Vertica Flattened Tables and Live Aggregate Projections: A Column-based Alternative to Materialized Views for Analytics

1st Yuanzhe Bei
Micro Focus Vertica
Cambridge, MA, USA
yuanzhe.bei@microfocus.com

2nd Thao Pham
Micro Focus Vertica
Cambridge, MA, USA
thaop@microfocus.com

3rd Akshay Aggarwal
Micro Focus Vertica
Cambridge, MA, USA
akshay.aggarwal@microfocus.com

4th Nga Tran
Micro Focus Vertica
Cambridge, MA, USA
nga.tran@microfocus.com

5th Jaimin Dave
Micro Focus Vertica
Cambridge, MA, USA
jaimin.m.dave@microfocus.com

6th Chuck Bear
Micro Focus Vertica
Cambridge, MA, USA
charles.bear@microfocus.com

7th Michael Leuchtenburg
Micro Focus Vertica
Cambridge, MA, USA
michael.leuchtenburg@microfocus.com

***Abstract*—Vertica is a column-oriented relational database management system built on massively parallel processing architecture. Rather than a traditional, monolithic implementation of materialized view, Vertica instead provides two separate features, flattened tables and live aggregation projections. These features not only support the basic functionality of materialized views, but also provide flexibility and consistency beyond the traditional materialized view implementation. Flattened tables contain denormalized columns whose main purpose is to materialize precomputed joins with other tables. Live aggregate projections are an always up-to-date layer built on top of individual tables, including flattened tables, which maintain real-time summaries of table contents. This paper will present the main architecture of these two features, discuss how they differ from traditional materialized views, and how they take advantage of Vertica's architecture to achieve high performance. Experimental results with TPC-DS benchmarks will be provided to demonstrate the claimed performance benefit.**

I. Introduction

Vertica's architecture is optimized for analytic queries, enabling it to process those queries efficiently. However, no matter how fast a relational database management system (RDBMS) can be, pre-computing (i.e. caching) helps in getting optimal performance by avoiding re-computation of expensive operations. Materialized views (MVs) are a state-of-the-art solution that have been implemented by most RDBMS, such as IBM DB2 [1], MS SQLSever [3], Oracle DB [4], and VoltDB [8]. MVs cache results of operations such as joins, aggregations, sorting, etc, which otherwise are expensive to recompute.

A well-known challenge for materialized views is when to refresh the data, and how efficient the refresh operation is. MVs are either lazily (i.e., on-demand) or eagerly (i.e., live) refreshed after data of their base tables are changed [10], [17], [26]. Obviously, live refresh in all cases would be impossible, because it would cause any data manipulation language (DML) statements on the base table to be prohibitively slow. Therefore, most RDBMS choose to support live refresh when the operation can be made fast enough and otherwise only perform on-demand refresh. Usually, it's the property of the query defining the MV and the availability of the MV's auxiliary data structures which determines if live refresh is feasible on the MV [5]. However, this can be both hard for the users to understand and remember and for the developers to implement and maintain.

Vertica does not implement the exact concept of MVs, yet supports the functionality of materialized views and beyond through (a) flattened tables (FTs), (b) projections, and (c) live aggregate projections (LAPs), taking advantage of our column-store architecture to provide efficiency and flexibility. FT, projections, and LAP allow users to control materialization from sorting to joins and aggregations. Since we have already discussed normal projections in our earlier paper about Vertica system [15], this paper will focus on FT and LAP.

To better demonstrate the concepts and usages of FT and LAP, we will use the following example, excerpted from TPC-DS benchmark.

Example Schema and Query

Figure 1 shows a part of TPC-DS schema which we use in our example. $item$ and $date\_dim$ are two dimension tables correspondingly representing the items being sold and date information. $store\_sales$ is a fact table, containing information for every sale in the store. Each sale record has an item key (ss_item_sk) and the date of the sale (ss_sold_date), which are foreign keys referencing $item$ table on i_item_sk and $date\_dim$ table on d_date_sk, respectively. We consider the following query, which is simplified from TPC-DS query

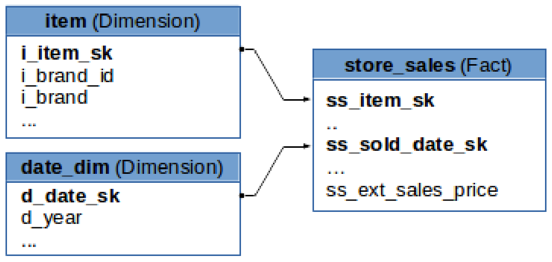


Fig. 1: Example schema, excerpted from TPC-DS

3. This query reports the total extended sales price per item brand each year [9].

```
SELECT dt.d_year,
       item.i_brand_id AS brand_id,
       item.i_brand AS brand,
       sum(ss_ext_sales_price) AS sum_agg
FROM date_dim dt, store_sales ss, item
WHERE dt.d_date_sk = ss.ss_sold_date_sk
      AND ss.ss_item_sk = item.i_item_sk
GROUP BY dt.d_year, item.i_brand, item.i_brand_id
ORDER BY dt.d_year, sum_agg DESC, brand_id
LIMIT 100;
```

The most expensive operations involved in answering the above query are (1) joining *store_sales* with *item* and *store_sales* with *date_dim*, and (2) grouping each brand on each year and compute the sum of sale prices on each group. If the above query is run frequent enough, we will benefit by materializing these operations.

A. Flattened Tables (FT)

Flattened Table denormalizes fact tables by incorporating values from dimension columns directly into them. Each denormalized column is defined separately by a query, and can be refreshed individually. Following is an example of a flattened table named *store_sales_ft*, which is essentially *store_sales* plus three denormalized columns used to serve the example query. The three added columns, with a SET USING clause specifying from where their data is derived, effectively materialize the joins between *store_sales* and *item* and *date_dim*.

```
CREATE TABLE store_sales_ft
(
    ss_sold_date_sk integer,
    ss_item_sk integer not null,
    ..
    ss_ext_sales_price decimal(7,2),
    i_brand char(50) SET USING
        (SELECT i_brand FROM item
         WHERE ss_item_sk = i_item_sk),
    i_brand_id int SET USING
        (SELECT i_brand_id FROM item
         WHERE ss_item_sk = i_item_sk),
    d_year int SET USING
        (SELECT d_year FROM date_dim
         WHERE d_date_sk = ss_sold_date_sk)
);
```

FT gives a lot of flexibility beyond a normal MV:

- **Single-partition, single-column refresh**: The denormalized columns in FT can be refreshed individually for a single partition, so there is no need for expensive, whole table refresh. This is made possible thanks to Vertica being a native column-store database.
- **Schema evolution for flexibility and less redundancy**: An FT can be an extension of a fact table; users only need to add denormalized columns to the fact table instead of having to create a separate MV on top of it. Similarly, if another query later requires materialization of another column, we can easily add the column to the FT instead of adding another MV. In both scenarios, FT greatly reduces redundancy and maintenance overhead. In the scenario where multiple denormalized columns would be included in a single MV, if a denormalized column is no longer needed, the sole column can be dropped individually without constructing a new MV.
- **Easy to define**: Defining each denormalization separately is intuitive and manageable. In addition, Vertica does not count the additional storage used for denormalized columns towards storage-based license, making it a desirable solution for our customers.

