting the available memory bandwidth. Smaller storage size is an additional benefit of compression but not the primary goal of Data Blocks. Hence, Data Blocks are compressed using light-weight compression methods which still allow efficient scans as well as point accesses for OLTP workloads. In particular, Data Blocks currently use the following compression formats:

For `dictionary-compressed` columns, a small dictionary containing the domain of the Data Block is built. The tuple values are represented as tightly-packed offsets into that dictionary. The width of the offset values varies between 1 and 8 bytes depending on the size of the dictionary.

If not only the domain size, but also the domain range is small, it is efficient to store the tuple values as relative offset to a base value. The values are `truncated` by subtracting the minimum value within the chunk from each tuple value. Thereby, the storage space for the dictionary can be saved. In other literature, this compression method is also referred to as `frame-of-reference` compression.

In case a column contains the same value for all rows in a chunk, it is `single-value` compressed, i.e., the value is only stored exactly once.

3 Domain Query Optimization

Domain queries are a special type of query issued by Tableau in order to obtain the domain of a column, i.e., the set of different values that occur in the column. The obtained domain is used, e.g., for populating filter lists and drop down menus. In this section, we describe how Hyper leverages duplicate-free structures in the storage layer (e.g., dictionaries, single value compression) to speed up domain queries. Furthermore, we show how we extend the applicability of the domain-query optimization (DQO) to additional queries which are not issued as domain queries, but happen to be applicable for this optimization, too. Finally, we evaluate the impact of DQO both on a synthetic test set and on real-world data.

3.1 Use Case

Tableau allows ad-hoc interactive data exploration. One of the most basic interactions is filtering, which allows users to drill down into a subset of their data. By using filters, users can restrict the set of data being visualized based on arbitrary predicates. For this purpose, Tableau provides the user controls with which they can zoom in on or exclude parts of the

data set. Figure 2 shows some of the available filters. Using the *Order Date* slider, the data can be filtered to a certain date range. The *Segment* and *Region* filters allow to include and exclude specific values from the domain. As can be seen, for all 3 filters Tableau offers all values available in the data set. In order to display this list of available data to the user, Tableau needs to know the domain of the filtered values. For the *Order date* slider, it must know the range, i.e., the minimum and maximum date present in the *OrderDate* column. For the *Segment* selection list and the *Region* drop-down menu, it must know all distinct values from the underlying column, i.e., its domain.

```
SELECT column_x FROM user_data GROUP BY column_x;
```

Listing 1: SQL query issued to populate a drop-down list

```
SELECT MIN(column_x),MAX(column_x) FROM user_data;
```

Listing 2: SQL query issued to populate a drop-down list

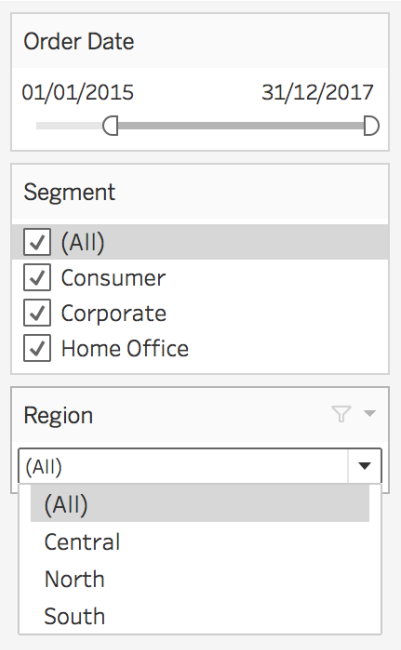

Fig. 2: Filters in Tableau

For both types of filters, Tableau issues queries to the underlying database system. Depending on the type of filter, Tableau issues a query in the form of either Listing 1 (for drop-down menus and selection lists) or 2 (for sliders).

Since the creation of filters in order to drill down into a visualization is part of the daily workflow of data analysts, answering the corresponding SQL queries quickly is important in order to provide a real-time experience. In the research context, Hyper's optimizations were geared towards standard benchmarks such as TPC-H, TPC-DS and TPC-C. Since those benchmarks do mostly not include domain queries, Hyper was initially not optimized for such queries.

In an ideal world, domain queries would not be challenging to process. On a normalized schema, a domain query can usually be answered by scanning a column that consists only of distinct values. Many real-world data models however are denormalized into a single flat table and cannot easily be re-normalized. As such, it is not uncommon to have many duplicates in a column that need to be filtered to return a duplicate-free domain.

3.2 Overview Of Approach

Both query types issued by Tableau to populate its filter controls can be answered without reading all row values, just by looking at the column's domain. For the query in Listing 1 this is obvious since the query explicitly asks for the column's domain. The query in Listing 2 can also be answered using only the column's domain since the result of a `MIN`/`MAX` aggregation is not influenced by duplicated values.

In some cases, domain queries can be removed as part of the logical optimization. If the query from Listing 1 is issued against a column which is known to be unique, the GroupBy operator will be eliminated by Hyper. For denormalized data models, however, initially unique values from the domain tables are duplicated by the joins that computed the denormalized, flat table representation. Hence, this logical optimization is in many cases not applicable. With no possibility to remove the *GroupBy* during logical query optimization, Hyper leverages optimizations during the execution phase to speed up domain queries.

Without DQO, Hyper uses a normal table scan over the queried column. This table scan produces one column value for each row. Doing so it, e.g., expands single value compressed columns by duplicating the column's static value for each row. Each of those tuple values is then passed to a GroupBy operator which removes duplicate values again.

Obviously, the execution time of this query can be improved by skipping the generation of the duplicates in the table scan. In order to speed up this query, we give a hint to the table scan operator and make it aware of the fact that duplicates will be discarded by its parent operator. The table scan then has the freedom to skip producing duplicated tuples altogether. It can take advantage of the underlying physical data representation, e.g., single value compression, dictionaries or run length encoding, to reduce the number of produced values.

