Automatic View Selection for Distributed Dimensional Data

Leandro Ordonez-Ante, Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert and Filip De Turck
Ghent University - imec, IDLab, Department of Information Technology,
Technologiepark-Zwijnaarde 15, Gent, Belgium

Keywords: Interactive Querying, View Selection, Clustering, Distributed Data, Dimensional Data, Data Warehouse.

Abstract: Small-to-medium businesses are increasingly relying on big data platforms to run their analytical workloads in a cost-effective manner, instead of using conventional and costly data warehouse systems. However, the distributed nature of big data technologies makes it time-consuming to process typical analytical queries, especially those involving aggregate and join operations, preventing business users from performing efficient data exploration. In this sense, a workload-driven approach for automatic view selection was devised, aimed at speeding up analytical queries issued against distributed dimensional data. This paper presents a detailed description of the proposed approach, along with an extensive evaluation to test its feasibility. Experimental results shows that the conceived mechanism is able to automatically derive a limited but comprehensive set of views able to reduce query processing time by up to 89%–98%.

1 INTRODUCTION

Existing enterprise applications often separate business intelligence and data warehousing operations—mostly supported by *Online Analytical Processing* systems (OLAP)—from day-to-day transaction processing—a.k.a. *Online Transaction Processing* (OLTP) (Plattner, 2013). While OLTP systems rely mostly on write-optimized stores and highly normalized data models, OLAP technologies work on top of read-optimized schemas known as *dimensional data models*, which leverage on denormalization and data redundancy to run computationally-intensive queries typical from decision-support applications (e.g. reporting, dashboards, benchmarking, and data visualization), that would result in prohibitively expensive execution on fully-normalized databases.

Traditional data warehousing systems are expensive and remain largely inaccessible for most of the existing small-to-medium sized business (SMEs) (Qushem et al., 2017). However, thanks to the advent of big data, more and more cost-effective (often open-source) tools and technologies are made available for these organizations, enabling them to run analytical workloads on clusters of commodity hardware instead of costly data warehouse infrastructure. Yet in such a distributed setting, some of the common challenges of conventional data warehousing systems become even more daunting to deal with: the way data is scattered and replicated across distributed file systems such as Hadoop's HDFS (Shvachko et al., 2010) makes it computationally expensive and time-consuming to run *Aggregate-Select-Project-Join* (ASPJ) queries which are one of the foundational constructs of OLAP operations.

For typical data warehousing and related applications using materialized views is a common methodology for speeding up ASPJ-query execution. The associated overhead of implementing this methodology involves computational resources for creating and maintaining the views, and additional storage capacity for persisting them. In this sense, finding a fair compromise between the benefits and costs of this method is regarded by the research community as the *view selection problem*.

In this regard, this paper presents an automatic view selection mechanism based on syntactic analysis of common analytical workloads, and proves its effectiveness running on top of distributed dimensionally-modeled datasets. The paper explores the techniques devised for abstracting feature vectors from query statements, clustering related queries based on an estimation of their pairwise similarity, and deriving a limited set of materialized views able to answer the queries grouped under each cluster.

The remainder of this paper is organized as follows: Section 2 addresses the related work. Section 3 describes the *view selection problem* and presents

Ordonez-Ante, L., Van Seghbroeck, G., Wauters, T., Volckaert, B. and De Turck, F.
Automatic View Selection for Distributed Dimensional Data.
DOI: 10.5220/0007555700170028
In *Proceedings of the 4th International Conference on Internet of Things, Big Data and Security (IoTBDS 2019)*, pages 17-28
ISBN: 978-989-758-369-8

an overview of the proposed approach for tackling it. Section 4 elaborates on the syntactic analysis conducted on analytical workloads, while Section 5 describes a proof-of-concept implementation of the devised mechanism, along with the experimental setup and performance results. Finally conclusions and future work are addressed in Section 6.

2 RELATED WORK

Extensive research has been conducted around the view selection problem, as evidenced in several systematic reviews on the topic such as those by (Thakur and Gosain, 2011), (Nalini et al., 2012), (Goswami et al., 2016), (Gosain and Sachdeva, 2017). The review elaborated in (Goswami et al., 2016) groups existing approaches in three main categories: (*i*) heuristic approaches, (*ii*) randomized algorithmic approaches, and (*iii*) data mining approaches.

Heuristic and randomized algorithmic approaches emerged as an attempt to provide approximate optimal solutions to the NP-Hard problem that view selection entails. Both types of approaches use multidimensional lattice representations (Serna-Encinas and Hoyo-Montano, 2007), AND-OR graphs (Sun and Wang, 2009; Zhang et al., 2009), or *multiple view processing plan* (MVPP) graphs (Phuboon-ob and Auepanwiriyakul, 2007; Derakhshan et al., 2008) for selecting views for materialization. Issues regarding the exponential growth of the lattice structure when the number of dimensions increases, and the expensive process of graph generation for large and complex query workloads, greatly impact the scalability of these approaches and their actual implementation in consequence (Aouiche et al., 2006; Goswami et al., 2016).

Unlike previously mentioned approaches, data-mining based solutions work with much simpler input data structures called *representative attribute matrices*, which are generated out of query workloads. These structures then configure a clustering context out of which candidate view definitions are derived. In (Aouiche et al., 2006; Aouiche and Darmont, 2009) candidate views are generated by merging views arranged in a lattice structure. Since the number of nodes in this lattice grows exponentially with the number of views, the procedure for traversing it can be expensive. Other data mining approaches for view selection, including the one from (Kumar et al., 2012), involve browsing across several intermediate and/or historical results, which is deemed to be a very costly and unscalable process (Goswami et al., 2016).

More recently, approaches such as (Goswami et al., 2017) and (Camacho Rodriguez, 2018) explore the application of materialized views on top of massive distributed data to speed up big data query processing. While the work of Goswami et al. (Goswami et al., 2017) addresses a solution based on a multi-objective optimization formulation of the view selection problem, it assumes as given the set of candidate views from which the selection is made. On the other hand, (Camacho Rodriguez, 2018) elaborates on the recently enabled support for materialized views in Apache Hive (Vohra, 2016b), however at the time of writing there is no indication of any built-in mechanism for supporting view selection with this new feature.