More details about FT will be described in Section III. Note that, although the major purpose of FT is to materialize pre-joined results (i.e. denormalized columns), it supports other kinds of pre-computation such as aggregation, complex function/expression evaluation, and more. This is because the data in the denormalized column is derived by a subquery that can be as complex as needed.

B. Projections and Live Aggregate Projections (LAP)

Vertica's projections are used to physically store data by specifying sort order, segmentation and aggregation on tables, including FTs. Traditional projections, which allowed materialization through sort order and segmentation, also served as a table's physical storage layer and has been discussed in an earlier publication [15]. In this paper we focus on LAP, a recently supported type of projections to fulfill the need of materializing aggregation, one of the most expensive operations for analytical queries.

While FT can be refreshed on request, projections, including LAPs, are always kept up-to-date with their anchor tables. This two-layer model enable flexible yet simple maintenance, from both developer and DB administrator's points of view, which will be demonstrated in Section III and IV.

LAPs can be defined on top of FT. In that case they can also be defined with aggregations on denormalized columns. In the following example a LAP is created on top of the FT *store_sales_ft* to serve the example query:

```
CREATE PROJECTION store_sales_ft_agg AS
SELECT i_brand_id, i_brand, d_year,
       sum(ss_ext_sales_price) AS sum_agg
FROM store_sales_ft
GROUP BY d_year, i_brand, i_brand_id;
```

With the above LAP defined on the FT *store_sales_ft*, the example query can be rewritten to take advantage of the materialization as follows:

```
SELECT d_year, i_brand_id AS brand_id,
       i_brand AS brand, sum_agg
FROM   store_sales_ft_agg
ORDER BY d_year, sum_agg DESC, brand_id
LIMIT 100;
```

C. Customer success stories

Vertica's solution for materialized view is designed with customers' requests in mind. One of our core customers also worked closely with us in drafting the functional specification for the feature, bringing to us their valuable feedback about what they truly need.

The solution, with FT at the core, has received a lot of interests from customers. When we first presented our FT prototype to our key customers, all of the customers not only showed great interest but also said they had continually created that kind of materialized views manually and FT would reduce a lot of heavy overhead for them. A year after the FT feature was officially available, at our annual conference, many customers told us "Today we have over several dozen FT in our system, more than we had expected because the overhead and maintenance are nothing now, and the performance is awesome." These are the strongest confirmation of the usefulness and efficiency of our approach.

D. Contributions

1) We present Vertica's unique solution for materialized views, which takes advantage of Vertica's columnar architecture to provide users with flexibility, ease of use, and performance.
2) We describe the novel designs of LAP and FT, the core component of Vertica's materialization solution.
3) We present experimental results showing the performance advantage of using FT and LAP.

II. Background

In this section, we summarize the data model of Vertica, focusing on how data is stored physically.

A. Projections and Super Projection

Like all SQL-based database management systems, Vertica models user data as tables consisting of different columns (attributes). Vertica supports the full range of standard INSERT, UPDATE, DELETE SQL statements for logically inserting and modifying data as well as a bulk loader and full SQL support for querying.

Vertica physically organizes table data into *projections*, which are sorted subsets of the attributes of a table. Vertica is a columnar database, meaning each column of each projection is stored in its own files. Any number of projections with different sort orders and segmentation and subsets of the table columns are allowed. In practice, most customers have one *super projection*, which contains all columns of the anchor table, and several narrow, non-super projections. Each projection has a specific sort order on which the data is totally sorted. Projections may be thought of as a restricted form of materialized view [11], [20]. They differ from standard materialized views because they are the only physical data structure in Vertica, rather than auxiliary indexes.

B. Partitioning

Users can specify a table-level partition expression and let Vertica keep data segregated in physical structures such that all tuples within the same storage container evaluated to the same distinct value of the partition expression. Partition expressions are most often time related.

Data partitioning allows Vertica to perform various fast bulk operations like move, copy, swap, delete, merge, etc. Partition pruning during query plan helps speed up query performance. Refer to [15] for more details.

C. Segmentation - Cluster Distribution

Segmentation is a way to split tuples among nodes. Unlike *partitioning*, which is specified at table level, segmentation is specified for each projection, which can be (and most often) different from the sort order. Projection segmentation provides a deterministic mapping of tuple value to node and thus enables many important optimizations such as fully local distributed joins, efficient distributed aggregations, etc.

Projections can either be *replicated* or *segmented* on some or all cluster nodes. A replicated projection stores a copy of each tuple on every projection node. A segmented projection store each tuple on exactly one specific node, determined by the value of the segmentation expression defined on the projection, which often to be a hash over a set of columns. Vertica can automatically recommend best projection design with its built-in Database Designer [25].

D. Storage modes

Vertica supports two architectures: Shared nothing (Enterprise mode) and Shared Storage (Eon mode). When deployed in shared nothing mode, each node of the cluster stores a segment of the data, hence is responsible for both compute and storage. On shared storage mode, data is stored permanently in communal storage shared between all nodes of the cluster. On the shared storage, data is also segmented *shards*. A node gains access to a shard by subscribing to it. Because the cost to access data from the communal storage is generally higher than from the local storage, each node can set aside a space on its local disk called *depot* to cache data to improve performance. The full details of Vertica Data Model in Enterprise and Eon are covered in [15] and [24] respectively. Vertica's solution for materialized view, as presented in this paper, works the same way on both storage modes.

III. Flattened Tables

Vertica Flattened Tables (FTs) are a column-store specific solution for maintaining denormalized, big flat fact tables involving one or many dependent dimensions tables. The main goal was to solve the scalability issues resulting from using traditional star/snowflake schema model which caused query slowdowns because of numerous joins and higher resource requirement. This normally gets even worse when concurrency increases. Existing solutions of keeping one big flat fact table to get rid of joins were not feasible mainly because of their bad update performance when the data in the dimension table changes. In most cases, a change in the dimension table's data only affects a small subset of denormalized columns in the big flat fact table, hence only those columns truly need to be refreshed. FT takes advantage of Vertica's distributed and columnar architecture to provide a solution for not only the update performance challenge but also various other use-cases.

