Dynamic Data Transformation for Low Latency Querying in Big Data Systems

Leandro Ordonez-Ante, Thomas Vanhove, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert and Filip De Turck
Ghent University - imec, IDLab
Department of Information Technology
Technologiepark-Zwijnaarde 15, Gent, Belgium
Email: {firstname.lastname}@UGent.be

***Abstract*—Big data storage technologies inherently entail high latency characteristics, preventing users from performing efficient ad-hoc querying and interactive visualization on large and distributed datasets. Most of the existing approaches addressing this issue thrive on de-normalization of the static data schema and creation of application specific (i.e. hard-coded) materialized views, which certainly reduce data access latency but at the expense of flexibility. In this regard, this paper proposes an approach that relies on an iterative process of data transformation intended to generate *read-optimized* data schemas. The transformation process is able to automatically identify optimization opportunities (e.g. materialized views, missing indexes), by analyzing the original data schema and the record of queries issued by users and client applications against the data set. An experimental evaluation of the proposed approach evidences a significant reduction in the query latency, ranging from 81.60% to 99.99%.**

***Keywords*-Low latency querying; Dynamic data transformation; Polyglot persistence; Big Data; Denormalization**

I. Introduction

Extensive research efforts have been undertaken during the last few years to come up with optimal mechanisms for managing large (and predominantly unstructured) data sets. There exists a wide range of technologies, tools and frameworks in the big data ecosystem: from cost-efficient and reliable distributed data storage, to end-user applications dealing with specific domain requirements, as well as cloud-based services in place to deploy, scale and manage these technologies. One of the main open challenges big data technology faces nowadays is to lower the data access latency inherent to these large and distributed data collections.

For data-driven business in application domains such as social media, Internet of Things (*IoT*) and smart cities, being able to make sense of data as soon as it is produced is becoming of cardinal importance to ensure their optimal operation, even as a critical part of their value proposition. A common requirement of these data-intensive application domains is to be able to support (near)real-time data analysis and visualization, and efficient ad-hoc querying. The current technology landscape offers a diversity of solutions claiming to meet these requirements ranging from SQL-on-Hadoop systems [1], to in-memory computing frameworks like Apache Ignite [2], and software architectural patterns such as the Lambda [3] and Kappa [4] architectures. However, according to previous research and industry benchmarks [5], [6], [7], [8], these solutions either failed to achieve low latency for ad-hoc querying workloads, or effectively offer interactive performance, but at the expense of flexibility by relying on precomputed information views.

This paper introduces a systematic method for data transformation intended to generate a dynamic *read-optimized* schema enabling interactive-level latency for both data exploration and visualization tasks. The proposed approach extends existing methods of data transformation for polyglot persistence with no application downtime [9], and automatic schema denormalization based on query workload analysis [10].

The remainder of this paper is organized as follows: Section II describes the motivation behind this work. Section III addresses the related works. Section IV elaborates on the proposed systematic data transformation approach. Section V deals with the experimental setup and results. Finally conclusions and pointers towards future work are provided in Section VI.

II. Motivation

With more and more companies perceiving an increased value of data, high expectations are set on business intelligence (BI) and data analytics applications in general. Business users typically demand from these solutions the ability to perform ad-hoc queries and get visual insight on business data as it becomes available, which implies the execution of read-intensive operations. Being able to carry out these tasks in a timely manner, which is required for interactive applications, entails a big challenge for BI software, imposed by the sheer and growing amount of data it has to deal with. To cope with this, existing enterprise applications often separate BI operations—mostly supported by *Online Analytical Processing* systems (OLAP)—from day-to-day transaction processing—a.k.a. *Online Transaction Processing* (OLTP) [11].

OLTP systems rely mostly on write-optimized storage and highly normalized data models, while BI software works on top of read-optimized schemas featuring data redundancy and precomputed information views derived from the transactional system data. However, these read-optimized schemas are mainly fixed, restricting the ability

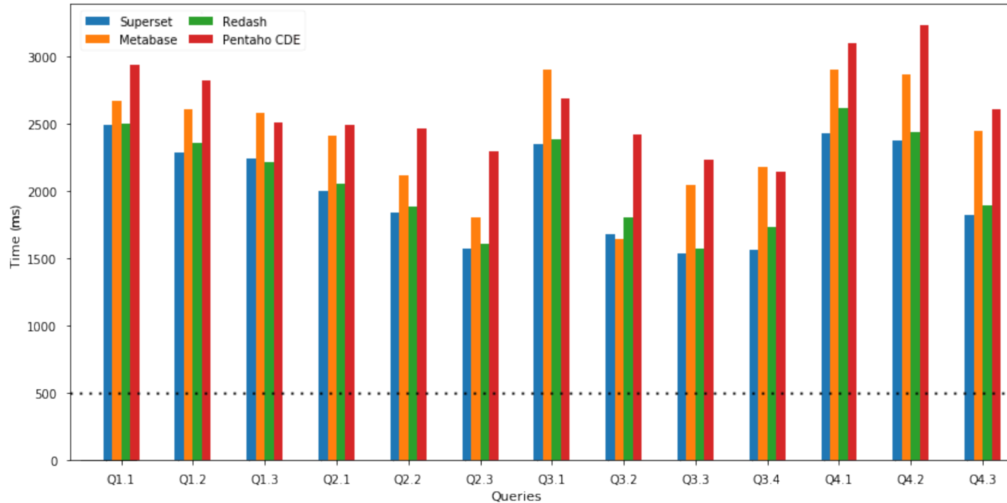

Figure 1. Open source BI tools: Average time per SSB query (SSB on PostgresSQL v9.6.2 with scaling factor SF = 1)

of BI systems to solve ad-hoc queries in consequence.

