Foreign Keys Open the Door for Faster Incremental View Maintenance

CHRISTOFOROS SVINGOS*, National & Kapodistrian University of Athens, Greece
ANDRE HERNICH, Amazon Web Services, Germany
HINNERK GILDHOFF, Amazon Web Services, Germany
YANNIS PAPAKONSTANTINOU†, Databricks & University of California, San Diego, USA
YANNIS IOANNIDIS, Athena Research Centre National & Kapodistrian University of Athens, Greece

Serverless cloud-based warehousing systems enable users to create materialized views in order to speed up predictable and repeated query workloads. Incremental view maintenance (IVM) minimizes the time needed to bring a materialized view up-to-date. It allows the refresh of a materialized view solely based on the base table changes since the last refresh. In serverless cloud-based warehouses, IVM uses computations defined as SQL scripts that update the materialized view based on updates to its base tables. However, the scripts set up for materialized views with inner joins are not optimal in the presence of foreign key constraints. For instance, for a join of two tables, the state of the art IVM computations use a UNION ALL operator of two joins — one computing the contributions to the join from updates to the first table and the other one computing the remaining contributions from the second table. Knowing that one of the join keys is a foreign-key would allow us to prune all but one of the UNION ALL branches and obtain a more efficient IVM script. In this work, we explore ways of incorporating knowledge about foreign key into IVM in order to speed up its performance. Experiments in Redshift showed that the proposed technique improved the execution times of the whole refresh process up to 2 times, and up to 2.7 times the process of calculating the necessary changes that will be applied into the materialized view.

ACM Reference Format:
Christoforos Svingos, Andre Hernich, Hinnerk Gildhoff, Yannis Papakonstantinou, and Yannis Ioannidis. 2023. Foreign Keys Open the Door for Faster Incremental View Maintenance. 1, 1, Article 40 (May 2023), 25 pages. https://doi.org/10.1145/40

1 INTRODUCTION

Serverless cloud-based warehousing systems are becoming increasingly popular because of their high elasticity and flexible billing. In serverless cloud-based warehouses the storage servers are separated from the compute ones, in order to scale automatically according to the queries demands. The users are only paying for the resources that their queries consume. These architectural choices have led to query processing techniques that do not need and/or use conventional main memory indices. Furthermore, they rely exclusively on cheap, "big block"-based cloud storage, thus delivering tremendously lower cost of ownership to customers.

Serverless cloud-based warehousing systems use materialized views in order to speed up predictable and repeated query workloads. Materialized views are database objects that store the

*Currently working for Amazon.
†Work done while at Amazon.

Authors' addresses: Christoforos Svingos, National & Kapodistrian University of Athens, Athens, Greece, csvingos@di.uoa.gr; Andre Hernich, Amazon Web Services, Berlin, Germany, ahernich@amazon.com; Hinnerk Gildhoff, Amazon Web Services, Berlin, Germany, hinnerk@amazon.com; Yannis Papakonstantinou, Databricks & University of California, San Diego, CA, USA, yannis.papakonstantinou@databricks.com; Yannis Ioannidis, Athena Research Centre National & Kapodistrian University of Athens, Athens, Greece, yannis@di.uoa.gr.

XXXX-XXXX/2023/5-ART40 $15.00
https://doi.org/10.1145/40

results of a query in order to not repeatedly recompute them when the same computations are needed. In order to be able to use materialized views in serverless cloud-based warehouses, we use the classic Incremental View Maintenance (IVM) techniques which were introduced for disk based relational database systems [4, 9, 12, 13, 19, 21, 24–26]. Classic IVM is required to efficiently bring a view up to date when its base tables are updated. More specifically, it computes the incremental changes and applies only them to the materialized views rather than recomputing its contents from scratch.