Note that the GroupBy operator remains in place even if the underlying table scan is instructed that it should not produce duplicate values. This means that the table scan has the freedom to skip duplicated tuples but is not forced to do so. E.g., if no dictionary is available the table scan is still allowed to produce duplicated tuples. It is even encouraged to do so instead of de-duplicating values on its own. After all, the GroupBy operator constitutes the most efficient de-duplication logic in Hyper. Table scan operators can only be more efficient than the GroupBy operator at de-duplicating values when they are able to exploit the underlying physical storage which would not be accessible to the GroupBy operator.

3.3 Detecting Applicability of DQO

In order for domain query optimization to be applicable, two conditions must be fulfilled:

- The consumer of the scanned tuples must be duplicate-agnostic, i.e., its output must not depend on the number of times tuples with identical values are emitted.
- The table must be stored in a format which allows one to easily suppress the generation of duplicates.

We decided to address the first requirement during logical optimization and set a flag on each table scan operator which is eligible for DQO. Only during query runtime the table scan then inspects the actual storage structures and suppresses duplicates as applicable. This split allows us to keep the strong separation between the logical query optimizer and the storage layer in tact. Hyper's query optimizer is agnostic of the storage layer and storage level knowledge is not being exposed to the logical optimizer. Furthermore, in many cases an ahead-of-time decision on the storage format is not even possible. During the compilation and optimization of prepared queries, for example, the data contained in the scanned columns and the used compression formats during the execution of the query might not yet be available.

We detect if a table-scan is in a duplicate-agnostic context by inspecting the algebra representation of the query plan. We do so after applying all other algebraic optimizations, most notably after constant folding and join reordering. Detecting domain queries after constant folding allows us to apply DQO in more cases since the algebra tree was already simplified and, e.g., tautological selection predicates are already removed.

Detecting domain queries only after join reordering comes with one potential drawback: The cardinality estimates used for join reordering are unaware of the potential cardinality reduction achieved through DQO. Hence, the join optimizer might pick a sub-optimal join order. We made this design decision in the sake of robustness: Even if we detect the applicability of DQO on the logical level, it is not yet clear if this can be exploited on the physical level. Hence, we prefer to use the non-DQO-aware cardinality estimates for join reordering since those estimates represent the worst-case cardinalities which we will experience if DQO cannot be applied, e.g., due to the absence of adequate physical storage structures.

The easiest set of conditions to identify either of the two queries shown in Section 3.1 would be:

1. The top level operator must be a GroupBy operator.
2. Its child operator must be a table scan.
3. The GroupBy operator must either a) be a grouping on only one column from the base table and contain no aggregates, or b) contain only `MIN`/`MAX` aggregates applied to one column from the base table.

While those conditions match both domain queries shown so far, they limit the applicability of DQO unnecessarily. Hence, we went for a more general approach to detect domain

queries: The optimizer tracks for each algebraic operator in which context it will be executed. We distinguish between two different types of contexts:

An operator in a **duplicate-sensitive context** needs to produce all instances of duplicated tuples. In contrast, in a **duplicate-insensitive context** the operator can decide to skip instances of duplicated tuples.

Duplicate-sensitivity is propagated top-down within the query tree. Most operators such as σ, $\bowtie$, Π just pass the duplicate-sensitivity down: if the operator itself occurs in a duplicate-insensitive context, all its inputs are also duplicate-insensitive and vice versa. Some operators, such as the set variants of `UNION`, `INTERSECT` and `EXCEPT` always evaluate their inputs in a duplicate-insensitive context. Another example for this case are semi-joins and anti-joins which pass down the duplicate sensitivity for one side while unconditionally evaluating the other input in a duplicate-insensitive context.

For domain queries as issued by Tableau, the most important operator is the GroupBy operator Γ. The GroupBy operator determines the input duplicate sensitivity independently of its output duplicate sensitivity. It only takes into account the computed aggregates. If all aggregates are duplicate-insensitive, the input operator is in an duplicate-insensitive context. For that purpose, we annotated all duplicate-insensitive aggregation functions, among others: `MIN`, `MAX`, `BOOL_OR` and all `DISTINCT` variants of aggregate functions such as `COUNT(DISTINCT ...)`.

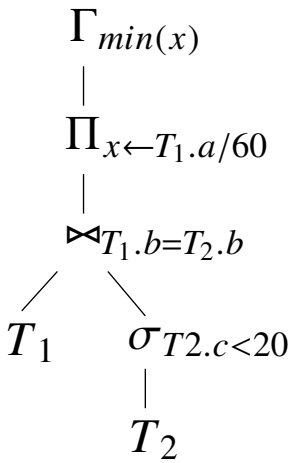

Fig. 3: Algebra tree of a more complex domain query

Figure 3 contains an exemplary algebra tree. The top-level Γ is evaluated in a duplicate-sensitive context as the result of a SQL query is by definition a multi-set and hence duplicate-sensitive. Since the *GroupBy* only computes a `MIN` aggregate, its result does not depend on the absence or presence of duplicated input tuples. Hence, the Π operator is evaluated in a duplicate-insensitive context. The duplicate insensitivity is passed down to the join which can be rewritten into a semi-join given that no columns from its right side are accessed by the upper parts of the plan. Independent of replacing the join by a semi-join, the duplicate insensitivity propagates down through the whole tree. Hence, both table scans can skip duplicated tuples.

However, the scan on T_1 cannot benefit from DQO. Two columns from T_1, a and b, are accessed. Data Blocks always compress columns individually and there is no data structure which allows to de-duplicate (a, b) tuples. Other storage formats might contain data structures such as dictionaries spanning multiple columns and hence could benefit from DQO. We are not aware of any such storage format which is widely adopted.

For T_2, the query also accesses multiple columns (b and c). In this case, the selection is a SARGable predicate and the query still benefits from DQO thanks to the stored SMA synopses (see Section 3.4).