Furthermore, according to a comparative evaluation performed on four open source technologies for data exploration and dashboarding (*Superset* from Airbnb [12], *Metabase* [13], *Redash* [14], and *Pentaho Community Dashboard Editor (CDE)* [15]), despite the fact of using denormalized schemas, these tools fail to run queries under low latency constraints on moderate-size data collections. The mentioned evaluation was conducted on six million records of the *Star Schema Benchmark (SSB)* [16], stored on PostgreSQL [17]. Four instances of the SSB dataset were deployed, as well as one instance of the four visualization tools under evaluation on top of each one of them.

Figure 1 shows the average time (over 60 iterations) each dashboarding tool takes to run the queries being part of each one of the *query flights* defined in the Star Schema Benchmark specification ($Q1.1 \sim Q4.3$ in Figure 1). In all cases query runtime widely exceeds the interactive time limit criteria, set to 500 ms (in [18] it has been shown that interactions with higher latency cause users to unconsciously interact less with the tools), resulting in a poor user experience.

The previously mentioned platforms and some similar ones like Tableau [19] are tailored for ad-hoc querying and data visualization. However these tools only hold interactive query resolution when running on moderate-sized collections of aggregated data. Moreover, apart from using traditional caching, these tools often lack effective mechanisms for optimizing data retrieval tasks.

The work addressed in this paper shows that the query response time can be lowered by incrementally and systematically optimizing the data schema according to query construction patterns. This work primarily focuses on read-intensive applications demanding ad-hoc querying under low latency constraints, leaving write operations as part of further research on the approach presented herein.

III. Related Work

The problem of access latency in data exploration and visualization has been addressed from several perspectives. SQL-on-Hadoop platforms such as Apache Drill [20] and Apache Impala [21] claim to have interactive performance and to support ad-hoc queries by using a custom query execution engine that bypasses MapReduce and its inherent high latency. However, according to the AMP Lab Big Data benchmarks [5], even though the response time of these platforms outperforms those from similar frameworks like Apache Hive [22], they cannot guarantee interactive-level response times.

Other approaches tackle the latency problem derived from accessing the wide variety of data storage technologies used in big data applications. In [23], Giese et al. argue that end-users depend on the assistance of IT experts to pose ad-hoc queries to heterogeneous data stores. The above is regarded as a bottleneck limiting the ability of users to efficiently and promptly access big data sets. Giese et al. propose to approach this problem using *ontology-based data access* (OBDA): capturing end-user conceptualizations in an ontology and use declarative mappings to connect the ontology to the underlying data sources. Then, end-users can pose queries in terms of concepts of the ontology which are then rewritten to queries against the sources, avoiding the intervention of IT experts. This approach however entails a number of limitations in terms of (*i*) usability, as end-users need to formulate queries using a formal query language, (*ii*) costs of creating and maintaining both ontology and mappings, and (*iii*) efficiency of both the translation process and the execution of the resulting queries.

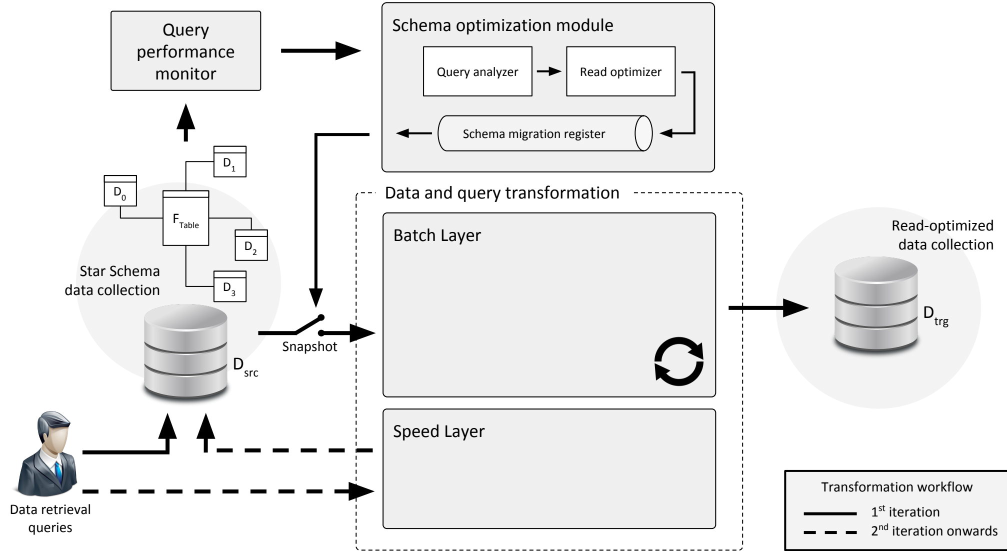

Figure 2. Dynamic data transformation for read-optimization: Architecture overview

In [9] the usage of multiple storage technologies in a single application is regarded as polyglot persistence. When optimally applied, polyglot persistence boosts application performance by storing each different data type into the data store technology best suited for it. This way for instance, highly accessed data may be loaded into a read-optimized data store. In this respect, Vanhove et al. propose a dynamic schema and data transformation approach for bringing the benefits of polyglot persistence to legacy applications, i.e. those storing their data in one relational data store. The mechanism conceived in [9] is able to transform schema and data between a source and a target data store without application downtime and avoiding code changes, as client applications are not required to change the query language they use, thanks to an ever running live query transformation. The transformation process is based on the Lambda Architecture [3]: the batch layer is in charge of transforming a snapshot of the data available in the source data store, while the speed layer performs transformations on the stream of new queries issued after the snapshot was taken.