A. Configuring Flattened Tables

1) Creating Flattened Tables: Creating a FT in Vertica is just like creating a regular table except that we also allow columns to be defined as query results from other tables. Vertica extends the SQL standard DEFAULT keyword to allow sub-queries to be used as any table column's definition. The following example illustrates how to define a denormalized column i_brand to a TPC-DS $store\_sales$ table, where column i_brand is denormalized from the $item$ table.

```
CREATE TABLE store_sales_ft
(
    ss_sold_date_sk integer,
    ss_item_sk integer not null,
    ss_ext_sales_price decimal(7,2),
    ...
    i_brand char(50) DEFAULT
        (SELECT i_brand FROM item
         WHERE ss_item_sk = i_item_sk)
);
```

2) Add / Alter column with DEFAULT expression: Vertica provides flexible schema evolution DDL that allows a user to define FT column definitions separately from the create table statement, even if the table is already filled with data. To add a new column with default values, the syntax **ALTER TABLE ... ADD COLUMN ... DEFAULT ...** is used. If the table is already filled with data, every row of the new column will automatically be populated with a value computed by the default expression using the table's corresponding row and dimension table rows.

Similarly, **ALTER TABLE ... ALTER COLUMN ... SET DEFAULT ...** can be used to assign an existing column with extended default expression. However, the existing data in the table will not be affected by this operation. To refresh the target column of the existing table based on the new column expression, the user must use the **SET USING** keyword (details in Section III-C) to define the column expression.

B. Data loading on Flattened tables

Once all FT columns are properly defined, using FT for queries and data loading is straightforward. Since querying a FT is just like querying a regular table, this section will focus on data loading. Non data loading DMLs such as DELETE, UPDATE and MERGE statements are not yet optimized for FTs. FT has its own unique approach to data modification in denormalized columns called ”refresh columns” which will be described in Section III-C.

When data is loaded to non-flattened columns, the flattened columns with DEFAULT expressions defined will be automatically calculated and loaded. Using the $store\_sales\_ft$ example above, the user does not need to explicitly mention the denormalized column i_brand in the DML but can let Vertica populate it automatically via the default expression. For every new row loaded into this table, Vertica will join the row with the dimension $item$ table using the join key $ss\_item\_sk = i\_item\_sk$. If there is exactly one match, the i_brand column value in the $item$ dimension table will be populated into the i_brand column in the flattened fact table; if there is no match, NULL values will be populated; if there are multiple matches, the DML will rollback with an error.

The mechanism behind the scene is a query rewrite. Vertica detects implicit column defaults needing to be populated. These column's default expressions will be plugged into the INSERT statement:

```
INSERT INTO store_sales_ft
SELECT ..., (SELECT i_brand FROM item
             WHERE ss_item_sk = i_item_sk)
FROM ...
```

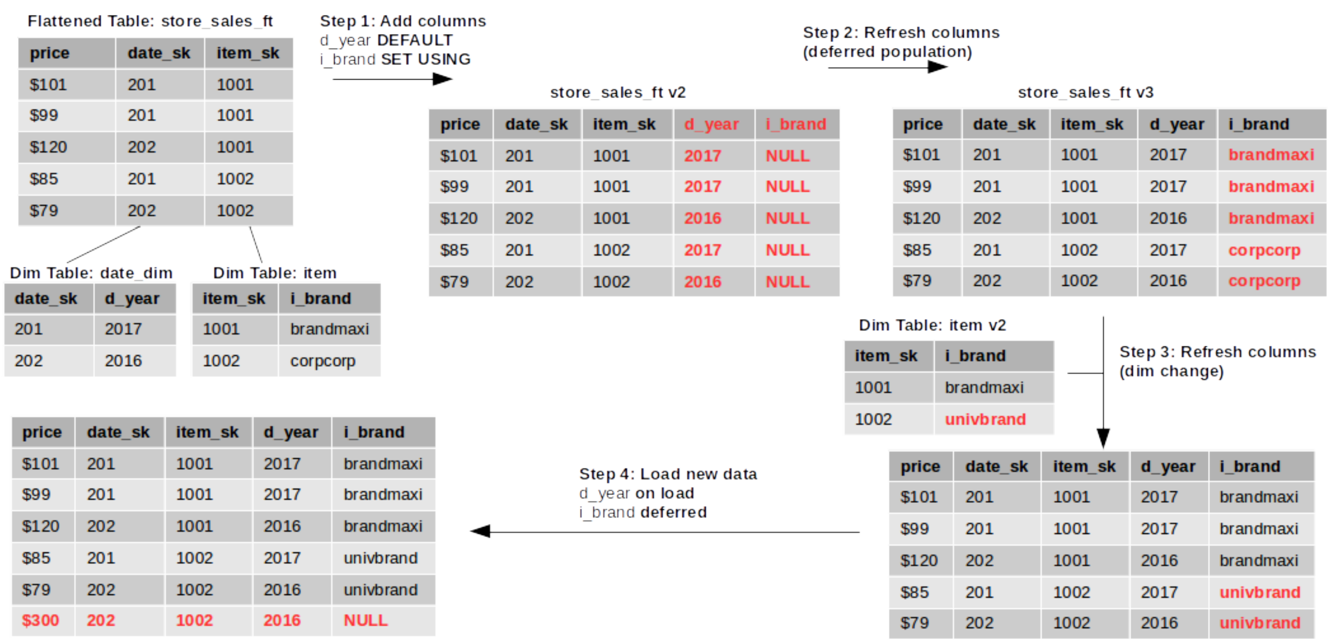
Vertica's Optimizer [22] [23] will produce a plan , including picking the best projections for the join, choosing the best join strategy and applying many other optimizations.

There are two main advantages of using FT instead of manually maintaining denormalized tables. Firstly, the schema dependencies are unhooked from the DML definition. Users who write DML statements do not need to remember the semantic meaning for those denormalized columns. In addition, if any denormalized column needs to be modified, the database administrator does not need to change every DML query. Secondly, loading data from the external sources become more efficient with FT because the denormalized values are populated on-the-fly without an intermediate staging table.