If a table scan occurs in a duplicate-insensitive context, a flag on that table scan is set. This flag will be picked up during execution of the compiled query. Except for this, the operator tree stays unchanged.

```
SELECT TITLE_CASE(c2s.segment_name)
   FROM orders
     JOIN customer_to_segment c2s
     ON orders.customer_id = c2s.id
   WHERE customers.order_date > '2015-01-01'::date
   GROUP BY TITLE_CASE(c2s.segment_name)
```

Listing 3: Complex domain query generated for the Segment filter from Figure 2

Note that Tableau actually issues such complicated queries. Listing 3 shows a SQL query which translates to a query tree of the shape shown in Figure 3. In this exemplary scenario, the `TITLE_CASE` function is issued since the Tableau user created a custom computation to cleanup the inconsistent casing of the segment names. The Join was added by the customer using Tableau's data modelling capabilities to map a customer to the corresponding segment. The `WHERE` clause is generated since the Segments list is implicitly dependent on the selected range of order dates. The Segment list only shows segments for which there exist orders in the selected date range. Thereby, it is ensured that users cannot accidentally filter down the input data so that no more data points qualify.

3.4 Efficient execution of domain queries

After detecting a domain query, it is up to the table scan to exploit the optimization potential of the physical storage format. If a DQO was detected, i.e., if the consumers of the table scan ignore duplicates, the table scan operator is free to suppress duplicate values. However, the table scan is by no means forced to do so. It can just as well produce partially de-duplicated tuples or just the original tuples, since its parent operator stays unmodified and still eliminates duplicates which might get produced.

Figure 4 shows the unoptimized execution of a simple domain query against the Data Block format. The GroupBy operator receives its input tuples from each individual block. The individual tuple streams are implicitly unioned before passing them to the aggregation operator. In the example the first block is compressed using a dictionary. When scanning this Data Block, Hyper looks up every token from the data stream in the dictionary and passes the resulting values to the GroupBy operator. The second Data Block contains only one distinct value and this value is stored using single-value encoding. When the table scan scans this block, it duplicates this single value once for every row stored in the Data Block,

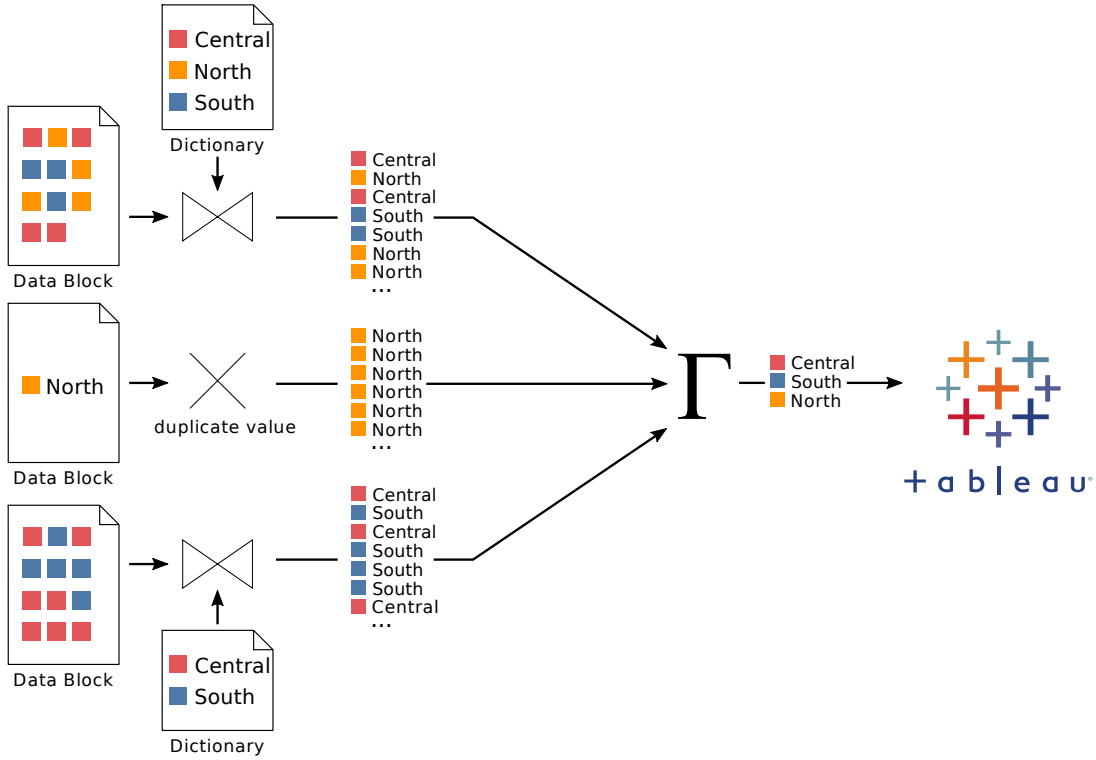

Fig. 4: Unoptimized execution of a domain query against Data Blocks with different compression methods

i.e., 2^{17} times. The third block of the column is stored using a dictionary again. This time however, the block does not contain any tuple with the region being *North*. Accordingly, the corresponding dictionary does not contain an entry for it. All in all the Γ operator receives 3×2^{17} tuples. It eliminates all duplicates and produces the domain consisting of only 3 tuples. Those 3 tuples are then sent to Tableau.

Comparing this to the optimized execution shown in Figure 5, the benefit from domain query optimization becomes obvious. Thanks to DQO, the Γ operator receives only 5 tuples instead of approximately 40 thousand tuples. Since every Data Block is a self-contained unit, there is no way to leverage the Data Block format to de-duplicate values across block boundaries. Hence, there still exist duplicated values which needs to be de-duplicated by the Γ operator.

Within each block, as many duplicates as efficiently possible are suppressed. For each block, the applied strategy depends on its compression method. For dictionary-compressed blocks (the first and the third block in Figure 5 the dictionary is scanned instead of the individual tuple values. For single-value-compressed blocks (the second block in our example), the duplication step is simply skipped and instead only the one single value is generated.