Workloads in data warehouses are characterized by queries with joins that follow the foreign key constraint. As a consequence, this is also true for materialized views in these workflows. A statistical analysis of 173 automatically extracted materialized views from the 99 queries of the TPC-DS benchmark showed us that 81% of them have at least one join that follows the foreign key constraint while 42% had three or more. This knowledge raises opportunities for further optimization of the refresh process. As an example, consider the materialized view V joining tables R and S, defined as $V = \sigma_{R.f=S.p}(R \times S)$. Assuming that column $R.f$ is a foreign key that references column $S.p$, which is the primary key of the relation S, we say that table R is the fact table, while table S is the dimension table of the materialized view V. For such kind of materialized views the next two Properties are followed:

Property 1.1. *Newly inserted tuples of the dimension table S can only be joined with newly inserted tuples of the fact table R.*

Proof. Consider the case that the materialized view V is up to date, and we insert one tuple into the dimension table S. In order for this Property to be violated, there must exist a tuple in the fact table R that joins with the newly inserted tuple. However, if such tuple existed in R, it would reference a non existing primary key, which violates the foreign key constraint. □

Property 1.2. *Newly deleted tuples of the dimension table S can only be joined with newly deleted tuples from the fact table R.*

Proof. In order not to violate the foreign key constraint semantics, a database should ensure that if we delete a tuple from a dimension table, all tuples from the fact table referencing this tuple should also be deleted. □

Despite the numerous works related to classic IVM, only a few [19, 26, 28] use the integrity constraints in order to speed up the materialized view's refresh process. The closest work to our use case is the optimization technique mentioned by Quass et al. [25]. In this work the authors use the previously defined Properties in order to accelerate only the process that calculates the set of insertions which should be applied into a materialized view. The ideas described there can be easily extended for deletions. We give an intuition on how this work takes advantage of these two Properties:

Example 1.3. Consider the following materialized view that uses tables defined in the TPC-DS benchmark:

```
CREATE MATERIALIZED VIEW V AS
SELECT *
FROM date_dim dd, store_sales ss, item i
WHERE dd.d_date_sk = ss.ss_sold_date_sk AND
      ss.ss_item_sk = i.i_item_sk;
```

The above query follows the star join schema. The *ss_item_sk* column is a foreign key of the *store_sales* table that references the *i_item_sk* column, the primary key of the *item* table. Also,

$\Delta(V)$	=	$(\Delta(date\_dim) \bowtie store\_sales' \bowtie item')$ *UNION ALL* $(date\_dim \bowtie \Delta(store\_sales) \bowtie item')$ *UNION ALL* ~~$(date\_dim \bowtie store\_sales \bowtie \Delta(item))$~~
$\nabla(V)$	=	~~$(\nabla(date\_dim) \bowtie store\_sales' \bowtie item')$~~ *UNION ALL* $(date\_dim \bowtie \nabla(store\_sales) \bowtie item')$ *UNION ALL* $(date\_dim \bowtie store\_sales \bowtie \nabla(item))$

Table 1. Delta computations for Example 1.1.

the *ss_sold_date_sk* column is a foreign key of the *store_sales* table that references the *d_date_sk* column, the primary key of the *date_dim* table.

Table 1 shows the computations needed in order to calculate the set of tuples that we should insert/delete into/from the materialized view V. As you can notice, the described computations execute two different queries in order to calculate the set of insertions ($\Delta(V)$) and the set of deletions ($\nabla(V)$). These queries use a UNION ALL operator with 3 branches. Each branch computes the set of insertions/deletions that should be applied into the materialized view V, based on the set of insertions/deletions that was applied into one of the three base tables. These queries use two different states of the base tables: the one refers to the base table before the changes occur and the other refers to the base tables with the changes applied. In our example we will use the "$'$" symbol to refer to the current state of a base table ($base\_table' = base\_table - \nabla(base\_table) + \Delta(base\_table)$).

Quass et al. [25] noticed that due to Property 1.1, the newly inserted tuples into the *item* or *date_dim* tables can only be matched with the newly inserted tuples of the *store_sales* table. Having this observation in their mind, they detected that the third UNION ALL branch from the query that calculates the $\Delta(V)$ can be eliminated since the join $store\_sales \bowtie \Delta(item)$ will produce zero results. Additionally, due to Property 1.2 we can also eliminate the first UNION ALL branch from the query calculates the $\nabla(V)$.