The approach discussed in this paper builds on top of this work, leveraging on its dynamic transformation capability to incrementally generate a read-optimized version of a source data collection.

The mechanism proposed herein involves discovering relationships between data entities encoded both in the dimension/table structure and in query constructs composing `SELECT` statements. Then, a number of read-optimization methods (such as table partitioning, table collapsing, materialized views, and indexing) are iteratively applied on the dataset under consideration, based on the learned relationships. In this way, a system implementing this approach is able to incrementally deliver interactive-level response times for ad-hoc querying and visualization tasks.

IV. Dynamic read-optimization

As indicated previously, current business intelligence software mainly operates on top of denormalized dimensional data schemas. The dynamic transformation approach introduced in this paper is intended to incrementally read-optimize the schema specifying the structure of a dimensionally modeled dataset, in order to minimize the query runtime. Figure 2 presents an overview of the architecture of the proposed mechanism. The diagram depicts the four main components of this approach:

- A source data store complying the star (dimensional) schema (D_{src}).
- A query performance monitoring system in charge of gathering measurements like query runtime, frequency, and result set size. This component leverages on existing tools and libraries for collecting performance statistics in open-source database technologies, such as *pg_stat_statements* from PostgreSQL and the *MySQL Performance Schema*.
- A schema optimization module, in charge of analyzing the information about query performance, identifying query usage/construction patterns and relationships between data entities, and defining which optimization method to apply.
- A data transformation component, responsible for applying the actions designated by the schema optimization module. This component builds further on the data transformation mechanism conceived by Vanhove et al. in [9], by turning it into an iterative process.

A. Transformation Workflow

The transformation process conducted by the proposed mechanism can be described in terms of iterations.

First iteration:

1) All data retrieval queries are directed towards the source data collection (D_{src}), while information regarding their execution is collected by the query performance monitoring system. Then, this performance information is fed to the schema optimization module.
2) Inside the schema optimization module, the query performance information is processed (see in Section IV-B). Then, a specification of the optimization actions to be applied on the (D_{src}) schema is generated and fed into an append-only log (*schema migration register* in Figure 2).
3) Finally, each of the schema migration specifications loaded into the schema migration register triggers the capture of a snapshot of D_{src}, which is then handed to the batch layer of the data transformation component. In this layer a job is submitted to transform both the structure and data present in the snapshot. This job results in a read-optimized version of D_{src} (target data collection, D_{trg} in Figure 2).

Subsequent iterations: All iterations following the first one start with a handover process, where D_{trg} becomes D_{src} and further optimization actions are applied on it. Once this handover process is complete, the data retrieval queries are directed towards the new D_{src} through the speed layer of the data transformation component, where queries are translated according to the structure of the new read-optimized data set. Then, the subsequent process comprises the same three steps previously described for the first iteration.

Thanks to the continuous query transformation applied in the speed layer, client applications (business intelligence software) are able to query the new read-optimized data collection (D_{trg}) as if they were querying the original D_{src}, this way avoiding code changes and ensuring uninterrupted operation.

It is worth noting that in case of performance degradation, it is possible to replay all the schema migrations applied over the original D_{src}, dismissing first the one causing the performance breakdown.

B. Query Analysis

Considering that the conceived transformation mechanism aims at lowering the latency for data access operations, only data retrieval queries, i.e. `SELECT` statements, are considered in this analysis.

In the specification of the *Star Schema Benchmark (SSB)*, a set of queries is defined to evaluate the performance of databases products [16], which comprises typical data retrieval operations performed in business intelligence applications. All statements composing this query set conform to the following structure:

Listing 1. SSB query structure

```
SELECT select_list
FROM table_expression
[ WHERE search_conditions ]
[ [ GROUP BY column_list]
  [ ORDER BY column_list] ]
```

where `select_list`, `table_expression` and `search_conditions` are constructs that refer to entities and properties defined in the schema of the data collection, and also operations performed on them. The elements composing these constructs may encode data relationships which are not explicitly defined in the schema shaping the collection at hand. Approaches like [10] define automatic denormalization methods that leverage on such implicit relationships to improve the performance of `JOIN` operations. Aligned with this idea, the dynamic data transformation proposed in this work starts by analyzing the content of these query constructs, identifying a number of cases in which their values commonly fall into.

Select list: The elements composing the `select_list` construct frequently fall into two cases: *projection* and *aggregation*:

- *Projection (PROJ)*: comprises a set of columns or fields belonging to one or more tables (facts or dimensions). Consider the following query for instance:

```
SELECT lo_revenue, d_year, p_brand
FROM lineorder_fct, date_dim, part_dim
WHERE lo_orderdate = d_datekey
  AND lo_partkey = p_partkey
```

- *Aggregation (AGGR)*: when the select list contains a function call (e.g. `COUNT`, `SUM`, `AVG`) that performs a calculation on one or more columns, for example:

```
SELECT SUM(lo_revenue), lo_orderdate
FROM lineorder_fct
GROUP BY lo_orderdate;
```

Table expression: In general terms, this construct may deal with *one (SING)* or *multiple (MULT)* tables, depending on the elements included in the `select_list` and `search_conditions` constructs. It also may be the case that the table expression contains a `SELECT` statement (i.e. a *nested query*) instead of a list of tables. In such cases, nested queries are regarded as independent queries.