In addition, to those two compression methods, we also implemented DQO for frame-of-reference-compressed data if the offset from the reference value is stored using 1 byte. For

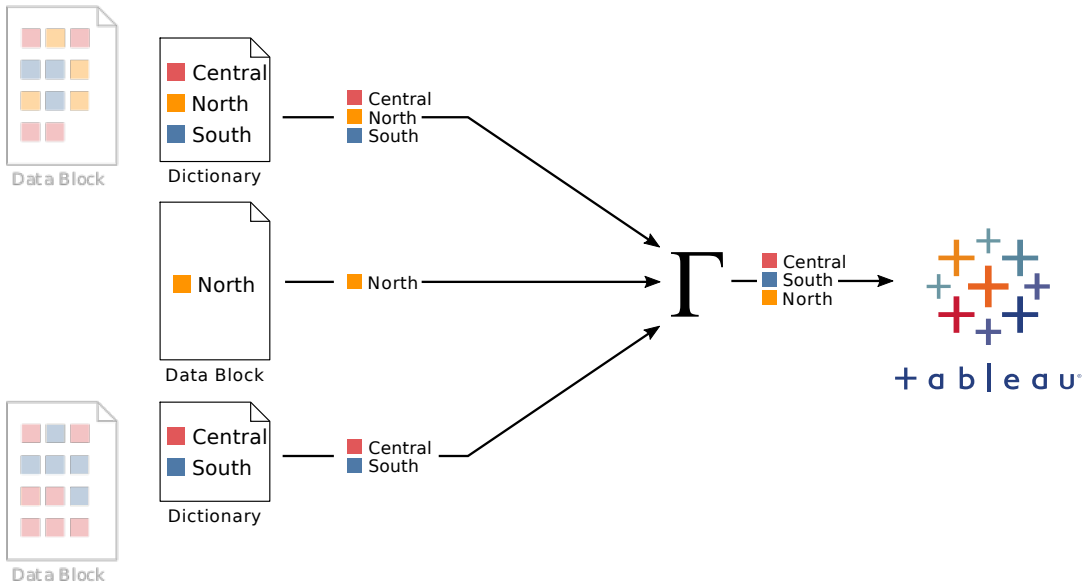

Fig. 5: Optimized execution of the query in Figure 4

this compression type, the values are de-duplicated on the fly: A small bit set is used to mark values which were already seen. Initially, all bits are set to false. Before passing a value to the parent operator, the corresponding bit in the bit set is checked. If it is set, the same value was already seen before and the duplicated value is skipped. Otherwise, the value is seen for the first time and gets produced. In this case. the corresponding bit in the bit set is set so that additional duplicates of the same value are suppressed.

We mentioned before that table scan operators should not do de-duplication since the GroupBy operator already provides the most efficient general purpose de-duplication algorithm. However, for 1-byte-offset-compressed blocks, this general guideline does not apply. Since the offset is stored in 1 byte, there are at most 256 distinct values among the 2^{17} rows of the Data Block. Therefore, it is known upfront that the de-duplication will remove a significant percentage of tuples. The upstream Γ operator implements a general purpose hash based algorithm. In this case, however, we know that all values are within a dense range of 256 integer values. We can exploit this knowledge and outperform Γ by using a small bit set instead of a more heavy-weight hash table.

For the other flavors of frame-of-reference compression, i.e., for 2-byte and 4-byte offsets, it is not clear upfront that de-duplication using a byte array will be helpful. The corresponding bit sets would require 8KB and 512MB of RAM, respectively. Since it is not clear if the additional effort to de-duplicate the values on the table scan level would pay off, we do not de-dupicate values at the table scan level for frame-of-reference encoding with 2 and 4 byte offsets.

Furthermore, the Data Block format allows to evaluate SARGable additional restrictions on

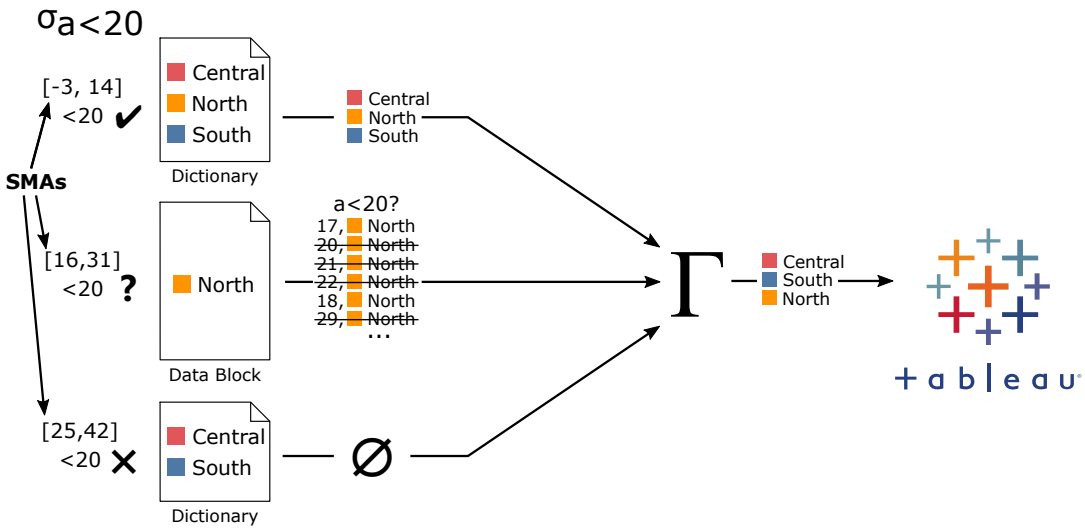

Fig. 6: Optimized execution of a domain query with an additional predicate

columns other than the column whose domain is being queried. By doing so, we allow the query from Listing 3 to also profit from DQO.