In view of the above, this paper elaborates on an automatic mechanism for materialized view selection on top of distributed dimensionally-modeled data. The mechanism presented in the following sections relies on syntactic analysis of query workloads using a representative attribute matrix as input data structure, assembled as a collection of feature vectors encoding all the clauses of each individual query in the workload at hand. With this input, a strategy for selecting a limited set of candidate materializable views is implemented, comprising the use of hierarchical clustering along with a custom query distance function complying with the structure of the feature vectors, and the estimation of a *materializable score* on the resulting clustering configuration, allowing to unambiguously identify materializable groups of queries.

3 MATERIALIZED VIEW SELECTION

Before addressing an overview of the proposed approach, let's first define the *view selection problem*.

Definition 1. View Selection Problem. *Based on the definition by Chirkova et al. (Chirkova et al., 2001): Let $\mathcal{R}$ be the set of base relations (comprising fact(s) and dimensions tables), $\mathcal{S}$ the available storage space, $\mathcal{Q}$ a workload on $\mathcal{R}$, $\mathcal{L}$ the function for estimating the cost of query processing. The view selection problem is to find the set of views $\mathcal{V}$ (view configuration) over $\mathcal{R}$ whose total size is at most $\mathcal{S}$ and that minimizes $\mathcal{L}(\mathcal{R}, \mathcal{V}, \mathcal{Q})$*

In the context of the view selection approach proposed herein, some assumptions are made for the system to identify and materialize candidate views out of the syntactic analysis of query workloads:

1. The source data collection (D_{src}) complies a star schema data model, i.e. it comprises a fact table

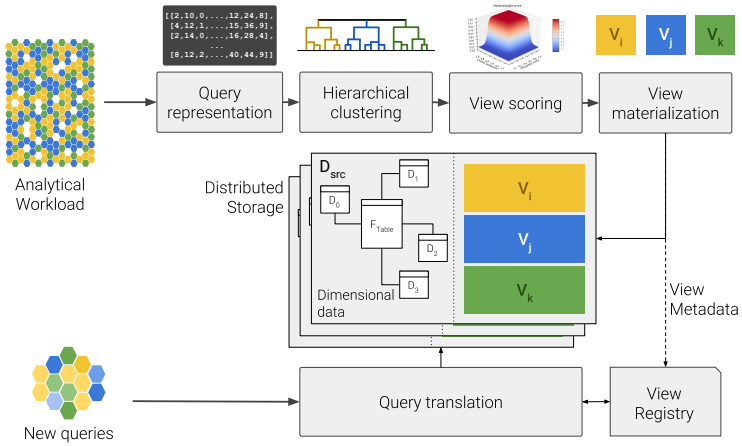

Figure 1: Materialized view selection: architecture overview.

referencing one or more dimension tables.

2. D_{src} is *temporary immutable*. This is a common scenario in some data warehousing systems, where analytical data is updated once in a substantial period of time —e.g. through an ETL procedure running on top of OLTP databases—, and queried multiple times during such a period.
3. Statistical information regarding D_{src}, such as the size (row count) of each of the base tables composing the dataset, as well as the cardinality of the attributes that make up these relations is available either by querying the metadata kept by the datastore, or by directly querying the base tables.
4. Latency is favored over view storage cost. This means that the decision on materializing candidate views is driven not by storage restrictions, but by the gain in query latency.

Figure 1 outlines the main components of the mechanism proposed herein to address the stated view selection problem. In terms of the definition 1, given a dimensionally modeled dataset $\mathcal{R}$ and a workload Q, the view selection mechanism starts by translating the queries in Q into feature vectors representing the attributes contained in each of the clauses of an ASPJ-query, i.e. aggregate operation, projection, join predicates and range predicates. In contrast to similar query representations such as the one used in (Aouiche et al., 2006), the method proposed herein accounts not only for query-attribute usage, but also for query structure by defining a number of regions/segments representative of each of the clauses of a Select-Project-Join (SPJ) query, i.e. aggregate operation, select list, join predicates and range predicates. This way, the devised query representation provides a more precise specification of the query statements in Q.

The collection of feature vectors of Q configure a clustering context $\mathcal{C}$. This context is then fed to a clustering algorithm able to identify groups of related queries based on a similarity score computed via a custom query distance function. Upon running the clustering job, the resulting clustering configuration $\mathcal{K}$ comprises several groups of queries the algorithm deemed to be similar. The idea behind building this clustering configuration is to be able to deduce view definitions covering the queries arranged under each cluster. The clustering algorithm might come up with spurious clusters, i.e. groups of queries that are actually not that related. To identify those spurious clusters and setting them apart from those clusters whose corresponding candidate views are worth materializing, a *materializable score* is defined, taking into account a measure of cluster consistency and the cluster size. Further details on this score and the clustering procedure are provided later in section 4.2.

Based on the results of the *materializable score* computed on the clustering configuration $\mathcal{K}$, a subset of the candidate views in $\mathcal{V}$, $\mathcal{V}_{mat}$, is prescribed to be materialized. Finally, with the views in place, the translation of new analytical queries matching said views is performed.

4 QUERY ANALYSIS

ASPJ-queries allow for summarization returning a single row result based on multiple rows grouped together under certain criteria (column projection and range predicates).

The syntactic analysis this work thrives on, starts by mining the information contained in the *select list* and *search conditions* clauses, encoding these values in a feature vector representation that enables further query processing.

4.1 Query Representation

The procedure for obtaining a text-mining-friendly representation of the queries takes each one of the `SELECT` statements from a workload Q and extracts the *aggregate* (ag_q) and *projection* (pj_q) elements, and *join* (jn_q) and *range* (rg_q) predicates, resulting in the following tuple:

$$q = (ag_q, pj_q, jn_q, rg_q) \tag{1}$$

The tuple above is the high-level vector representation of the queries from Q. Consider for example the following `SELECT` statement:

```
SELECT SUM(lo_revenue), d_year,
    p_category
FROM lineorder, dwdate, part
WHERE lo_orderdate = d_datekey
    AND lo_partkey = p_partkey
    AND d_year > 2010
GROUP BY d_year, p_category
```

For the query above:

$ag_q = [\texttt{SUM}, \texttt{lo\_revenue}]$

$pj_q = [\texttt{d\_year}, \texttt{p\_category}]$

$jn_q = [\texttt{d\_datekey}, \texttt{p\_partkey}]$

$rg_q = [\texttt{d\_year}]$

Each element of the above high-level vector representation gets mapped to a vector using a binary encoding function, as described below.