C. ”SET USING” and Column-wise Refresh

Section III-B described how FT automatically populate denormalized values when columns are defined with DEFAULT sub-queries. However, some users would like denormalized value population to be deferred and avoid paying any overhead at data loading time. It is also possible that the data in some dimension tables may change, making the denormalized value in the FT become stale. To handle either situation, an efficient refresh mechanism is required.

Vertica FT provides a unique solution called ”refresh column” to modify flattened column data on demand. This mechanism takes the advantage of Vertica's column store architecture, where individual column can be refreshed individually without touching the rest of the columns. FTs in real world scenarios can be expected to contain hundreds of columns. Hence restricting data modification to those affected columns has significant performance advantage.



Flattened Table: store_sales_ft

price	date_sk	item_sk
$101	201	1001
$99	201	1001
$120	202	1001
$85	201	1002
$79	202	1002

Dim Table: date_dim

date_sk	d_year
201	2017
202	2016

Dim Table: item

item_sk	i_brand
1001	brandmaxi
1002	corpcorp

store_sales_ft v2

price	date_sk	item_sk	d_year	i_brand
$101	201	1001	2017	NULL
$99	201	1001	2017	NULL
$120	202	1001	2016	NULL
$85	201	1002	2017	NULL
$79	202	1002	2016	NULL

store_sales_ft v3

price	date_sk	item_sk	d_year	i_brand
$101	201	1001	2017	brandmaxi
$99	201	1001	2017	brandmaxi
$120	202	1001	2016	brandmaxi
$85	201	1002	2017	corpcorp
$79	202	1002	2016	corpcorp

Dim Table: item v2

item_sk	i_brand
1001	brandmaxi
1002	univbrand

price	date_sk	item_sk	d_year	i_brand
$101	201	1001	2017	brandmaxi
$99	201	1001	2017	brandmaxi
$120	202	1001	2016	brandmaxi
$85	201	1002	2017	univbrand
$79	202	1002	2016	univbrand

price	date_sk	item_sk	d_year	i_brand
$101	201	1001	2017	brandmaxi
$99	201	1001	2017	brandmaxi
$120	202	1001	2016	brandmaxi
$85	201	1002	2017	univbrand
$79	202	1002	2016	univbrand
$300	202	1002	2016	NULL

Fig. 2: A comprehensive example of FT schema / data evolution using a simplified TPC-DS schema setup. Each of the four steps illustrates a common operation of FT: Step 1 - add denormalized columns; Step 2 - deferred population; Step 3 - column-wise update; Step 4 - data loading. The modified parts after each step are highlighted in RED.

To use the "refresh columns" feature, user first define the column definition just like the extended DEFAULT syntax, but instead use **SET USING** keyword, in particularly:

```
CREATE TABLE store_sales_ft
(
    ss_item_sk integer not null,
    ss_ext_sales_price decimal(7,2),
    ...
    i_brand char(50) SET USING
        (SELECT i_brand FROM item
         WHERE ss_item_sk = i_item_sk)
);
```

Just as with the **DEFAULT** keyword, the **SET USING** attribute can be added, modified, and removed from the FT using "ALTER TABLE" command.

Columns with **SET USING** definitions will not be automatically populated values during DMLs. Instead, whenever the user would like to refresh the denormalized column based on latest data in dimension tables, they can call the following meta-function:

```
SELECT refresh_columns('store_sales_ft',
                       'i_brand',
                       'REBUILD');
```

The following example in Figure 2 uses some sample data to illustrate how different FT features work together. The example is composed of one fact table *store_sales* and two dimension tables *date_dim* and *item*[1]. Both dimension tables contain a key column referenced by a foreign key column in the fact table and a value column that need to be denormalized into the fact table.

Step 1: Assuming that user calls **ADD COLUMN** command to add two denormalzed columns *d_year* and *i_brand* respectively, while the former materializes *date_dim.d_year*, the latter materializes *item.i_brand*, and PK-FK joins are used in both cases. If *d_year* column is defined as DEFAULT, the up-to-date denormalized *d_year* values will be populated for each row; if *i_brand* column is defined as SET USING, the new column will be filled with default NULL values. Of course adding column *i_brand* will be slower than column *d_year* as it need to run the join query plan and populate the data.

Step 2: The user invokes refresh columns on column *i_brand*. Vertica will populate the new column data and replace the old NULL values. This process does not touch the data files of the rest of the columns.

Step 3: Sometimes the dimension table gets slightly changed. In this case one row in the *item* table has the *i_brand* value which got updated. The denormalized column *i_brand* will catch up the new *i_brand* value after the user invokes the refresh columns command again. As before, this process does not touch any other columns.

Step 4: When new rows are loaded to the table, denormalized column *d_year* is automatically populated with value but *i_brand* is filled with NULL value. User can choose to run

[1]From now on, for brevity, in all figures we remove the *ss* prefix in the names. Also, *sold_date_sk* is shortened to *date_sk*, and *ext_sales_price* to *price*).

refresh columns on i_brand column or wait for more rows to be loaded.

D. *Partition-based Column-wise Refresh*

Most Vertica users partition their big fact tables by time-related key (e.g, loading date as 'YYYY-MM-DD' format). The more recently loaded data are usually associated with larger partition key. In the context of FT use case, user would like to defer populating denormalized columns for recent loaded partitions with "refresh_columns" feature. The user will never want to refresh the entire column of the table, which may contain billions of rows of old data collected for a couple of years. The performance is obviously the main reason but also it introduced a lot of redundant work, as the non-recent partitions may already have denormalized columns populated. Vertica also allows users to call "refresh_columns" with additional two arguments indicating the partition range of the data they would like to refresh, while the data outside the provided partition range will not be touched, as the following:

```
SELECT refresh_columns('store_sales_ft',
                       'i_brand',
                       'REBUILD',
                       '<min_partition_key>',
                       '<max_partition_key>');
```

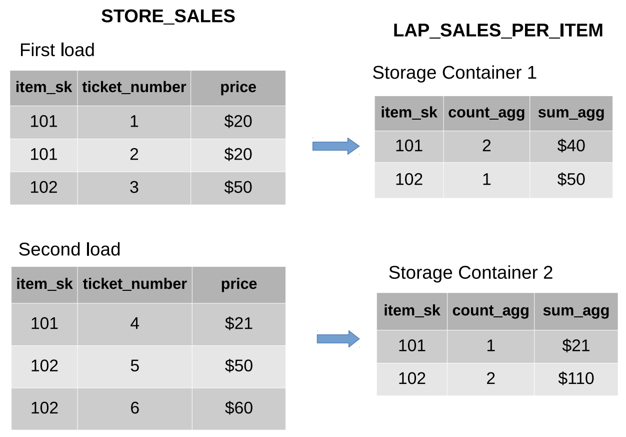
As expected, our in-house performance tests have proved that the time to refresh a column is proportional to the data size (row count) within the provided partition range. The performance benefit can be even more significant with the Shared Storage mode (mentioned in Section II-D), as recent partitions are most likely to be available in the local cache (a.k.a. depot). As a result, refreshing columns for most recent partitions can be quite light-weighted. Being light-weighted opens a door to more enhanced usability and automation. For example, Vertica can keep track of the last refreshed partition such that the next time column refresh will only touch the data loaded after that, without user to explicitly remember and specify the partition range in the "refresh_columns" call above. In addition, Vertica can also allow user to register timer service to automatically run column-wise refresh on latest partitions at idle time period (e.g., at night).