The previous example showed how the techniques described in the bibliography exploit the foreign key constraints in order to simplify delta computations. However, the described optimization doesn't fully exploit the foreign key constraints in order to produce the minimal possible computations. As we are going to see, we can: a) eliminate one more UNION ALL branch by rewriting the described equations in order to take advantage of the fact that the join $store\_sales \bowtie \Delta(date\_dim)$ and $store\_sales' \bowtie \nabla(item)$ will also produce zero results and b) apply the foreign key constraint optimization into the state of the art equations defined by Gupta et al. [12], which handle both insertions and deletions in one pass rather than separately computing the set of deletions and the set of insertions. More specifically, our contribution can be summarized as follows:

► We are the first describing delta computations that take advantage of the foreign key constraints and simultaneously handle both insertions and deletions in one pass rather than separately computing the set of deletions and the set of insertions. While in the literature there are a few works that describe delta computations which exploit foreign keys [16, 19, 26], all of them calculate separately

the set of insertions and the set of deletions, leading in equations that use two different queries and do double work to calculate the appropriate deltas.

► We are the first that describe equations which are minimal in the essence of the number of UNION ALL branches. More specifically, we propose an algorithm that produces delta computations with as many union branches as the number of relations whose primary key isn't used by any join that follows the foreign key - primary key relationship. This results in "delta" computations without a UNION ALL for queries that follow the snowflake join schema (and as a consequence, the star join schema).

► We provide proof of correctness for all of the described computations and we explicitly define cases where the described optimizations can't be applied. As we are going to see, there are some cases, like updates into dimension tables, that violate our integrity constraints assumptions and cause our equations to produce faulty results. In this work we explicitly define such cases and describe the process we follow in order to discover them.

► The proposed algorithm is based on widely used incremental view maintenance algorithms [12, 25] and is easy to implement. As a result, the improvements presented in this paper can be applied in many serverless cloud-based warehousing systems. Even in cases of materialized views for which not all conditions are met (except for foreign-key joins), the proposed approach allows for existing computations to seamlessly take over.

► We implemented the proposed optimization in Redshift serverless cloud-based warehousing system, and we ran experiments to evaluate it. The experiments showed that for a set of incrementally updated materialized views that followed various join schemas, the proposed technique sped up the whole refresh process up to 2 times and the process of calculating the "deltas", in particular, up to 2.7 times.

The described optimizations are most applicable to serverless, cloud-based warehouses, which do not feature conventional main memory indices. The benefits of the optimization are lesser in the presence of main memory index structures that enable rapid retrieval of the "fact" rows that join with a "dimension" key. They are still nontrivial, however, as in database systems featuring IVM that uses main memory indices [15–17, 22, 27, 28] the described optimization fully eliminates access to one or more indices that, if accessed, would return empty lists of matches for these accesses.

2 PRELIMINARIES

In this section we are going to represent some of the algorithms and terminology used throughout the rest of this work. First, we describe the "Counting Algorithm" introduced by Gupta et al. [12], next, we illustrate the way that Redshift internally stores its materialized views, and finally, we define the FK-Graph which will be used to represent the foreign key-primary key constraints in the materialized views' joins.

2.1 The Counting Algorithm

The basic idea in the "Counting Algorithm" is to keep a count of the number of derivations for each view tuple as extra information in the view. It uses the changes in the base tables and the old values of the base tables to produce as output the set of changes that need to be made to the view. Let's illustrate the counting algorithm using an example:

Example 2.1. Consider the following materialized view:

```
CREATE MATERIALIZED VIEW Hop AS
SELECT DISTINCT l1.s, l2.d
```

```
FROM Link l1, Link l2
WHERE l1.d = l2.s
```

Given Link = {(a,b), (b,c), (b,e), (a,d), (d,c)}, the view Hop evaluates to {(a,c), (a,e)}. The tuple Hop(a, e) has a unique derivation. Hop(a,c), on the other hand, has two derivations. The counting algorithm pretends that the view has duplicate semantics, despite the distinct operator, and stores these counts.