Definition 2. Binary Mapping Function. *Let* R *be a relation defined as a set of* m *attributes* $(a_1, a_2, ..., a_\mathsf{m})$ *—with* a_m *being the primary key of* R*—, and given* r *an arbitrary set of attributes, the binary mapping of* r *according to* R*, denoted by* $bm_\mathsf{R}(\mathsf{r})$*, is defined as follows:*

$$bm_\mathsf{R}(\mathsf{r}) = \{b_i\}, 1 \leq i \leq \mathsf{m}$$
$$b_i = \begin{cases} 1, & \text{if } a_i \in \mathsf{r} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

Using the mapping function above, the vector representation of each one of the query elements in Eq.1 (designated henceforth as *segments*), for a dimensional schema comprising one fact table and N dimension tables, is defined as follows:

$\mathbf{ag_q} = [aggOpCode, bm_\mathsf{Fact}(ag_q)]$

$\mathbf{pj_q} = [bm_\mathsf{Fact}(pj_q), bm_\mathsf{Dim1}(pj_q), ..., bm_\mathsf{DimN}(pj_q)]$

$\mathbf{jn_q} = [bm_\mathsf{Dim1}(jn_q), ..., bm_\mathsf{DimN}(jn_q)]$

$\mathbf{rg_q} = [bm_\mathsf{Fact}(rg_q), bm_\mathsf{Dim1}(rg_q), ..., bm_\mathsf{DimN}(rg_q)]$

where *aggOpCode* designates the aggregate operation using one-hot encoding, namely, `COUNT`: 00001, `SUM`: 00010, `AVG`: 00100, `MAX`: 01000, `MIN`: 10000.

A complete feature vector $\mathbf{q}$ representing a query $q \in Q$ is set by putting together the above-mentioned segments, that is:

$$\mathbf{q} = [\mathbf{ag_q}, \mathbf{pj_q}, \mathbf{jn_q}, \mathbf{rg_q}]$$

Accordingly, considering the `SELECT` statement in the example above and the dimensional schema described in (O'Neil et al., 2009) which comprises one fact table and four dimension tables, a complete feature vector instance (its decimal equivalent for length and clarity) is shown below:

$$\mathbf{q} = [[2,8],[0,0,16,8,0],[0,1,1,0],[0,0,16,0,0]]$$

The collection of feature vectors representing the queries from Q are arranged as a *representative attribute matrix*, configuring a clustering context $\mathcal{C}$.

4.2 Query Clustering and View Materialization

The view selection approach documented herein relies on *hierarchical clustering* (Friedman et al., 2009) for deriving groupings of similar queries. In contrast to other well-known clustering methods such as *K-Means* or *K-medoids*, hierarchical clustering analysis does not require the number of clusters upfront as parameter. Instead, it generates a hierarchical representation of the entire clustering context in which observations and groups of observations are stacked together from lower to higher levels, according to a distance measure based on the pairwise dissimilarities among the observations.

This way, a *dissimilarity metric* is required to apply hierarchical clustering analysis on a clustering context $\mathcal{C}$, along with a *linkage criterion* which estimates the dissimilarity among groups of queries as a function of the pairwise distance computed between queries belonging to those groups. In this sense, a distance function, *qDst*, is defined in which similarity between two queries is determined to be proportional to the number of attributes they share in a per-segment and per-relation (fact and dimensions) basis. Since vectors in $\mathcal{C}$ do not lie in an euclidean space, the *Weighted Pair Group Method with Arithmetic Mean* (WPGMA) clustering method is used as linkage criterion, instead of methods such as *centroid*, *median*, or *ward* (Müllner, 2011).

Under this set-up, the clustering procedure (detailed in algorithm 1) starts by assigning each query to its own cluster (see line 1). Then, the pairwise dissimilarity matrix between these singleton clusters, $\mathbf{D}$,

Algorithm 1: WPGMA clustering procedure.

1: $\mathcal{K} \leftarrow \mathcal{C}; \mathcal{C} = \{\mathbf{q_0}, \mathbf{q_1}, \ldots, \mathbf{q_N}\}$ ▷ Initializing clusters (singleton clusters)
2: $\mathbf{D} \leftarrow qDst(\mathbf{q}_i, \mathbf{q}_j)$ for all $\mathbf{q}_i, \mathbf{q}_j \in \mathcal{K}, i \neq j$ ▷ Pairwise dissimilarity matrix
3: $\mathbf{L} \leftarrow []$ ▷ Output matrix
4: **while** $|\mathcal{K}| > 1$ **do**
5: $(\mathbf{a}, \mathbf{b}) \leftarrow argmin(\mathbf{D})$ ▷ Get the nearest clusters
6: append $[\mathbf{a}, \mathbf{b}, \mathbf{D}[\mathbf{a}, \mathbf{b}]]$ to $\mathbf{L}$
7: remove $\mathbf{a}$ and $\mathbf{b}$ from $\mathcal{K}$
8: create new cluster $\mathbf{k} \leftarrow \mathbf{a} \cup \mathbf{b}$ ▷ Merge $\mathbf{a}$ and $\mathbf{b}$ into one cluster
9: update $\mathbf{D}$: $\mathbf{D}[\mathbf{k}, \mathbf{x}] = \mathbf{D}[\mathbf{x}, \mathbf{k}] = \frac{qDst(\mathbf{a},\mathbf{x}) + qDst(\mathbf{b},\mathbf{x})}{2}$ for all $\mathbf{x} \in \mathcal{K}$
10: $\mathcal{K} \leftarrow \mathcal{K} \cup \mathbf{k}$
11: **end while**
12: **return** $\mathcal{K}, \mathbf{L}$ ▷ $\mathbf{L}$: WPGMA dendrogram: $((N-1) \times 3)$-matrix

is computed and an empty matrix ($\mathbf{L}$) specifying the resulting dendrogram is initialized (lines 2-3). From $\mathbf{D}$, the two most similar (nearest) clusters are merged into one, and appended to $\mathbf{L}$ along with the distance between them (lines 5-6). Then, the pairwise dissimilarity matrix gets updated using the WPGMA method for computing the distance between the newly formed cluster and the rest of the currently existing clusters (eq. 3):

$$\mathbf{D}[(\mathbf{a} \cup \mathbf{b}), \mathbf{x}] = \frac{qDst(\mathbf{a}, \mathbf{x}) + qDst(\mathbf{b}, \mathbf{x})}{2}, \quad (\mathbf{a}, \mathbf{b} \quad and \quad \mathbf{x} \quad being \quad clusters) \tag{3}$$

This procedure is then repeated until there is only one cluster left. Finally, both the clustering configuration ($\mathcal{K}$) and the dendrogram matrix ($\mathbf{L}$) are returned (line 12).