IV. Live Aggregate Projections

Live Aggregate Projections (LAPs) are a special type of projection which materialize pre-calculated aggregations on columns selected by the user. Similar to regular projections, LAPs are also always up-to-date and support auto-rewrite to use them in queries. In this section we first explain the usage of LAPs, then discuss their implementation in Vertica.

A. *Define and use LAP*

1) Create LAP: Creating a LAP follows the same syntax used to create a normal projection, where a query specifies how the projection's data is derived from the anchor table. In the following example, which is based on the schema presented in Figure 1, we define a LAP for table $store\_sales$, which



First load

item_sk	ticket_number	price
101	1	$20
101	2	$20
102	3	$50

Storage Container 1

item_sk	count_agg	sum_agg
101	2	$40
102	1	$50

Second load

item_sk	ticket_number	price
101	4	$21
102	5	$50
102	6	$60

Storage Container 2

item_sk	count_agg	sum_agg
101	1	$21
102	2	$110

Fig. 3: An example of how data is partially aggregated and stored in LAP

materializes the number of sale transactions and the total sale price for each item.

```
CREATE PROJECTION lap_sales_per_item AS
SELECT ss_item_sk,
       count(*) AS count_agg,
       sum(ss_ext_sales_price) AS sum_agg
FROM store_sales
GROUP BY ss_item_sk;
```

2) Auto query rewrite to use a LAP: Once a LAP is created, the Optimizer will consider picking LAP for answering the query. If query can be rewritten using LAP, then it will automatically do as shown in following example:

```
SELECT ss_item_sk,
       sum(ss_ext_sales_price) AS sum_agg
FROM store_sales
GROUP BY ss_item_sk
HAVING sum(ss_ext_sales_price) > 500;
```

When generating an execution plan for the above query, Vertica Optimizer identifies LAP $lap\_sales\_per\_item$ defined on top of $store\_sales$ which materializes the aggregation that the query needs. Therefore, the Optimizer will rewrite the query to use the LAP, as follows:

```
SELECT ss_item_sk, sum_agg
FROM lap_sales_per_item
WHERE sum_agg > 500;
```

B. *Incremental live updates on LAP*

In Vertica, a table can have one or more projections. When data is loaded into a table through INSERT or COPY (bulk load) command, the appropriate data of each column is computed and written to the right projections.

When data is loaded in to a table, multiple files are created one per each column within a projection. For LAP, such as $lap\_sales\_per\_item$ as defined in Section IV-A, the data is pre-computed **only on the newly loaded part** before writing to the new files. As a result, the data in LAP will be stored as partially aggregated.

Figure 3 illustrates this design. When the first batch of data is loaded into table $store\_sales$, a copy of it is aggregated and stored in Storage Container 1 of the table's LAP $lap\_sales\_per\_item$. When the second batch is loaded, a second storage container is created for $lap\_sales\_per\_item$, which contains the data aggregated from the second load only. Note that after the second load, there are a partially aggregated values of $count\_agg$ and sum_agg for each item in both of the storage containers.

This design brings two main benefits: Firstly, the data loading does not affect the existing data files, so it does not block queries reading from the corresponding LAP. Secondly, the cost of loading data is reduced, as it does not need to fully aggregate the whole data set.

The partially aggregated data is then fully aggregated via two ways:

1) Merge-out tasks: Vertica periodically runs **merge-out** tasks to merge multiple data files that belong to the same projection. For LAP, these tasks also combine the partially aggregated values in the merged files.
2) Queries: When a query read from LAP, it will apply a final aggregation operation on top of the data to produce correct results. This final aggregation operation is expected to be insignificant.

Vertica currently supports MAX, MIN, $COUNT$, and SUM as aggregate function used in LAP. AVG can be easily calculated from SUM and $COUNT$ with little overhead. While loading new data into a table never requires data in its LAPs to be recomputed, deleting data requires refreshing of the LAPs' partitions touched by the delete operation[2]. Note that deleting a whole partition of a table is very efficient and does not involve refreshing of the LAPs.

C. LAP on FT

To support materializing aggregation that involved columns from other tables, we allow LAPs to be defined on top of an FT. LAPs on an FT are defined in the same way as LAPs on a normal table, and are kept up to date with the anchor FT. This means when a denormalized column is refreshed on the FT, any LAP that involves that column in its definition must also be refreshed at the same time.

Figure 4 shows an example with the FT $store\_sales\_ft$ as defined in the example in Section I. The column i_brand in this FT is a denormalized column, which is populating through the join between $store\_sales\_ft$ and $item$. The FT has a LAP named $store\_sales\_ft\_agg$ which materializes the total sale price of each branch in each year. As such, the definition of $store\_sales\_ft\_agg$ involves the denormalized column i_brand. When the dimension table $item$ is updated and users issue a $refresh\_columns$ command on the column i_brand of the FT, the refresh process is as follows:

- For any normal projections of the FT that has column i_brand, update it by rebuilding the corresponding data files of the column in each projection. For normal projections only the column requested by the $refresh\_columns$ command is updated. Also, only the part of the column's data in the relevant partitions is refreshed. The other columns and partitions are not touched.
- Find all LAPs that has the refreshed column in their definition, which includes only $store\_sales\_ft\_agg$ in this example. For each of the LAPs, execute an optimized plan to recompute the aggregation, then replace all the old data files in the relevant partition(s) of the LAP with the new ones containing the latest values.

While normal projections of the FT only need to refresh the corresponding column, the LAP needs to rebuild the whole partition(s) which contains the updated values. This is because although the values of the other columns (d_year and sum_agg in the example) stay the same, there is no way to guarantee that the ordering of the values aligns correctly with that in the newly-created data files of the refreshed columns. In addition, there is no way to map newly populated aggregations to the existing partially aggregated storage containers. This whole-partition refresh, in practice, has acceptable cost, because LAPs contain aggregated data and therefore are typically much smaller than the raw data in the FT. We will demonstrate this claim with experimental result in Section V-A.