Figure 6 shows the evaluation of a domain query The restriction is first evaluated against the SMA of the restricted column. In case, this comparison yields that the condition always evaluates to true for all values in the, the block's domain is scanned scanned instead of the individual tuples. If the SMA indicates that the condition evaluates to false for all values in the block, the whole block is skipped. Only if the evaluation of the restrictions on the SMA is inconclusive, the block must be scanned in a normal fashion, i.e., tuple-by-tuple. The fallback to a normal scan is done on a block-by-block basis. Thereby, even if individual blocks cannot take advantage of DQO, other blocks can still profit from it.

In the special case of `MIN`/`MAX` aggregates, we can even go a step further and directly use the minimum and maximum values from the SMA.

3.5 DQO-aware selection of compression method

Since not all available compression schemes support domain queries well, we also adjusted the heuristic for picking the compression. Earlier, the compression method was chosen purely based on the size of the resulting compressed data. In general, this heuristic also aligns with scan performance: Incidentally, the compressions providing better compression ratios, also provided the better scan performance. With DQO, this statement no longer holds true, though. E.g., Frame of reference compression with 2 byte offsets is smaller than dictionary compression with 2 byte token indices since dictionary compression needs additional space to store the dictionary. On the other hand, domain queries on dictionary-compressed data

are more efficient than the same query on frame-of-reference-compressed data. While one can simply scan the dictionary to answer a domain query on dictionary-compressed data, the frame-of-referenced compressed data would require a full scan of all tuple values within the block.

To mitigate this effect, we adapted the heuristic to take the impact on domain queries into account. We still select compressions primarily based on the achieved compression ratio. However, compression methods which support domain queries get a 16KB head start. In other words, dictionary compression with 2 byte tokens is preferred over Frame of reference compression with 2 byte offsets as long as the dictionary is not bigger than 16KB. In theory this could regress performance, as we are using more instructions to unpack each tuple. However, even with the additional indirection through the dictionary, we are still limited by memory bandwidth and not instruction count.

4 Evaluation

To illustrate the impact of domain query optimization, we provide performance numbers for hand-crafted domain queries against artificial input data. We controlled the duplication ratio within this data set. In this set-up, we measured the end-to-end performance within Hyper including all phases of query processing (parsing, semantic analysis, query optimization, query compilation and query execution). Thereby, this evaluation provides us a notion of the overall impact of DQO on query time for individual SQL queries issued against Hyper. We did not include the time spent by Tableau to generate the SQL queries.

Furthermore, we evaluated DQO on our test set introduced in [Vo18]. This test set contains over 60 thousand real-world Tableau visualizations demonstrating the real-world impact of DQO. We first present statistics on the data set which show the potential impact of DQO. Next, we measured the impact of DQO on the query times for loading a complete visualization without taking user interaction into account.

In all experiments, we also monitored changes in file size. As described in Section 3.5, the heuristic for choosing the compression was adapted to prefer compression schemes which lead to a slightly larger on-disk size if it thereby enabled domain scans. However, in practice we did rarely see an increases in file size caused by the changed compression heuristic. The additional space consumption was masked by already existing padding and alignment in the database file.

4.1 Performance Improvements on Synthetic Data

In order to gauge the impact of DQO in isolation, we measured the execution times for the 3 queries shown in Listing 4.

```
SELECT column_x FROM user_data GROUP BY column_x;
SELECT column_x FROM user_data
  GROUP BY column_x ORDER BY column_x;
SELECT MIN(column_x) FROM user_data;
```

Listing 4: SQL queries used for measuring DQO performance benefits.

The first query (subsequently referenced as *distinct*) requests the domain of a column in its most straightforward way. It represents a minimal domain query. Removing any part from it would make it no longer eligible for DQO. This query allows the most targeted performance measurement focused only on DQO. The second query (referenced as *distinct sorted*) is similar but sorts the data before returning it to the client. Thereby, it requires some additional processing to the performance measurement and the gains from DQO are expected to be smaller in comparison to the overall execution time. Domain queries issued by Tableau are posed in this exact form. The third query (referenced as *min*) focuses on measuring the time spent in the table scan. In contrast to the other queries, it does not require de-duplicating the values and thereby does not require building a hash table. Instead, the minimum is already computed while iterating over the tuples. Since keeping the minimum updated is cheap, this query measures mostly the time spent in the table scan. For this reason, this query is expected to be the fastest one among the test queries.

The queries were executed against generated data sets with a fixed number of 5 duplicated values within an increasing number of rows. The columns contain strings with moderate string lengths of up to 30 characters. We vary the number of tuples between 2^{17} and 1024×2^{17}. Thereby, the smallest tested relation consists of 1 Data Block while the largest one consists of 1024 Data Blocks.

By varying the number of tuples while keeping the distinct count constant, this experiment measures the most important axis along which domain queries must scale for Tableau: Tableau users are usually interested in filtering on a small domain, e.g., the regions in which they sold their products. The set of different regions rarely increases and tend to be quite small. At the same time, the number of orders and hence the overall number of tuples in the table increases over time.

All queries were run 5 times and we report the average times of the last 4 runs. The first run was excluded since it exposed large variance due to disk caching effects Compilation times and execution times were measured separately. *Compilation time* refers to the time needed for parsing the SQL query, turning the abstract syntax tree into an operator tree, logically optimizing the operator tree, generating & optimizing the corresponding LLVM and compiling the LLVM code to machine code. It does not include the time needed to receive the SQL text over the network. *Execution time* includes the time needed for the actual query execution and for serializing the results into network buffers. However, it does not include the time, until the client actually received the result, i.e., the network buffers

were not flushed. All reported times were measured on a Ubuntu Desktop machine with 64GB RAM a 16-core Intel Xeon E5-2630 CPU clocked at 2.40 GHz with all 32 hyper threads enabled.