As mentioned earlier, spurious clusters might be found in the derived clustering configuration. To avoid further processing of those query groups a score was defined indicating to what extent it is worth to materialize the view derived from a particular cluster.

Definition 3. Materializable Cluster. *A cluster* $\mathbf{c}$ *from a clustering configuration* $\mathcal{K}$ *is said to be* materializable *if the following conditions are met:*

1. *Queries in* $\mathbf{c}$ *are highly similar to each other.*
2. *Queries in* $\mathbf{c}$ *are clearly separated (highly dissimilar) from queries in other clusters.*
3. $|\mathbf{c}|$ *is large enough in proportion to the size of the workload* $|\mathcal{Q}|$.

A cluster meeting the first two conditions is said to be a *consistent cluster*, while the third condition prevents singleton and small clusters from being further processed. Based on the above definition, the *materializable score* of a cluster ($mat(\mathbf{c})$ in eq. 4) is computed as the product of two sigmoid functions: one on the per-cluster *silhouette score* (S) (Rousseeuw, 1987)—defined below in eq. 5—and the other on the per-cluster proportions (P).

$$mat(\mathbf{c}) = \left(\frac{1}{1 + e^{-k(S(\mathbf{c}) - s_0)}}\right)\left(\frac{1}{1 + e^{-k(P(\mathbf{c}) - p_0)}}\right) \tag{4}$$

With:

$$S(\mathbf{c}) = \frac{1}{|\mathbf{c}|} \sum_{\mathbf{q}_i \in \mathbf{c}} \frac{b(\mathbf{q}_i) - a(\mathbf{q}_i)}{max\{a(\mathbf{q}_i), b(\mathbf{q}_i)\}}, \quad P(\mathbf{c}) = \frac{|\mathbf{c}|}{|\mathcal{Q}|} \tag{5}$$

Where,

- k is a factor that controls the steepness of both of the sigmoid functions,
- s_0 and p_0 are the midpoints of the silhouette and cluster-proportion sigmoids respectively,
- $a(\mathbf{q}_i)$ is the average distance between $\mathbf{q}_i$ and all queries within the same cluster,
- $b(\mathbf{q}_i)$ lowest average distance of $\mathbf{q}_i$ to all queries in any other clusters.

Upon factoring out the spurious clusters, the next step is deriving view definitions covering the queries arranged under each of the *materializable clusters* ($\mathcal{K}_{mat} \subseteq \mathcal{K}$). Algorithm 2 below details the procedure conducted to derive the views V_i meeting this containment condition on each of the materializable clusters. In this procedure, the ASPJ clauses of the resulting views are defined in terms of the union of the corresponding attributes from each query in the cluster (*aggregate* ($ag_{\mathbf{q}}$), *projection* ($pj_{\mathbf{q}}$), *join* ($jn_{\mathbf{q}}$), and *range* ($rg_{\mathbf{q}}$) predicates in lines 4-7).

Algorithm 2: Procedure for deriving view definitions.

1: Let **c** be a cluster in $\mathcal{K}_{mat}$
2: $V \leftarrow [ag_V, pj_V, jn_V, groupBy_V]$ ▷ Output view definition
3: **for** each query **q** in **c** **do**
4: $ag_V \leftarrow ag_V \cup ag_{\mathbf{q}}$
5: $pj_V \leftarrow pj_V \cup pj_{\mathbf{q}} \cup rg_{\mathbf{q}}$
6: $jn_V \leftarrow jn_V \cup jn_{\mathbf{q}}$
7: $groupBy_V \leftarrow groupBy_V \cup pj_{\mathbf{q}} \cup rg_{\mathbf{q}}$
8: **end for**
9: **return** V

5 EVALUATION

5.1 Proof-of-concept Implementation

A bottom-up approach was adopted to test the view selection mechanism detailed in the previous sections. In this way, starting from a set of predefined view definitions, the effectiveness of the proposed mechanism is estimated in terms of its ability for identifying the same set of views and reconstructing their definitions, upon analyzing a query workload generated from query templates fitting the original set of views (see Figure 2).

This proof-of-concept implementation leverages the *Star Schema Benchmark* (SSB) as baseline schema and dataset, and therefore both the predefined views and query templates, as well as the query generator module were designed and built so they conform to the data model the SSB embodies.

Thirteen ASPJ-query statements compose the full query set of the SSB, arranged in four categories/families designated as *Query Flights* (a detailed definition of the SSB is available at (O'Neil et al., 2009)). For this proof-of-concept, three view definitions were derived based on the original SSB query set, and from each view definition, four query templates were prepared. Additionally, one template per each one of the 13 canonical SSB queries were also composed. With this set of 25 templates as input, a module that generates random instances of runnable queries enabled the creation of query workloads of arbitrary size. Listings below present the definitions of each one of the mentioned views.