D. Extensions of LAP

With LAP, we also support materializing Top-K (Top-K LAPs) and user-defined aggregation (UDTF LAPs). Due to space limitation, we refer readers to Vertica documentation [7] for more information on Top-K LAPs and UDTF LAPs.

V. Experimental Results

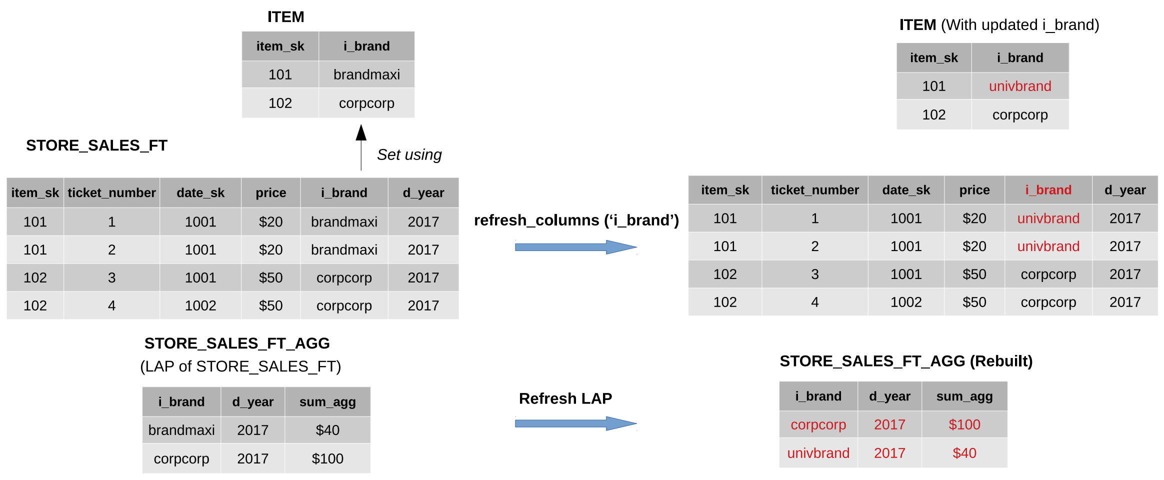
In order to demonstrate the strength of FT and LAP, we need to answer three questions:

- How much faster is refreshing individual columns than rebuilding the whole table?
- How much performance overhead is there to populate denormalized columns while loading new data to FT?
- Will queries run significantly faster with FT and LAP than running on regular fact and dimension tables?

To answer these questions, we ran three experiments on a 4-node, HPE Proliant G9 Server cluster, with TPC-DS data loaded at scale factor 100.

Three tables were used in these experiments: one fact table $store\_sales$ and two dimension tables $item$ and dim_date. At scale factor 100, the fact table contains about **287M** rows of data. $store\_sales$, which includes 23 columns, is extended by three denormalized columns: i_brand, i_brand_id and d_year to make a FT. These columns are necessary for TPC-DS query 3, which is used in the experiments. In addition, a LAP is created on top of the FT which computes aggregation $ss\_ext\_sales\_price$ for each group identified by these three denormalized columns. The LAP contains about **408K** aggregated rows. Our full script for all experiments is available upon request. All experiments were run three times and average timings were collected and reported.

[2]In Vertica, updating a row is equivalent to deleting the old row and inserting a new one. Hence the partitions touched by the delete operation needs to be refreshed



ITEM

item_sk	i_brand
101	brandmaxi
102	corpcorp

STORE_SALES_FT

item_sk	ticket_number	date_sk	price	i_brand	d_year
101	1	1001	$20	brandmaxi	2017
101	2	1001	$20	brandmaxi	2017
102	3	1001	$50	corpcorp	2017
102	4	1002	$50	corpcorp	2017

STORE_SALES_FT_AGG
(LAP of STORE_SALES_FT)

i_brand	d_year	sum_agg
brandmaxi	2017	$40
corpcorp	2017	$100

ITEM (With updated i_brand)

item_sk	i_brand
101	univbrand
102	corpcorp

item_sk	ticket_number	date_sk	price	i_brand	d_year
101	1	1001	$20	univbrand	2017
101	2	1001	$20	univbrand	2017
102	3	1001	$50	corpcorp	2017
102	4	1002	$50	corpcorp	2017

STORE_SALES_FT_AGG (Rebuilt)

i_brand	d_year	sum_agg
corpcorp	2017	$100
univbrand	2017	$40

Fig. 4: An example of refresh column on FT which involves refreshing a LAP

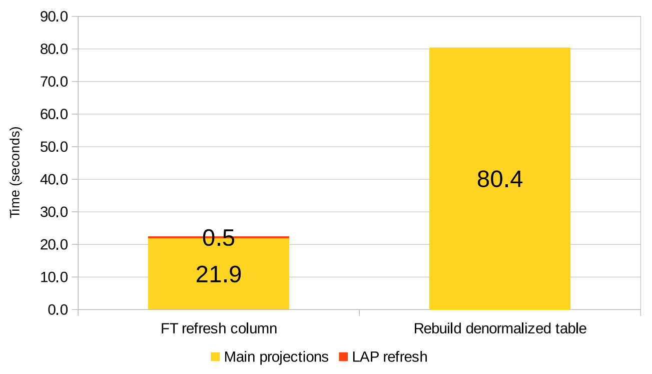


Fig. 5: Performance comparison of single-column FT refresh column and populating data in new denormalized table.

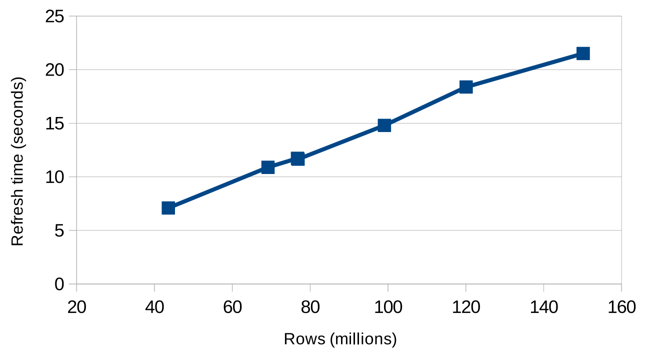


Fig. 6: The time to run refresh column operation is proportional to the data size involved in the given partition range.