Search conditions: Search conditions are logical tests applied to each data record. These conditions consist of two value expressions and one relational operator that tests the relationship between the two values, for example:

```
value_1 = value_2
column_x > 2
```

Three cases are considered for this construct:

- *Filter on a range of values (RNGE)* (*val_exp1*: Column; *val_exp2*: ordinal/nominal/interval/ratio value), as in the following statement:

```
SELECT *
FROM lineorder_fct
WHERE lo_revenue > 15
```

- *Match between columns from different tables* (*val_exp1*: Column from Table X; *val_exp2*: Column from Table Y), used in `JOIN` queries, for example:

```
SELECT lo_revenue, d_year, p_brand, s_name
FROM lineorder_fct, date_dim, part_dim,
    supplier_dim
WHERE lo_orderdate = d_datekey
    AND lo_partkey = p_partkey
    AND lo_suppkey = s_suppkey
```

  Search conditions containing this kind of value expressions frequently reveal data relationships that may or may not be explicit on the schema specification. From the conditions in the previous query for instance, it may be possible to imply a relationship between `lineorder_fct` and each one of the dimension tables `date_dim`, `part_dim`, `supplier_dim`. Identifying these kind of relationships is key for the subsequent read-optimization process.

- *No filter criteria specified*: thoughtful queries always include at least one search condition, however it might be the case for a query to apply an aggregate function over all the records from a data entity, for instance:

```
SELECT COUNT(*)
FROM lineorder_fct
```

The query analysis mechanism keeps track of the occurrence of the above cases in queries issued against D_{src}, as well as execution statistics associated to each distinct query running on the data collection. The input of this analysis consist of tuples of the form:

$$q_i = (qst_i, et_i, rs_i, c_i)$$

where,

qst_i is the text of a `SELECT` statement,
et_i refers to the q_i execution time,
rs_i is the size of the result set retrieved by executing q_i,
and c_i is the number of times q_i has been executed.

Tuples for which the execution time is below the interactive time limit criteria ($et < 500ms$) are ruled out. For each of the remaining tuples the qst element is extracted and parsed to obtain the `select_list` ($\mathbf{sl_i}$), `table_expression` ($\mathbf{te_i}$), and `search_conditions` ($\mathbf{sc_i}$) constructs.

Finally, each one of these sets ($\mathbf{sl}$, $\mathbf{te}$, $\mathbf{sc}$) is processed and their elements are classified according to the cases previously discussed, turning each set into an associative array:

$$q'_i = (qst_i, (\mathbf{sl'_i}, \mathbf{te'_i}, \mathbf{sc'_i}), et_i, rs_i, c_i)$$

with,
$\mathbf{sl'_i} = \{AGGR : \mathbf{sl_{i_a}}, PROJ : \mathbf{sl_{i_p}}\}$
$\mathbf{te'_i} = \{SING : \mathbf{te_i}\} \quad \| \quad \{MULT : \mathbf{te_i}\}$
$\mathbf{sc'_i} = \{RNGE : \mathbf{sc_{i_r}}, JOIN : \mathbf{sc_{i_j}}\} \quad \| \quad \emptyset$

With this input, the procedure that comes next, defines an action or set of actions aimed at read-optimizing the schema of the data collection.

C. Read Optimization

The read optimization conceived in this approach consists in dynamically applying a number of methods for altering the structure of the data set, according to the characteristics of the queries being issued against it.

The methods used for read optimization can be classified into two categories: (1) *redundant structures*: *column/field indexing* and *query precomputing (materialized views)*, and (2) *non-redundant structures*: *denormalization (table collapsing* and *table partitioning)*. Methods from the first category involve storage costs and processing overhead when dealing with data updates, which is why the current approach has to compromise between low latency querying and storage/processing costs in order to decide on the right method to use.

Indexing: Indexes are one of the primary and most common means to improve data access performance in relational databases. Missing indexes are often the reason for queries to run slow, since this commonly imply the execution of costly full table scans. However, indexing is not always a suitable method for read-optimization, in the sense that it only improves the performance of queries retrieving a small percentage of rows from a table. In this way, the read optimization mechanism relies on the concept of *selectivity ratio* to decide whether or not to apply indexing on candidate columns.

The *selectivity ratio* is defined as follows:

$$\textit{selectivity ratio} = \frac{\textit{Number of unique values in column}}{\textit{Number of rows in table}}$$

As a general guideline, if the selectivity ratio for a column is less than 0.85, then processing a query with a condition on such column using a full-table scan is less expensive than using an index scan.

Thus, given the tuple representation of a query (q''), the read optimization mechanism first looks for columns lacking of indexes, listed in the `search_conditions` ($\mathbf{sc'}$)

under the *RNGE* category. Then, the *selectivity ratio* is computed for each of the resulting columns and the creation of an index is prescribed as a schema transformation specification, for those columns whose *selectivity ratio* is greater than 0.85.

Materialized views: When indexing is not a suitable method for speeding up data retrieval operations, the optimization mechanism resorts to query precomputing so that queries are no longer evaluated against the raw data, but against snapshots of the result sets corresponding to those queries. These snapshots are known as *materialized views* and may be regarded as read-only tables.

Particularly, this method is applied on time-consuming queries featuring aggregate operations (e.g. `COUNT`, `SUM`, `AVG`) and/or JOIN conditions. In this regard, when a query meeting these attributes enters the schema optimization mechanism, the creation of a materialized view is issued as a schema transformation specification, which is then appended to the schema migration register.

Denormalization: The last method for read optimization may involve a major transformation of the schema of the data collection. In this case, depending on the features of the query under inspection, two operations are applied as follows:

- *Table merging*: this is a common operation used to speed up query execution time by collapsing tables, this way dismissing joins between them. The optimization mechanism prescribes this operation when identifying time consuming queries featuring recurring *JOIN*-like search conditions. Consider for instance the following query:

  ```
  SELECT SUM(lo_revenue), d_year
  FROM   lineorder_fct, date_dim
  WHERE  lo_orderdate = d_datekey
  GROUP  BY d_year
  ```

  In this case, the query performs a join between the `lineorder_fct` and the `date_dim` through the `lo_orderdate` and `d_datekey` columns respectively.