Listing 1: Definition of View A.

```
SELECT sum(lo_revenue), p_brand1,
    c_region,
       s_region, d_year
FROM lineorder, customer, dwdate,
    part, supplier
WHERE lo_custkey = c_custkey
  AND lo_orderdate = d_datekey
  AND lo_partkey = p_partkey
  AND lo_suppkey = s_suppkey
GROUP BY p_brand1, c_region,
    s_region, d_year
ORDER BY p_brand1, c_region,
    s_region, d_year
```

Listing 2: Definition of View B.

```
SELECT sum(lo_ordtotalprice),
    p_category, c_city,
       s_city, d_yearmonthnum
FROM lineorder, customer, dwdate,
    part, supplier
WHERE lo_custkey = c_custkey
  AND lo_orderdate = d_datekey
  AND lo_partkey = p_partkey
  AND lo_suppkey = s_suppkey
GROUP BY p_category, c_city, s_city,
     d_yearmonthnum
ORDER BY p_category, c_city, s_city,
     d_yearmonthnum
```

Listing 3: Definition of View C.

```
SELECT sum(lo_supplycost - lo_tax),
    c_region, p_mfgr,
       s_region, c_nation, d_year
FROM lineorder, customer, dwdate,
    part, supplier
WHERE lo_custkey = c_custkey
  AND lo_orderdate = d_datekey
  AND lo_partkey = p_partkey
  AND lo_suppkey = s_suppkey
GROUP BY c_region, p_mfgr, s_region,
     c_nation, d_year
ORDER BY c_region, p_mfgr, s_region,
     c_nation, d_year
```

5.2 Definition of the Data Serialization Format

Since serialization formats determine the way data structures are turned into bytes and sent over the network, and how said structures are stored on disk, such formats have a major impact on the response time of data processing and retrieval operations performed in a distributed fashion. This is why, prior to evaluating the performance of the proposed view selection mechanism, the decision on which serialization format to use for encoding and storing the SSB datasets into HDFS needed to be made. Figure 3 outlines the setup arranged to conduct a benchmark analysis on three different data serialization formats, one being a text-based format (`CSV`) and two binary schema-driven formats (`Parquet` (Vohra, 2016c) and `Avro`

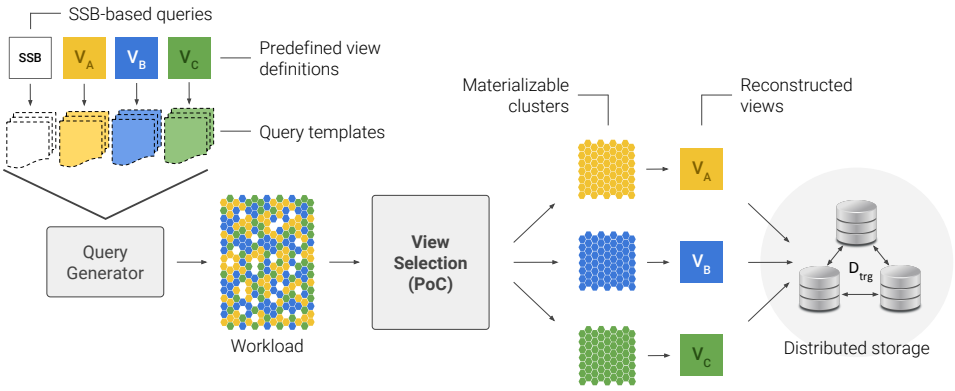

Figure 2: Proof-of-concept implementation of the proposed view selection mechanism.

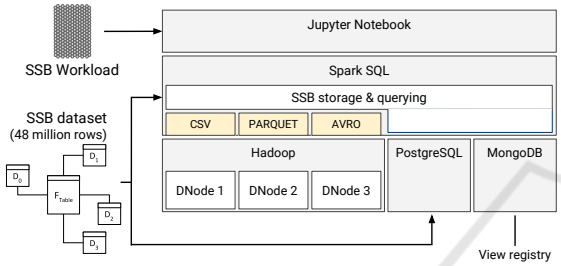

Figure 3: Set-up for deciding on the data serialization format.

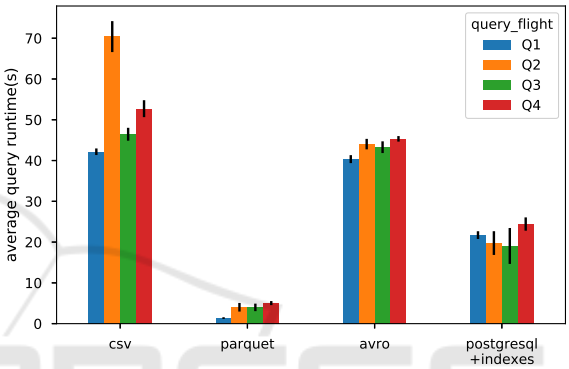

Figure 4: Average query-flight runtime per data serialization format (*SSB SF* = 8).

(Vohra, 2016a)). By leveraging on the built-in support Spark SQL provides for these serialization formats, a 48-million-row SSB dataset was encoded into CSV, Parquet and Avro and stored in HDFS. Then, the canonical SSB query set (consisting of 13 queries) was run against each of the encoded datasets, as well as against a separate dataset placed in a single-node PostgreSQL[1] database serving as a reference.

Figure 4 shows the results obtained from measuring the average query runtime (over 10 runs) for each one of the serialization formats. Notice how, in the mentioned conditions, queries running against the Parquet-encoded dataset ran up to 10 times faster than the reference relational database. Also, Parquet was the only serialization format that managed to outperform the average query runtime of PostgreSQL.

Serialization formats also have a significant impact on the size of the encoded data structures. While for text-based human-readable formats such as CSV data is stored *as-is*, binary formats like Parquet and Avro do apply compression on the data they encode. Figure 5 shows a comparison between the reported sizes (in gigabytes) of the SSB dataset for each of the considered serialization formats. According to these results Parquet is once again the most efficient serialization format, reaching a compression ratio of 3.6:1 in relation to the uncompressed CSV-encoded dataset, and 6.5:1 in relation to the PostgreSQL reference database (including primary key indexes). In consequence, Parquet was the chosen serialization format for encoding the various datasets involved in the evaluation of the proposed view selection approach.

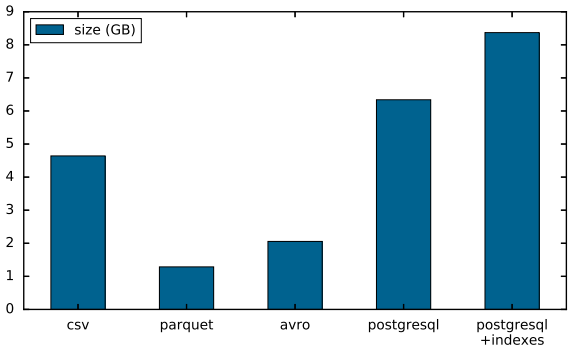

Figure 5: Disk space usage per data serialization format (*SSB SF* = 8).