A. Experiment 1: Column-wise refresh performance

This experiment is designed to demonstrate the performance benefit of using Vertica FT column-wise refresh to update denormalized column values when referenced dimension tables have some data change. In this experiment, we assume data in the i_brand column in dimension table $item$ has changed and compare two options to bring the column i_brand in the denormalized fact table up-to-date: 1) use Vertica FT's "refresh_columns" command; 2) re-load the denormalized table using INSERT SELECT.

The test result is illustrated in Figure 5. Using FT refresh columns to update column i_brand takes 22.4 seconds, about 4 times faster than the 80.4 seconds to re-insert the joined result from the $store\_sales$ table with the $item$ table into a denormalized table. Of the 22.4 seconds of the FT option, refreshing the main projection took most of the time (21.9 seconds) while the consequent LAP refresh only took 0.5 seconds. This proves that even if the whole LAP need to be rebuilt because one of its columns was changed, the actual overhead is small because LAP size is usually much smaller than the full table size. In this case the LAP is about 700x smaller in terms of rows and 6.5x smaller in terms of columns. The performance boost in this case is around 4x. In real world cases the benefit will be even more significant as many denormalized fact tables are much wider, scaling out to hundreds of columns.

To demonstrate the theory in section III-D, we also tested the performance of refresh column operation on partition ranges with different data size (number of rows). The result in Figure 6 clearly showed that the performance is proportional to the data size of the given partition range. This means that even if the whole fact table is huge (billions of rows), as long as the newly loaded data is much smaller, refreshing denormalized columns for the new data can be very light weighted.

B. Experiment 2: Data loading overhead

While Experiment 1 is designed to simulate a deferred update use case which is very common in practice, in many scenarios it's also necessary to provide denormalized column values immediately on load of new fact rows. For that purpose, the user will define columns as DEFAULT rather than SET USING. Hence it is essential to evaluate the overhead of joining with the dimension table column when loading data

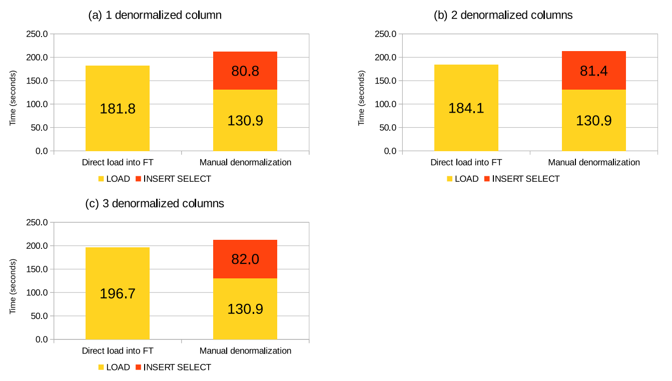


Fig. 7: Performance comparison of loading external data into FT vs. manually loading into denormalized table.

into the FT. In this experiment, we compare two options: 1) data is loaded into FT, *store_sales_ft*, where the joins implicitly specified by the table definition; 2) data is directly loaded into the fact table, *store_sales*, and then INSERT SELECT into denormalized table where those joins explicitly specified in the SELECT clause.

The test result is illustrated in Figure 7. Loading new rows to FT takes 182.8, 184.1, and 196.7 seconds when there are 1, 2, and 3 denormalized columns respectively in the FT to be populated automatically during data loading. This is about 40% overhead comparing to direct loading into the fact table (130.9s), which is expected because FT need to spend extra resources to calculate the joins during data load. On the other hand, it takes 80.8, 81.4, and 81.9 seconds to populate a denormalized table with 1, 2, and 3 denormalized columns respectively. So from an end-to-end perspective, loading to FT is faster than manual denormalization. Moreover, without sacrificing performance, the process that directly loading external data to Vertica FT has quite a few usability advantages: firstly, all fact-dim relationships are defined in table definition so the loading script can be much simpler; secondly, loading to FT is a one-step process which is transactional and does not require intermediate storage for the staging fact table.

In certain scenario where user prefers to avoid any overhead they can defer the update using the refresh column functionality which has been demonstrated to be also efficient in Section V-A. Users have the flexibility to choose the mechanism that best fits the needs of their application.

C. *Experiment 3: query performance benefit*

The last experiment is designed to demonstrate that queries can actually benefit from both the materialized joins and aggregations maintained by FT and LAP. In this experiment, the simplified TPC-DS query 3 and its equivalent FT version (shown below) was measured among three different settings: 1) FT + LAP setup; 2) FT only setup; and 3) regular fact + dimensions setup.

```
SELECT d_year, i_brand_id AS brand_id,
       i_brand AS brand,
       sum(ss_ext_sales_price) AS sum_agg
FROM store_sales_ft
GROUP BY d_year, i_brand, i_brand_id
ORDER BY d_year, sum_agg DESC, brand_id
LIMIT 100;
```

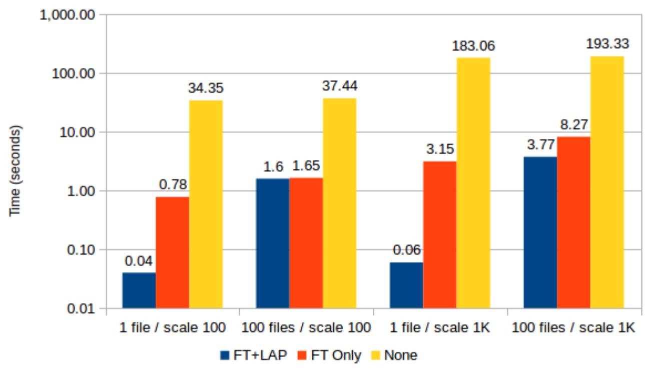


Fig. 8: Performance comparison of running simplified TPC-DS query 3 and its equivalent FT version against different levels of Vertica materialization mechanism: 1) FT with LAP created; 2) FT only; and 3) traditional fact and dimension tables.

Results, as shown in Figure 8, demonstrate that in all scenarios using FT+LAP can significantly improve the query performance if the query can benefit from the materialized pre-joined and pre-aggregated results. The experiment was run with different scales of data size: 100GB and 1TB. For each scale setup, we loaded data in 1 batch and 100 batches, the latter helping us see the performance impact of multiple physical files which haven't yet been through mergeout. In all scenarios, running the query on the FT+LAP setup is significantly faster (up to 1000x) than with regular setup.