- *Table partitioning*: splitting data tables row-wise (*horizontal partitioning*) or column-wise (*vertical partitioning*) is also a common operation used depending on the needs of the accessing application, to speed up query execution. The optimization mechanism prescribes table partitioning as a more storage efficient alternative to materialized views when indexing is not applicable. Horizontal partitioning is applied on the fact or collapsed tables (i.e. fact joined to dimension tables) based on columns composing recurrent search condition predicates. On the other hand, vertical partitioning is used to lower the costs of query execution by isolating frequently queried columns from those seldom accessed. In this way queries are issued against a usable/compacted version of the data collection, thus taking less time to run.

D. Schema and Data Transformation

The mechanism for data transformation proposed by Vanhove et al. in [9] is adapted and used in this work. Such mechanism relies on a centralized canonical data model for performing the translation from a source data store to a target data store. The canonical data model is represented as a directional graph encoding data entities, their attributes and relations between them. This graph-based canonical model enables ontology-like reasoning on the data, and allows for a technology independent data transformation process, endowing it with flexibility and extensibility to any data store technology. Figure 3 illustrates an example of the canonical model representing a data set comprising two entities.

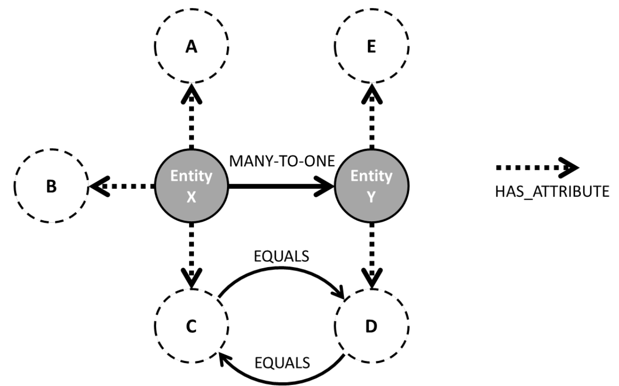

Figure 3. Canonical model for the structure of a data set. source: [9]

The read-optimization mechanism introduced in this work leverages on this canonical data model and the transformation approach proposed by Vanhove et al. [9], not to translate data between different data stores, but to modify a given data set schema according to the optimization operations detailed in the previous section. This introduces a new step to the original data transformation workflow proposed in [9]: the iterative alteration of the canonical model representing the structure of the source data set. Conceptually, these successive transformations can be seen as a layered sequence of canonical data models, with the one of the original data store (CM_{src}) lying on the first layer, and the subsequently derived canonical models ($CM_{trg<i>}$) being stacked upon one another as illustrated in Figure 4. Notice how relationships exist between entities from consecutive layers. Such relationships represent the optimization operations applied over previous data models, making the continuous translation of queries conducted in the speed layer possible, and

also allowing for traceability and reversibility in case of performance degradation.

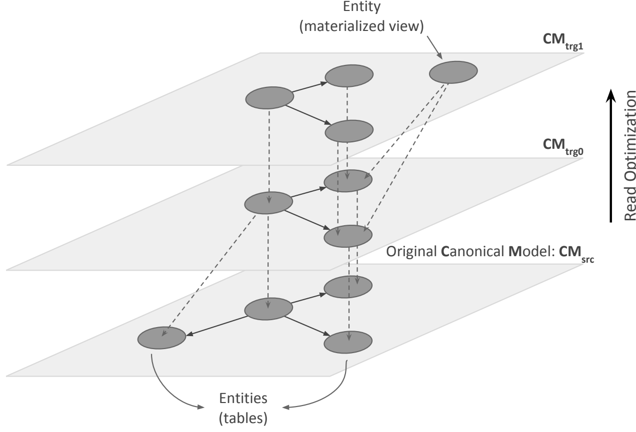

Figure 4. Layered sequence of canonical model transformations

Algorithm 1 Pseudo-code for the data transformation procedure

1: $Schema_{src} \leftarrow$ get D_{src} schema
2: $DataTuples \leftarrow$ read D_{src} snapshot
3: $CM_{src} \leftarrow$ transform $Schema_{src}$ into canonical model
4: detect relations in CM_{src}
5: $CurrentCM \leftarrow CM_{src}$ ▷ Current **C**anonical **M**odel
6: **for** each schema transformation specification (sts) **do**
7: $CM_{trgi} \leftarrow$ apply transformations from sts to $CurrentCM$
8: generate relations between entities from CM_{trgi} and $CurrentCM$
9: $Schema_{trg} \leftarrow$ generate new schema for D_{trg} from CM_{trgi}
10: load $Schema_{trg}$ into D_{trg}
11: **for** each tuple in $DataTuples$ **do**
12: transform to D_{trg}
13: **end for**
14: load data into D_{trg}
15: $CurrentCM \leftarrow CM_{trgi}$
16: $DataTuples \leftarrow$ read D_{trg} snapshot
17: **end for**

The pseudo-code in Algorithm 1 details the procedure conducted in the batch layer for transforming a given data set (D_{src}) into a read-optimized version of it (D_{trg}). Notice how the layered sequence of canonical model transformations is assembled through $CurrentDM$ (see lines 5, 8 and 15). The procedure inside the `for` loop at line 6 manages the generation of the read-optimized version of the canonical data model (CM_{trgi} at line 7), the target schema ($Schema_{trg}$ at line 9), as well as the translation of the data records and their further storage into D_{trg} (see lines 12 and 14).