[1]PostgreSQL 9.5.8 working with the default configuration and deployed on a VMWare® virtual machine as the ones used for the Hadoop cluster (postgresql.org)

5.3 Experimental Setup

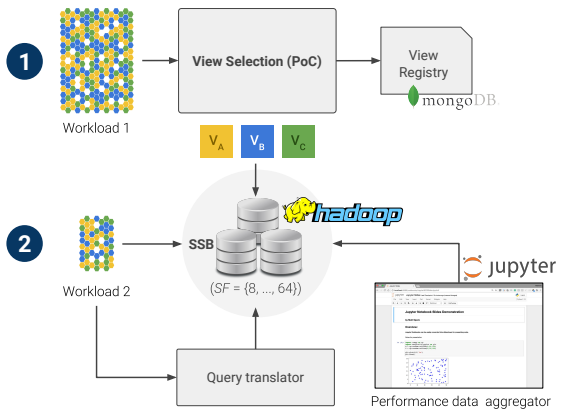

Figure 6: View selection experiment set-up.

Figure 6 depicts the arrangement of components and technologies used for conducting the experimental evaluation of the proposed view selection approach. This evaluation comprised two major stages:

(*1*) Running the view selection implementation on a 400-query workload to the point it materializes views A, B, and C (defined in section 5.1), while keeping track of the runtime involved in the procedures of *query clustering*, *view scoring* (using the materializable score defined in section 4.2), and *view creation (i.e. materialization)*.

(*2*) Once the views are materialized, run a 100-query workload against both the base SSB dataset and the materialized views. In doing the latter, workload queries first pass through a translation component that gathers the details of the available materialized views from the *view registry* (stored in a *MongoBD*[2] 2.6.10 document database), and adapts the incoming query statements accordingly.

For all the stages, the performance information collected from running the tests were aggregated and visualized using *Jupyter notebook*[3]. During this evaluation, workloads were run against eight different sizes of the SSB dataset, as specified below in table 1. These datasets were stored into a 3-Node *Hadoop*[4] 2.7.3 cluster deployed on 3 VMWare® virtual machines, each one with the following specifications: Intel® Xeon® E5645 @2.40GHz CPU, 16GB RAM, 250GB hard disk.

[2]Available at mongodb.com
[3]Available at jupyter.org
[4]Available at hadoop.apache.org

Table 1: SSB dataset sizes.

Scaling Factor (SF)	Dataset size (# rows)
8	48×10^6
16	96×10^6
24	144×10^6
32	192×10^6
40	240×10^6
48	288×10^6
56	336×10^6
64	384×10^6

5.4 Results

5.4.1 View Selection Overhead

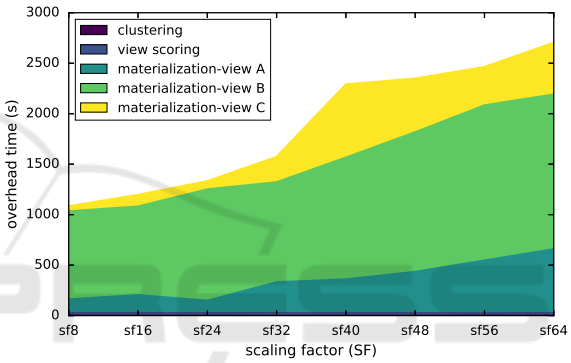

Figure 7: View selection runtime per process ($|Q| = 400$). Time needed for view materialization grows as the dataset size increases, while the overhead due to clustering and view scoring remains invariant.

The overhead of the view selection implementation was estimated first by running it on a 400-query workload throughout the considered range of sizes of the SSB dataset. Results show that, for a fixed-size workload, the runtime overhead grows nearly proportional to the size of the data collection (See Figure 7), with a major part of said overhead due to the view materialization itself. As mentioned before, the proposed view selection mechanism involves the execution of a sequence of steps: (1) *query clustering*, (2) *view (or cluster) scoring*, and (3) *view materialization*. Out of these only the first two steps have to do with the syntactical analysis of query sets described throughout this paper, while the last one refers to the actual materialization of the derived views in Parquet. Figure 7 shows the execution times for each one of the three mentioned steps, including the individual materialization of each one of the three selected views. Note how clustering and view scoring amount to only 20 seconds, and remain largely invariant as the dataset size grows larger. Nonetheless, it is worth mentioning

that the behaviour evidenced in Figure 7 for the materialization step cannot be assumed the same for any arbitrary set of views, since this part of the runtime overhead depends not only on the size of the dataset, but also on factors such as view size, the join predicates in the view definition, and —given that views are placed in a distributed file system— also the latency of the network.

On the other hand, the implementation of the WPGMA method used in the clustering analysis relies on the *nearest-neighbors chain algorithm* which is known to have $O(N^2)$ time complexity (Müllner, 2011). As Figure 8 shows, the overhead due to said analysis features a quadratic growth as the number of queries in the workload increases, outperforming alternative approaches with exponential complexity discussed back in section 2.

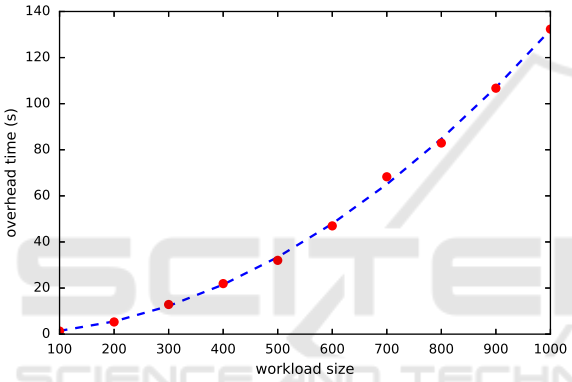

Figure 8: View selection overhead vs Workload size. The syntactical analysis features a quadratic growth w.r.t. the size of the query set.