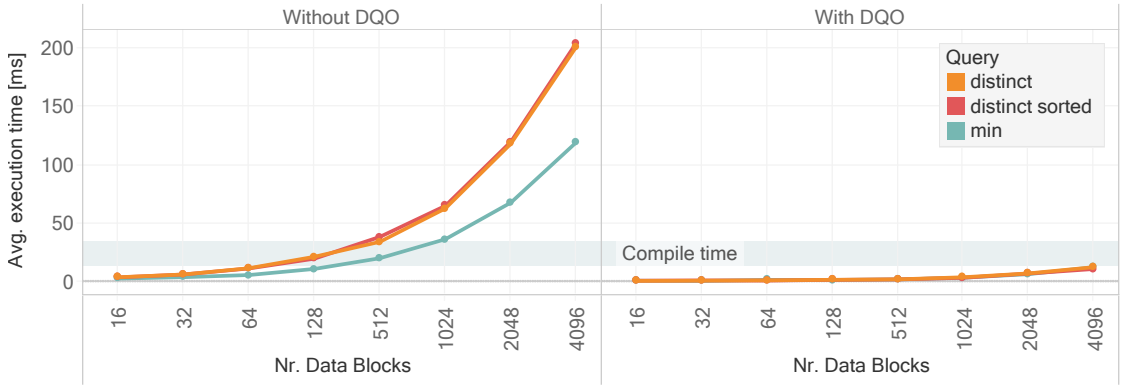

Fig. 7: Execution times for the domain queries with and without DQO on an increasing number of tuples with the number of distinct values being fixed to 5

Figure 7 shows the measured execution times. As expected, the *min* query represents a lower bound on the execution time for all other queries. Looking at the numbers without DQO, comparing execution times for the *distinct* and *distinct sorted*, they are mostly on-par. This means that the sort operation which is part of the *disctinct sorted* has a negligible overhead and that the main contributor to query time for domain queries as issued by Tableau is the actual duplicate elimination.

With DQO enabled, execution times for all scenarios are by orders of magnitude faster. The improvement is larger as the data sets get larger. With DQO, compilation time dominates query time in all measured experiments. Query compilation accounts for approximately 0.11 seconds independent of scanned input data. Compared to the execution time which are still less than 1 millisecond for the largest measured this input data, query compilation is an order of magnitude slower.

Note that even with DQO, the query time is still linear in the table size and not in its domain size. Although DQO significantly reduces the number of tuples produced by the table scan, the number of tuples still depends linearly on the number of tuples in the base table. This is because Data Blocks build per-block dictionaries. Duplicates are only getting de-duplicated within each Data Block, and each Data Block passes on the deduplicated values from its dictionary. Hence, the number of scanned tuples and thereby also the query execution time is still linear in the number of Data Blocks. Since the number of Data Blocks is linear in the number of overall tuples, the number of tuples produced by the table scan is still linear in the number of tuples.

4.2 Impact of DQO on Tableau Public

To judge the potential impact of DQO, we first did a static analysis of all 60 thousand visualizations from Tableau Public. Those 60 thousand visualizations are backed by approximately 120 thousand database files.

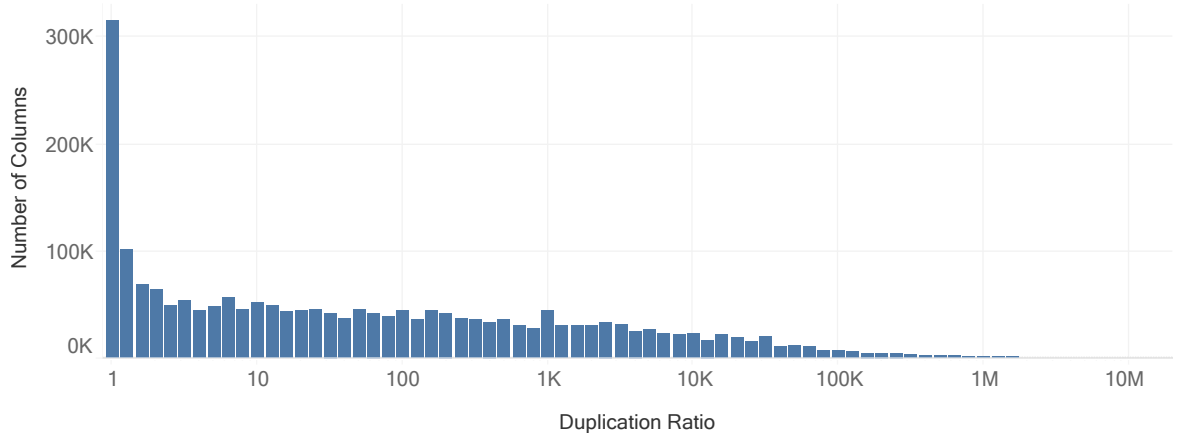

Fig. 8: Duplication ratio on Tableau Public visualized as the number of columns grouped by their duplication ratio

Figure 8 shows a histogram binning the two million columns contained in the database files by their duplication factor. The duplication factor is calculated as the number of tuples divided by the number of distinct values in the column. It thereby gives an upper bound on the number of tuples eliminated by DQO. It is depicted on the depicted on the abscissa using a logarithmic scale. The ordinate axis shows the number of columns with a given duplication factor. Highly duplicated columns are not shown in the figure since they would have distorted the overall scale of the chart. The long tail reaches a duplication factor of 116 million, i.e each unique value appears 116 million times.

The first bar of the histogram represents all columns with less than 5% duplicate values. $250k$ columns fall in this category. Among those columns $201.5k$ columns are completely duplicate free. For the duplicate-free columns, i.e., for 9.5% of all columns, DQO cannot provide any benefit. Also for the remaining $48.5k$ columns from this category the gain from DQO is only marginal, since the duplication factor is less than 5%. Hence, 12% of the columns are not profiting from DQO. The remaining 88% could potentially profit from DQO. In fact, for 71% of the columns at least 50% of the values are duplicates. Besides the presences of duplicates, DQO also requires one of the supported compression methods to be used. To our advantage, the compression heuristic tends to select eligible compression methods for duplicated data: The higher the duplication, the more likely a DQO-suitable encoding is chosen.