V. Evaluation

This section reports on the estimation of the efficiency of the read-optimizing operations detailed in section IV-C, using the *Star Schema Benchmark (SSB)* as an objective testing dataset. The SSB was designed to measure the performance of data storage technologies against a data warehouse scheme, which comprises a dimensional data model (four dimensions, one fact table), an extensible dataset (size depending on a *scaling factor—SF*), and a set of queries typical from business intelligence applications. Those queries are arranged in four categories/families designated as *Query Flights*. A succinct description of the SSB query flights is presented next (a detailed definition of the SSB is available at [16]).

A. SSB Query Flights

***Q1**:* Performs an aggregate summation over the full fact table while filtering on columns of a single dimension (`date_dim`), and the `lo_discount` and `lo_quantity` columns of the fact table:

```
SELECT SUM(lo_extendedprice*lo_discount) AS
    revenue
FROM lineorder_fct, date_dim
WHERE lo_orderdate = d_datekey
  AND {d_year | d_yearmonthnum } = {YEAR |
      YEAR_MONTH_NUM}
 [AND d_weeknuminyear = {WEEK_NUM_IN_YEAR}]
  AND lo_discount BETWEEN {random(2..9) - 1} AND {
      random(2..9) + 1}
  AND lo_quantity BETWEEN {random(2..49) - 1} AND
      {random(2..49) + 1}
```

***Q2**:* This type of query features an aggregate summation over a multiple-table join, with restrictions on two dimension tables (`part_dim` and `supplier_dim`), grouping and sorting the results by columns of `date_dim` and `part_dim`:

```
SELECT SUM(lo_revenue), d_year, p_brand1
FROM lineorder_fct, date_dim, part_dim,
    supplier_dim
WHERE lo_orderdate = d_datekey
  AND lo_partkey = p_partkey
  AND lo_suppkey = s_suppkey
  AND {p_category | p_brand1} = {CATEGORY | BRAND}
  AND s_region = {REGION}
GROUP BY d_year, p_brand1
ORDER BY d_year, p_brand1
```

***Q3**:* As in the previous query flight, these queries summarize over a multiple join, this time filtering and grouping by columns from three dimension tables (`date_dim`, `customer_dim` and `supplier_dim`), and sorting the outcome by the resulting aggregate summation:

```
SELECT {c_nation, s_nation | c_city, s_city},
       d_year, SUM(lo_revenue) AS revenue
FROM lineorder_fct, customer_dim, supplier_dim,
    date_dim
WHERE lo_custkey = c_custkey
  AND lo_suppkey = s_suppkey
  AND lo_orderdate = d_datekey
```

```
  AND {condition on c_region, c_nation or c_city}
  AND {condition on s_region, s_nation or s_city}
  AND {d_year range | d_yearmonth condition}
GROUP BY {c_nation, s_nation | c_city, s_city},
    d_year
ORDER BY d_year ASC, revenue DESC
```

Q4: The last query flight comprises an aggregate summation over a multiple table join involving the full set of dimensions, grouped and sorted by columns from 2-3 dimensions, and filtering by fields belonging to 3-4 dimensions:

```
SELECT d_year, c_nation, SUM(lo_revenue-
    lo_supplycost) AS profit1
FROM lineorder_fct, date_dim, customer_dim,
    supplier_dim, part_dim
WHERE lo_custkey = c_custkey
  AND lo_suppkey = s_suppkey
  AND lo_partkey = p_partkey
  AND lo_orderdate = d_datekey
  AND c_region = {REGION}
  AND s_region = {REGION}
  AND {condition on p_mfgr or p_category}
 [AND {condition d_year}]
GROUP BY d_year, {c_nation | s_nation} [, {
    p_category | p_brand1}]
ORDER BY d_year, {c_nation | s_nation} [, {
    p_category | p_brand1}]
```

Thirteen `SELECT` statements conforming to these query structures compose the full query set of the SSB: Q1: 3 queries, Q2: 3 queries, Q3: 4 queries, Q4: 3 queries. For evaluation purposes 130 queries were derived from the original query set, composing an evaluation workload which ran against six different dimensions of the SSB dataset: 6 million rows (SF = 1), 12 million rows (SF = 2), 24 million rows (SF = 4), 48 million rows (SF = 8), 96 million rows (SF = 16), and 192 million rows (SF = 32). These datasets were stored into PostgreSQL[1] databases deployed on two `n1-highmem-2`[2] Google Compute Engine instances. In this way a query latency baseline was built to measure the impact on the data retrieval performance upon applying the read-optimization operations over the testing dataset.

B. Results

Once the read optimization operations were applied to the different settings of the SSB dataset, it was possible to link each SSB query flight to the operations that deliver the best latency performance for their particular features.

Q1: Queries from this flight are conditioned on columns of the fact table (`lo_discount` and `lo_quantity`). Two methods are deemed suitable for this scenario: *materialized views* and *table partitioning*. Figure 5 shows the effect on the query latency upon applying different horizontal partitioning schemes to the 48 million record SSB dataset. The best performance is achieved when partitioning on the combination of distinct values of the *search condition* columns. This outcome can be explained in terms of the higher selectivity ratio of the composition of `lo_discount` and `lo_quantity` ($\frac{550}{48\times10^6}$), in contrast to their selectivity ratio measured individually ($\frac{11}{48\times10^6}$ and $\frac{50}{48\times10^6}$ respectively). With this setting, latency values are comparable to the interactivity threshold ($500ms$), which means a relative reduction in the query running time ranging from 81.6% to 94.3%. As argued earlier in section IV-C *non-redundant structures* methods are favored over *redundant structures* methods, therefore in this particular case, horizontal partitioning is regarded as the most suitable of the read optimization methods at hand.