When FT is used without a LAP, query performance is between the FT+LAP and the normalized setup. The difference between the three setups varies in different cases. When data is loaded in 1 batch, the query performance of FT-only is in the middle of FT+LAP and normalized setup, with significant margin on both sides; when data is loaded in 100 batches, the query performance of FT-only is close to FT+LAP. This is due to the additional work needed to combine the load batches during the query - with 100 files to combine, there is not as much benefit to pre-aggregation. The results showed that both FT materialization for joins and LAP materialization for aggregations can help improving query performance rather than one materialization dominating the other.

VI. Related Work

Efficient view maintenance has been a major focus of earlier works on materialized views (e.g., [11], [12], [19]. The ones most closely related to our work on LAPs are those providing incremental maintenance for materialized view with aggregates ([12]–[14], [16]–[18], [26]). The basic approach in all these papers is to compute a delta summary of the changes in a propagation phase, then apply the delta summary to the view in an installation phase.

There have been many industrial implementations of MVs, each with their own rules of when incremental live update on the views are supported. MIN/MAX and TopK are not supported in indexed views in Microsoft SQL Server [3]. MIN/MAX are also not supported in IBM DB2 materialized query tables with immediate refresh [1]. IBM Netezza [2] forbids aggregates in materialized views altogether. Oracle DB [4] supports incrementally refreshed materialized views with non-distributive aggregates via materialized view log tables (MV logs). MV logs store update deltas to the master tables which are then used to do incremental refreshes of views. Teradata supports aggregate join indexes (i.e. MVs with aggregates) with MIN/MAX since 14.10 [6], and as far as we know, it supports it via applying of update deltas to aggregate join indexes in a way similar to Oracle. VoltDB's [21] implementation of incrementally refreshed MVs with non-distributive aggregates is a work in progress. VoltDB in some cases supports updates for MVs with MIN/MAX via indexing source tables on grouping columns used in the aggregates. It allows quickly finding all the rows in the groups affected by the update to compute new min/max values for those groups.

There are two major differences between the above works and Vertica's approach of FTs and LAPs. Firstly, they maintain only one delta view for each materialized view at a time, while for LAPs we can have many of them before a merge-out operation is triggered. This is more suitable for Vertica's column-store infrastructure, since in-place update is costly. Oracle and Teradata maintain multiple deltas, but they need to use auxiliary data structures while Vertica uses normal data storage. Secondly, the MVs discussed in the above papers does not have the flexibility of Vertica's FT to add, drop, and refresh individual columns as needed. The MVs also need to be defined separately, while in Vertica users can extend any existing table to turn it into a FT.

VII. Conclusions

We presented in this paper a novel approach in Vertica to support the basic functionality of MV, via FT and LAP. Our approach, taking advantage of Vertica's columnar architecture, provides efficiency and flexibility beyond the traditional implementation of MV. With FT, users can add denormalized columns to a fact table when needed instead of having to define multiple separate MVs, reducing storage redundancy and maintenance costs. In addition, FT's columns can be refreshed individually on a single partition as needed, which, as we have shown in our experiments, gives much better performance than whole table/MV refresh.

The design of our materialized view solution were based on real-life customer cases. Our rules regarding how to keep data in FT and projections including LAPs in sync with source tables are simple and consistent, making it easy to use and maintain. We have been continuously confirmed by our customers that our solution has help them speed up queries while reducing cost and maintenance effort.

VIII. Acknowledgments

We especially thank one of our key customers, who prefers to stay anonymous, for not only listing clear requirements but also working with us on the preliminary designs of FT. These two features would not be available without a tremendous effort of cross functional team involving developers, QA, and documentation teams, including authors of this paper, Natalya Aksman, Carl Gerstle, Sarah Lemaire, Michael Kronenberg, Ben Vandiver, and former Verticans, Alper Okcan, Vivek Bharathan, and Zhijie Wen.

References

[1] IBM DB2, http://www-01.ibm.com/software/data/db2/.
[2] IBM Netezza, http://www-01.ibm.com/software/data/netezza/.
[3] Microsoft sql server, http://www.microsoft.com/en-us/server-cloud/products/sql-server/try.aspx.
[4] Oracle Database, https://www.oracle.com/database/index.html.
[5] Refresh Materialized Views in Oracle, https://www.oracle.com/database/index.html.
[6] Teradata, http://www.teradata.com.
[7] Vertica documentation, https://www.vertica.com/docs/9.1.x/HTML/.
[8] Voltdb, http://voltdb.com/.
[9] Tpc benchmark ds - standard specification, June 2018.
[10] A. G. Bello, K. Dias, A. Downing, J. Feenan, J. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in oracle. In *VLDB Proceedings*, 1998.
[11] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *VLDB*, 1991.
[12] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *ACM SIGMOD Proceedings*, SIGMOD '95, New York, NY, USA, 1995. ACM.
[13] H. He, J. Xie, J. Yang, and H. Yu. Asymmetric batch incremental view maintenance. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 106–117. IEEE, 2005.
[14] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance issues in incremental warehouse maintenance. In *In VLDB Proceedings*. Stanford InfoLab, 1999.
[15] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. In *VLDB Proceedings*, 2012.
[16] C.-P. Li and S. Wang. Efficient incremental maintenance for distributive and non-distributive aggregate functions. *Journal of Computer Science and Technology*, 21(1), 2006.
[17] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *VLDB Proceedings*. VLDB Endowment, 2002.
[18] D. Quass. Maintenance expressions for views with aggregation. 1996.
[19] K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: Asynchronous incremental view maintenance. In *In ACM SIGMOD Proceedings*, SIGMOD '00, New York, NY, USA, 2000. ACM.
[20] M. Staudt and M. Jarke. Incremental Maintenance of Externally Materialized Views. In *VLDB*, 1996.
[21] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2), 2013.
[22] N. Tran, S. Bodagala, and J. Dave. Designing query optimizers for big data problems of the future. *PVLDB*, 6(11), 2013.
[23] N. Tran, A. Lamb, L. Shrinivas, S. Bodagala, and J. Dave. In I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, editors, *ICDE*, 2014.
[24] B. Vandiver, S. Prasad, P. Rana, E. Zik, A. Saeidi, P. Parimal, S. Pantela, and J. Dave. Eon mode: Bringing the vertica columnar database to the cloud. In *ACM SIGMOD Proceedings*, New York, NY, USA, 2018. ACM.
[25] R. Varadarajan, V. Bharathan, A. Cary, J. Dave, and S. Bodagala. Dbdesigner: A customizable physical design tool for vertica analytic database. In I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, editors, *ICDE*, 2014.
[26] J. Zhou, P.-A. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *VLDB Proceedings*, 2007.