When it comes to storage cost, view size varies depending on the cardinality of the fields used in the group-by clause of their definition (Aouiche et al., 2006), and whether or not there are hierarchical relations between such attributes (e.g. the one between `c_region`, `c_nation` and `c_city`). Figure 9 shows the size per materialized view, as well as the number of rows in the SSB base schema for a range of values of the scaling factor (*SF*). Notice that while the number of records of views A and C is fairly negligible in comparison with the base schema, view B and the base dataset have comparable sizes for small values of *SF*. Then, the size of view B tends to stabilize around 10^8 records as the base data set gets larger. Such difference in size among the views has to do with the cardinality of the fields used when defining said views. For instance, view B uses fields `c_city`, `s_city` and `d_yearmonthnum` whose cardinality is far larger than fields such as `c_region`, `c_nation` and `d_year`, used in the remaining views.

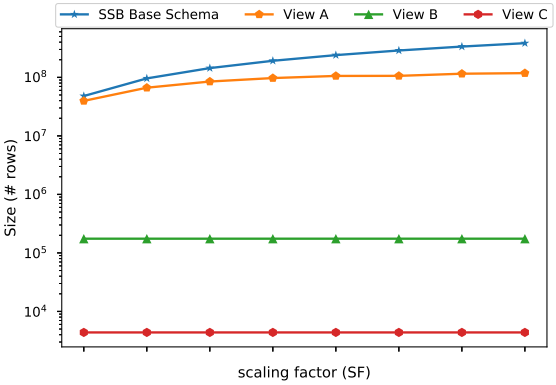

Figure 9: View size overhead vs SSB scaling factor.

5.4.2 View Selection Performance

With the selected views already materialized, a 100-query workload was run against each one of the eight base SSB datasets to get a query latency baseline. Out of those 100 queries, 75 were covered by the three available materialized views (25 queries per view), and the remaining ones were canonical SSB-based queries. Once the latency baseline was built, the same workload was issued this time with the query translation module in place, so that incoming queries matching any of the definitions of the available materialized views get rewritten and issued against them. Figure 10 illustrates the contrast between the baseline query runtime and the response time when queries run against materialized views. In the light of these results it is worth to highlight three key facts:

(*i*) For all the selected views the time required for queries to run against the base dataset steadily grows as the scaling factor (*SF*) increases from 8 (48 million rows) to 40 (240 million rows). This describes an expected behaviour since the base dataset is also growing at a uniform rate (48 million rows per step).

(*ii*) From $SF = 48$ (288 million rows) onwards, there is a strong though less regular increase in the average response time of queries issued against the base datasets: queries run 5-9 times slower compared to those running on the immediate smaller dataset ($SF = 40$), when only a 22-27% mean increase in the query runtime was expected. Such a stark difference is consequence of Apache Spark changing the mechanism it uses to implement join operations. Up to $SF = 40$, the size of the dimension tables is small enough for Spark to broadcast them across the executors —collocated with the Hadoop DataNodes— and use its most performant join strategy known as *Broadcast Hash Join*. However, for $SF \geq 48$ some of those dimen-

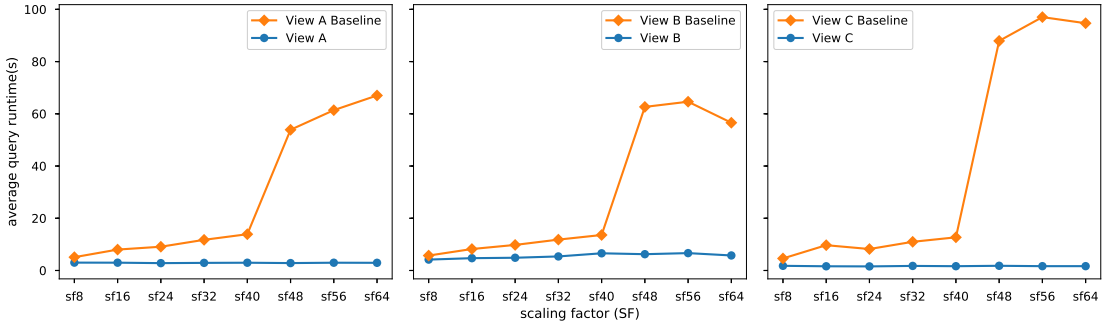

Figure 10: Query runtime per view: Baseline runtime vs. View runtime ($|Q| = 100$, Spark running on *yarn-client* mode with 3 executors and $spark.executor.memory = 1Gb$).

sions grows larger than the threshold set in Spark for them to be regarded as *broadcastable* datasets, compelling Spark to fall back to *Sort-Merge Join*, which entails an expensive sorting step on the tables involved in the join operation, ultimately impacting the query response time.

(*iii*) Materialized views outperform the base datasets for any value of $SF \geq 8$. Queries running on views A and C perform 2-8 times faster than the corresponding base dataset for values of SF between 8 and 40, and 22-65 times faster for larger values of SF. Likewise, queries running against the second view (view B) run up to 2 times faster than queries running on the base dataset for $SF \leq 40$, and up to 10 times faster for larger values of SF. The expensive join operations performed for queries running on the base dataset are bypassed for those matching any of the available selected views, allowing Spark to run those queries in a fraction of their original query response time.

Table 2 summarizes the results obtained from running the above test, stating the reduction in query runtime achieved through each one of the views relative to the average baseline query runtime.

Table 2: Query latency reduction per view ($|Q| = 100$).

	% Reduction in query response time		
SSB Dataset size (SF)	**View A**	**View B**	**View C**
8	40.86	26.99	61.21
16	62.66	42.64	83.36
24	69.05	50.14	80.95
32	75.16	54.48	84.06
40	78.60	51.56	87.05
48	96.02	88.96	98.00
56	96.10	89.15	98.24
64	95.36	89.82	98.46

6 DISCUSSION AND CONCLUSIONS

The analytical workloads typical in OLAP applications feature expensive data processing operations whose cost and complexity increases when running on a distributed setting. By identifying recurrent operations in the queries composing said workloads and saving their resulting output on disk or memory in the form of distributed views, it is possible to speed up the processing time of not only known but also previously unseen queries. That is precisely the premise behind the mechanism detailed in this paper, which leverages on syntactic analysis of OLAP workloads for identifying groups of related queries and deriving a limited but comprehensive set of views out of them. The views the devised mechanism comes up with proved an effective method for circumventing expensive distributed join operations and subsequently reducing the query processing time by up to 89%–98% with reference to the runtime on the base distributed dimensional data.