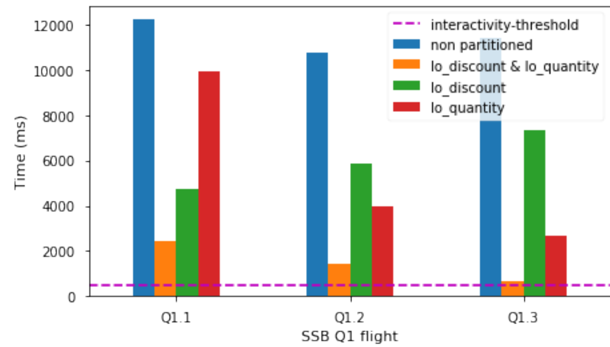

Figure 5. *Q1*: Horizontal partitioning latency, SSB with SF = 8

Q2: Range predicates (*RNGE* search conditions) in this family of queries comprise columns from dimension tables only. Thus, data retrieval will not benefit from applying denormalization methods like table partitioning in this particular case, which leads to the generation of materialized views. Figure 6 reports on the effect of using this read optimization method on the latency of the queries belonging to Q2, in contrast with the performance of the original dataset schema. One materialized view was generated for answering any query derived from the structure of this query flight (*query-flight view*). Results show how latency declines remarkably upon applying this method, from $400s$ (around $7min$) to $6ms$ per query on average, far below the interactivity threshold.

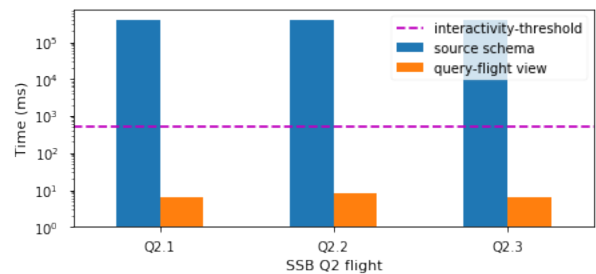

Figure 6. *Q2*: Materialized views latency, SSB with SF = 32

[1]PostgreSQL v9.5.8 working with the default configuration

[2]see https://cloud.google.com/compute/docs/machine-types#highmem

SSB storage per table (SF = 32)					Aggregated size	Materialized views size	% storage overhead
lineorder_fct	***date_dim***	***customer_dim***	***supplier_dim***	***part_dim***			
39GB	432KB	135MB	8600KB	160MB	39.3GB	2.07GB	5.27%

Table I
MATERIALIZED VIEWS: STORAGE OVERHEAD

Q3: Like Q2, this flight poses conditions on columns from dimension tables only, covering in this case a wider variety of queries. As a result and for experimentation purposes, two kinds of views were created: one *query-fight view* enabled for answering any query derived from this flight structure, and four views for dealing with queries based on the `SELECT` statements linked to Q3: Q3.1, Q3.2, Q3.3 and Q3.4. Figure 7 shows the variation on the latency resulting from using both generic and specific views, compared to the performance of querying the original dataset. By using materialized views, latency decreases by up to 99.86% for the *query-flight view*, and up to 99.99% for *query-specific views*. This means that one query that used to take $13min$ to run, now would take around $1.0s$ using a generic view, and from $2.0ms$ to $1.0s$ with specific views. It is worth noting that in some cases, latency is roughly the same for both generic and specific views (see Q3.2 and Q3.4 in Fig. 7). This is because in those cases, the size of the corresponding *query-specific view* match the size of the *query-flight view* (about 5 million records).

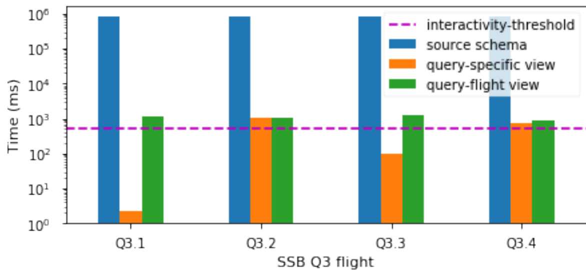

Figure 7. *Q3*: Materialized views latency, SSB with SF = 32

Q4: Due to the way a materialized view is computed for queries involving summarizing and grouping operations, its size is upper bounded by the number of different (distinct) values taken by the concatenation of the columns composing the view. In that sense, it is expected that the more columns a view includes, the larger it would become. Queries from Q4 feature the largest set of columns in the *search conditions* in comparison with the other query flights. In consequence, the *query-flight view* computed for Q4 comprises multiple columns from the four dimensions available in the dataset, reaching a size comparable to the size of the fact table. The above is the reason why query latency registers almost no improvement when using a generic view for this query flight (see Figure 8). As for the *query-specific views*, latency drops under the interactivity threshold in most of the cases, except for Q4.3, where again the amount of columns included on its corresponding `SELECT` statement, leads to the generation of an overly-large materialized view.

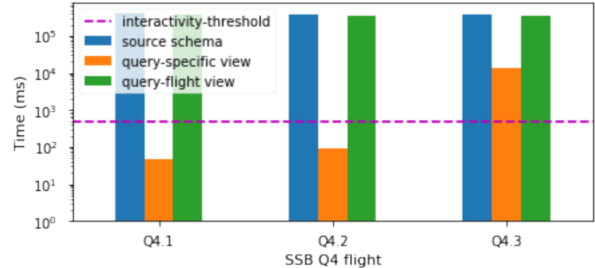

Figure 8. *Q4*: Materialized views latency, SSB with SF = 32

In cases where the use of materialized views leads to a significant reduction in query latency, the storage space overhead associated to these redundant structures doesn't exceed 6% of the total dataset size, which can be regarded as a reasonable cost considering the benefit they offer. Table I contrasts the size of the materialized views derived in the experimental evaluation reported in this section, in relation to each data structure of the largest SSB dataset used.