Most of the visualizations hosted on Tableau Public are based on only a small data set. Half of all columns contain less than thousand rows. If this data was saved using Data Blocks,

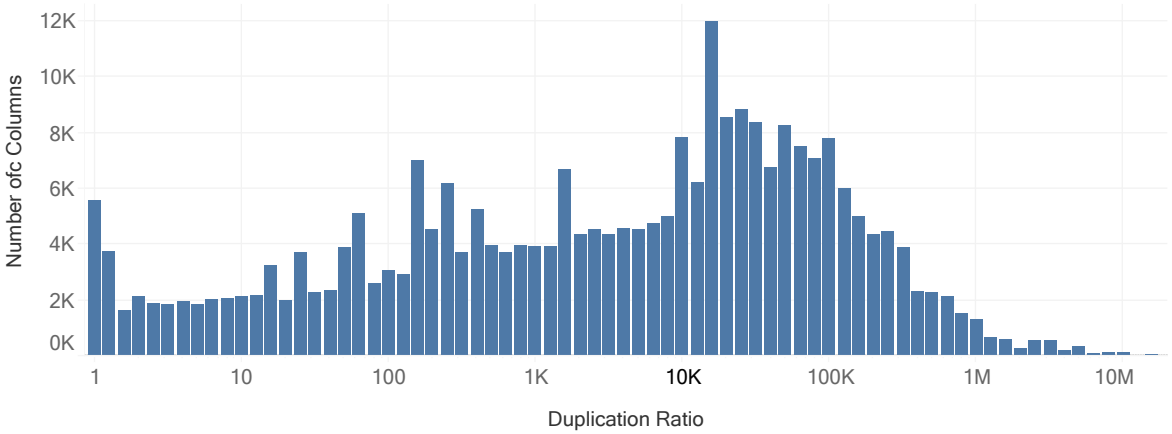

Fig. 9: Number of columns with more than one Data Block grouped by the duplication ratio.

for the vast majority of visualizations not even one Data Block would be filled. Queries on such small data sets are answered in a few milliseconds and most time is spent in the communication layer, for SQL parsing and for scheduling overheads. Focusing on columns with a significant number of rows, i.e., filling at least one Data Block, the numbers shift considerably. Figure 9 shows the same data as Figure 8 again, this time focusing on columns with a larger number of tuples.

We observed that larger datasets tend to contain more duplicate values. Our working theory to explain this correlation between data set size and duplication is the often denormalized data model: We suspect that many of the duplicates would not exist if the data was represented in a normalized schema and are only introduced during denormalization. In contrast, most small datasets were generated from Excel files or CSV files and did not join any data. Hence, for those small datasets Tableau did not denormalize the data and did not introduce the duplication which can be found in most larger extracts.

With this overview of the data in mind, the logical follow-up question is: How often are domain queries posed? In order to answer this question, we manually inspected the queries posed against 88 data sets uploaded to Tableau Public. The data sets were chosen based on their size, focusing on large data sets. The selected data sets have a zipped size between 300 megabytes and 1 gigabyte. In the process of rendering the visualizations based on those data sets, 2,940 queries were posed against Hyper. Most of those queries, in total 1,742, were meta data queries asking, e.g., for the contained tables and their columns. The remaining 1,198 queries resulted in 1,240 table scans on base tables. Approximately 12% of them (in absolute numbers: 157) would be theoretically eligible for DQO. Not all of those table scans were able to profit from DQO, however. This was because all of those table scans were operating on a column stored with a suitable encoding/compression. 128 of the table scans were able to profit from DQO for at least one of the scanned Data Blocks.

To measure the impact of DQO on an actual workload, all 2,940 queries from the examined visualizations were run against Hyper. We used the Ubuntu hardware described in Section 4.1. Without DQO Hyper needed 94.6 seconds on average to execute all queries including non-domain queries and meta data queries against a given dataset. DQO brings this time down by 1.5% to 93.2 seconds. These times include all time spent for query processing in Hyper, including both query compilation and query execution time. They do not include the time needed by Tableau to generate the queries and to interpret and visualize the results. To simulate a moderate load situation, we were always running two queries were run in parallel. The whole experiment was repeated 10 times and we report the arithmetic average over all 10 runs. Given the previous numbers on the duplication factor usually encountered in extracts and that about 6% of the queries are domain queries, the performance benefit of only 1.5% might be surprising at first. In most cases other queries dominate processing time.

To see how the measured 1.5% performance improvement generalizes beyond those 88 data sets, we measured the query times on all 60 thousand visualizations in our test set. Note, that we used a different hardware setup for this last experiment. We used a machine with a Xeon E5-2699 CPU clocked at 2.2GHz with 2 processors and 44 cores. The machine is equipped with 512GB RAM and is running under Windows Server 2012. We chose this differing test machine configuration to make the measurements finish in a feasible time. We do not think, the different hardware to influence the relative benefit of DQO significantly.

Among the already fast visualization (i.e., the query time was already smaller than 200ms), there was only a marginal gain as DQO improved the geometric mean of the query times by 0.15%. For those workbooks, the impact of DQO is small mostly because the computation of the visualization was already fast anyway and most time was spent in other stages of query processing. In general, we are targeting our performance features to address slowness experienced by the customer. As such, we usually focus on the visualizations taking more than than $0.2s$ to compute. For fast scenarios a customer usually does not notice improvements. Focusing on the slower visualizations, we measured an improvement of 1.69% in the geometric mean which aligns with the 1.5 percent improvement measured in the previous experiment.