While the convenience of distributed materialized views is more prominently perceived as the dimensional data grows larger, one of the main open challenges of the proposed approach has to do with the unbounded size of the views that the mechanism is able to compose, which increases the associated processing overhead and cuts down the relative benefit of using these redundant data structures. To cope with this limitation, the view selection mechanism needs to be aware not only of the recurrent attributes and operations of queries but also of the cardinality of such attributes, so that views including attributes with high cardinality (consider for instance view B in section 5.1) get materialized as multiple size-bounded child views corresponding to partitions of the original view. Additionally, by keeping track of how the selection conditions of incoming workloads change

over time, it is possible to implement a continuous view maintenance strategy that performs horizontal partitioning on the derived views, allowing the proposed mechanism to adapt to workload-specific demands, using an approach similar to the one presented in (Ordonez-Ante et al., 2017). The implementation of the cardinality-awareness feature for the proposed view selection mechanism, as well as the view maintenance strategy discussed above are the future extensions of the work presented in this paper.

ACKNOWLEDGEMENTS

This work was supported by the Research Foundation Flanders (FWO) under Grant number G059615N - "Service oriented management of a virtualised future internet" and the strategic basic research (SBO) project DeCoMAdS under Grant number 140055.

REFERENCES

Aouiche, K. and Darmont, J. (2009). Data mining-based materialized view and index selection in data warehouses. *Journal of Intelligent Information Systems*, 33(1):65–93.

Aouiche, K., Jouve, P.-E., and Darmont, J. (2006). Clustering-based materialized view selection in data warehouses. In Manolopoulos, Y., Pokorný, J., and Sellis, T. K., editors, *Advances in Databases and Information Systems*, pages 81–95, Berlin, Heidelberg. Springer Berlin Heidelberg.

Camacho Rodriguez, J. (2018). Materialized views in apache hive 3.0. https://cwiki.apache.org/confluence/display/Hive/Materialized+views. Last accessed: 2018.10.15.

Chirkova, R., Halevy, A. Y., and Suciu, D. (2001). A formal perspective on the view selection problem. In *VLDB 2001*, volume 1, pages 59–68.

Derakhshan, R., Stantic, B., Korn, O., and Dehne, F. (2008). Parallel simulated annealing for materialized view selection in data warehousing environments. In *ICA3PP 2008*, pages 121–132. Springer.

Friedman, J., Hastie, T., and Tibshirani, R. (2009). Clustering analysis. In *The elements of statistical learning: Data mining, inference and prediction*, chapter 14, pages 501–520. Springer series in statistics, New York.

Gosain, A. and Sachdeva, K. (2017). A systematic review on materialized view selection. In Satapathy, S. C., Bhateja, V., Udgata, S. K., and Pattnaik, P. K., editors, *FICTA 2017*, pages 663–671, Singapore. Springer Singapore.

Goswami, R., Bhattacharyya, D., and Dutta, M. (2017). Materialized view selection using evolutionary algorithm for speeding up big data query processing. *Journal of Intelligent Information Systems*, 49(3):407–433.

Goswami, R., Bhattacharyya, D. K., Dutta, M., and Kalita, J. K. (2016). Approaches and issues in view selection for materialising in data warehouse. *International Journal of Business Information Systems*, 21(1):17–47.

Kumar, T. V. V., Singh, A., and Dubey, G. (2012). Mining queries for constructing materialized views in a data warehouse. In *Advances in Computer Science, Engineering & Applications*, pages 149–159, Berlin, Heidelberg. Springer Berlin Heidelberg.

Müllner, D. (2011). Modern hierarchical, agglomerative clustering algorithms. *Computing Research Repository (CoRR)*, abs/1109.2378.

Nalini, T., Kumaravel, A., and Rangarajan, K. (2012). A comparative study analysis of materialized view for selection cost. *World Applied Sciences Journal (WASJ)*, 20(4):496–501.

O'Neil, P. E., O'Neil, E. J., and Chen, X. (2009). The star schema benchmark (revision 3, june 5, 2009). https://www.cs.umb.edu/ poneil/StarSchemaB.PDF.

Ordonez-Ante, L., Vanhove, T., Van Seghbroeck, G., Wauters, T., Volckaert, B., and De Turck, F. (2017). Dynamic data transformation for low latency querying in big data systems. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2480–2489.

Phuboon-ob, J. and Auepanwiriyakul, R. (2007). Two-phase optimization for selecting materialized views in a data warehouse. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 1(1):119–123.

Plattner, H. (2013). *A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases*. Springer Publishing Company, Inc.

Qushem, U. B., Zeki, A. M., Abubakar, A., and Akleylek, S. (2017). The trend of business intelligence adoption and maturity. In *UBMK 2017*, pages 532–537.

Rousseeuw, P. J. (1987). Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20(1):53–65.

Serna-Encinas, M. T. and Hoyo-Montano, J. A. (2007). Algorithm for selection of materialized views: based on a costs model. In *Current Trends in Computer Science, 2007. ENC 2007. Eighth Mexican International Conference on*, pages 18–24. IEEE.

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *MSST 2010*, pages 1–10, Washington, DC, USA. IEEE Computer Society.

Sun, X. and Wang, Z. (2009). An efficient materialized views selection algorithm based on pso. In *ISA 2009*, pages 1–4. IEEE.

Thakur, G. and Gosain, A. (2011). A comprehensive analysis of materialized views in a data warehouse environment. *International Journal of Advanced Computer Science and Applications*, 2(5):76–82.

Vohra, D. (2016a). Apache avro. In *Practical Hadoop Ecosystem*, pages 303–323. Springer.

Vohra, D. (2016b). Apache hive. In *Practical Hadoop Ecosystem*, pages 209–231. Springer.

Vohra, D. (2016c). Apache parquet. In *Practical Hadoop Ecosystem*, pages 325–335. Springer.

Zhang, Q., Sun, X., and Wang, Z. (2009). An efficient ma-based materialized views selection algorithm. In *CASE 2009*, pages 315–318. IEEE.