VI. CONCLUSIONS

The ability to efficiently conduct ad-hoc querying and get visual insights from ever-increasing amounts of data is nowadays key for organizations to support their business decisions. Being able to carry out these read-intensive operations under low latency constraints entails a considerable challenge for current data technologies including BI software. In this regard, this paper explores an approach for coping with data retrieval tasks from a dimensionally modeled dataset, under such low latency constraints. The proposed approach relies on a systematic method of data transformation intended to generate read-optimized data schemas. The formulated method takes as input the latency performance and construction patterns from queries being issued against the data collection, and adapts and optimizes

the source dataset accordingly, by iteratively altering a canonical model representing its structure.

The experimental evaluation focuses on measuring to what extent the query latency is affected as a result of applying the read-optimization operations defined in the data transformation process. Results show how operations involving *redundant-structures* (namely, *materialized views*) lead to a substantial improvement in terms of data retrieval latency (from 81.6% to 99.9% drop), enabling query execution times to go from several minutes to hundreds of milliseconds. However, considering the storage cost and maintenance overhead such *redundant-structures* entail, the proposed mechanism favors when possible alternative *non redundant-structure* methods, particularly table partitioning which proves effective for lowering query latency when `SELECT` statements are mainly conditioned on columns of the fact table.

Upcoming work on this research will incorporate polyglot persistence (i.e. intelligently scatter data over multiple data store technologies) to the optimization plans addressing also write-operations, and will develop a method for optimizing for multiple client services, with potentially competing behavior, which will involve the definition of a mechanism of classification on the user information needs.

Acknowledgment

This work was carried out in the scope of the ACThings High Impact Initiative.

References

[1] S. Pal, *SQL-on-Big-Data Challenges & Solutions*. Berkeley, CA: Apress, 2016, pp. 17–33. [Online]. Available: http://dx.doi.org/10.1007/978-1-4842-2247-8_2

[2] B. Anthony, K. Boudnik, C. Adams, B. Shao, C. Lee, and K. Sasaki, "In-memory computing in hadoop stack," *Professional Hadoop®*, pp. 161–182, 2016.

[3] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[4] J. Kreps, "Questioning the lambda architecture," *Online article, July*, 2014.

[5] AMPLab-UC-Berkeley. (2014) Amplab big data benchmark. [Online]. Available: https://amplab.cs.berkeley.edu/benchmark/

[6] J. Klahr. (2017) Tech talk: Bi performance benchmarks with google bigquery. [Online]. Available: http://blog.atscale.com/bi-benchmarks-with-google-bigquery

[7] P. Leszczyski and P. Zawistowski. (2015) Fast data hackathon. [Online]. Available: http://allegro.tech/2015/06/fast-data-hackathon.html

[8] L. Ordonez-Ante, T. Vanhove, G. Van Seghbroeck, T. Wauters, and F. De Turck, "Interactive querying and data visualization for abuse detection in social network sites," in *Internet Technology and Secured Transactions (ICITST), 2016 11th International Conference for*. IEEE, 2016, pp. 104–109.

[9] T. Vanhove, M. Sebrechts, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck, "Data transformation as a means towards dynamic data storage and polyglot persistence," *International Journal of Network Management*, pp. 1–23, 2017.

[10] S. Idicula, S. Petride, and N. Agarwal, "Automatic denormalization for analytic query processing in large-scale clusters," May 12 2015, uS Patent 9,031,932.

[11] H. Plattner, *A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases*. Springer Publishing Company, Incorporated, 2013.

[12] Airbnb-Inc. (2017) Github - airbnb/superset. [Online]. Available: https://github.com/airbnb/superset

[13] Metabase. (2017) Metabase - the simplest, fastest way to get business intelligence and analytics to everyone in your company. [Online]. Available: http://www.metabase.com/

[14] Redash. (2017) Redash - make your company data driven. connect to any data source, easily visualize and share your data. [Online]. Available: https://redash.io/

[15] M. R. Patil and F. Thia, *Pentaho for big data analytics*. Packt Publishing Ltd, 2013.

[16] P. E. ONeil, E. J. ONeil, and X. Chen, "The star schema benchmark (ssb)," *Pat*, vol. 200, no. 0, p. 50, 2007.

[17] PostgreSQL-Global-Development-Group. (2017) Postgresql. [Online]. Available: https://www.postgresql.org/

[18] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2122–2131, 2014.

[19] Tableau-Software. (2017) Tableau: Business intelligence and analytics. [Online]. Available: https://www.tableau.com/

[20] M. Hausenblas and J. Nadeau, "Apache drill: interactive ad-hoc analysis at scale," *Big Data*, vol. 1, no. 2, pp. 100–104, 2013.

[21] M. Kornacker and J. Erickson. (2012) Cloudera impala: Real time queries in apache hadoop, for real. [Online]. Available: http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real

[22] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009. [Online]. Available: https://doi.org/10.14778/1687553.1687609

[23] M. Giese, D. Calvanese, P. Haase, I. Horrocks, Y. Ioannidis, H. Kllapi, M. Koubarakis, M. Lenzerini, R. Möller, M. Rodriguez-Muro *et al.*, "Scalable end-user access to big data," *Big Data Computing*, pp. 205–245, 2013.