When taking a closer look at the visualizations profiting from DQO, we saw that for 1,454 visualizations out of the 60,000 visualizations, the overall query time increased performance by more than 10%. In 2 cases we even saw performance improvements of over 100%. In the most extreme case we saw, performance was increased by slightly more than 250%, reducing query time from $386ms$ to $110ms$.

5 Related Work

Tableau is based on Stolte's dissertation work on interactive data visualization [St03; STH08]. In earlier work, we analyzed the database workload of typical Tableau applications and found that it differs very much from the well-known analytical TPC benchmarks [Vo18].

Before being supplanted by Hyper, data storage and query processing in Tableau was performed by the Tableau Data Engine (TDE) [Te15; WET11; WT14]. In TDE, domain queries are optimized as part of logical query optimization. TDE exposes dictionary lookups to the logical optimizer as joins between a table containing the dictionary tokens and the dictionary [WET11]. Domain query optimization is thereby subsumed by join culling: The optimizer notices that the table contents itself are not required to answer the query. Therefore, it removes the table scan from the query plan and only the dictionary scan remains.

Hyper was designed as a hybrid OLTP and OLAP main-memory database system [KN11]. It compiles queries and stored procedures to machine code via the LLVM compiler framework [Ne11], and uses morsel-driven parallelism for execution on multi-core CPUs [Le14]. Hyper uses Datablocks [La16] as its columnar storage format using lightweight compression. The tradeoffs of various compression schemes have been studied by Damme et al. [Da17]. Column sketches [HKI18] are a recently proposed form of lossy compression of columns that allows one to carry out the query processing directly on the compressed data. Moerkotte [Mo98] introduced Small Materialized Aggregates as lightweight synopses of data columns. The same idea was later also utilized in IBM Netezza as zone maps[6]. Binnig et al. [BHF09] developed SAP HANA's ordered dictionary, which allows processing range queries on compressed columns.

6 Conclusion

In this paper, we introduced the notion of domain queries in the context of BI platforms such as Tableau. Domain queries are mostly used to populate filters with a list of all distinct values available in a data set. We have also shown how these domain queries can be answered efficiently in the Hyper database system, that is the data engine in Tableau products for data extracts. For this purpose, we first showed how to detect domain queries on relational algebra trees. After this, we went on to demonstrate how to optimally execute domain queries against Hyper's Data Block storage format.

We evaluated our implementation of domain query optimization (DQO) on synthetic data sets. Those experiments have shown that DQO can easily reach query time speedups of multiple orders of magnitude. To show the real-world impact of DQO, we further analyzed a set of 60 thousand real-world Tableau visualization from Tableau Public [Vo18]. A static analysis of these workbooks was conducted to get a feeling for the potential impact of DQO.

[6] https://www.ibm.com/developerworks/community/blogs/Wce085e09749a_4650_a064_bb3f3b738fa3/entry/understanding_netezza_zone_maps?lang=en

We manually inspected the queries posed against the largest data sets in our test set: 12% percent of the inspected queries qualified as domain queries. Measuring the impact of DQO on the query times for the queries issued during rendering those visualizations, we saw a performance improvement of up to 250%.

References

[BHF09] Binnig, C.; Hildenbrand, S.; Färber, F.: Dictionary-based order-preserving string compression for main memory column stores. In: SIGMOD. Pp. 283–296, 2009.

[Da17] Damme, P.; Habich, D.; Hildebrandt, J.; Lehner, W.: Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In: EDBT. Pp. 72–83, 2017.

[FKN12] Funke, F.; Kemper, A.; Neumann, T.: Compacting Transactional Data in Hybrid OLTP & OLAP Databases. PVLDB 5/11, pp. 1424–1435, 2012.

[HKI18] Hentschel, B.; Kester, M. S.; Idreos, S.: Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. In: SIGMOD. Pp. 857–872, 2018.

[KN11] Kemper, A.; Neumann, T.: HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In: ICDE. Pp. 195–206, 2011.

[La16] Lang, H.; Mühlbauer, T.; Funke, F.; Boncz, P. A.; Neumann, T.; Kemper, A.: Data Blocks: hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In: SIGMOD. Pp. 311–326, 2016.

[Le14] Leis, V.; Boncz, P.; Kemper, A.; Neumann, T.: Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In: SIGMOD. Pp. 743–754, 2014.

[Mo98] Moerkotte, G.: Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In: VLDB. Pp. 476–487, 1998.

[Ne11] Neumann, T.: Efficiently compiling efficient query plans for modern hardware. Proceedings of the VLDB Endowment 4/9, pp. 539–550, 2011.

[Ra08] Raman, V.; Swart, G.; Qiao, L.; Reiss, F.; Dialani, V.; Kossmann, D.; Narang, I.; Sidle, R.: Constant-Time Query Processing. In: ICDE. Pp. 60–69, 2008.

[St03] Stolte, C.: Query, analysis, and visualization of multidimensional databases. PhD thesis, Stanford University/, 2003.

[STH08] Stolte, C.; Tang, D.; Hanrahan, P.: Polaris: a system for query, analysis, and visualization of multidimensional databases. Communications of the ACM 51/11, pp. 75–84, 2008.

[Te15] Terlecki, P.; Xu, F.; Shaw, M.; Kim, V.; Wesley, R.: On improving user response times in Tableau. In: SIGMOD. Pp. 1695–1706, 2015.

[Vo18] Vogelsgesang, A.; Haubenschild, M.; Finis, J.; Kemper, A.; Leis, V.; Muehlbauer, T.; Neumann, T.; Then, M.: Get Real: How Benchmarks Fail to Represent the Real World. In: Proceedings of the Workshop on Testing Database Systems. 2018.

[WET11] Wesley, R.; Eldridge, M.; Terlecki, P.: An analytic data engine for visualization in Tableau. In: SIGMOD. Pp. 1185–1194, 2011.

[WT14] Wesley, R.; Terlecki, P.: Leveraging Compression in the Tableau data engine. In: SIGMOD. Pp. 563–573, 2014.