Suppose the tuple Link(a,b) is deleted. Then, we can see that the Hop can be recomputed as {(a, c)}. The counting algorithm infers that one derivation of each of the tuples Hop(a, c) and Hop(a, e) is deleted. The algorithm uses the stored counts to infer that Hop(a, c) has one remaining derivation and, therefore, only deletes Hop(a, e), which has no remaining derivations. □

The "Counting Algorithm" thus works by storing the number of alternative derivations of each tuple in the materialized view. As mentioned before, given a view V, the counting algorithm uses the changes made to the base tables and the old values of the base tables to produce as output the set of changes that need to be made to the view. This set of changes called summary deltas and from now on we will symbolize it as ($\mathfrak{D}(V)$). In summary deltas, insertions are represented by positive counts and deletions by negative counts.

In order to implement the "Counting Algorithm" the union operator $\uplus$ is defined over the sets of tuples with counts. Given two such sets S1 and S2, S1 $\uplus$ S2 is defined as follows:

(1) If tuple t appears in only one of S1 or S2 with a count c, then tuple t appears in S1 union S2 with a count c.
(2) If a tuple t appears in S1 and S2 with counts of c_1 and c_2 respectively, and $c1 + c2 \neq 0$, then tuple t appears in S1 $\uplus$ S2 with a count $c_1 + c_2$. If $c_1 + c_2 = 0$ then t does not appear in S1 $\uplus$ S2.

The Cartesian product operator is also redefined for relations with counts: when two or more tuples paired, the count of the resulting tuple is a product of the counts of the paired tuples. We symbolize this operator with the $\otimes$ symbol.

The "Counting Algorithm" gives the advantage of handling both insertions and deletions in one pass, rather than separately computing the set of deletions and the set of insertions. Thus, the "Counting Algorithm" and variants [12, 13, 18, 23] are used by a variety of commercial systems [3, 23, 29], including Redshift [], in order to incrementally update their materialized views. The "Counting Algorithm" is useful not only for maintaining views involving projection and duplicate elimination, but also for "Group-By-Aggregation". In this work, we focused on views of the form "Select-Project-Join", however, the same equations can be used by "Group-By-Aggregation" materialized views. The reader is referred to [13, 18] in order to find the methodology that should be followed for incremental maintenance of this kind of views.

2.2 Redshift's Internal View Representation

Figure 1 shows the set of tuples inside the *Link* table and the *Hop* materialized view of Example 2.1, as these were stored by Redshift after the changes were applied to them (*Link'*, *Hop'*). It also represents the summary deltas of *Hop* materialized view occur after the deletion of Link(a, b) tuple ($\mathfrak{D}(Hop)$). As it is shown, Redshift stores some extra information in its relations and materialized views which are kept hidden and used during the IVM process. More specifically, Redshift includes the *"count"* column inside each materialized view (see *Hop'*). This column stores the number of derivations for each tuple of the materialized view. Additionally, Redshift assigns a unique identifier to each tuple inside a relation, called RowId (see *Link'*). RowIds are used by "Select-Project-Join" materialized views in order to uniquely identify each of their tuples. More specifically, in the case of "Select-Project-Join" materialized view each row of the materialized view will be identified by the set of RowIds of the joined relations. In our example we use the distinct operator, so it is not

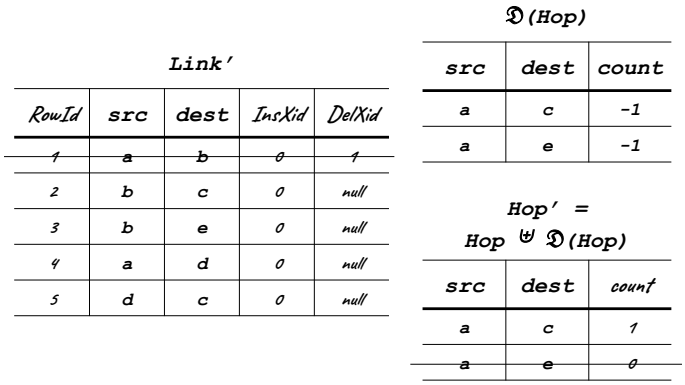


Fig. 1. Redshift's View Representation.

needed to store extra RowIds inside the view's internal table. Finally, all tuples inside a relation have chronologically ordered transaction ids assigned to them. The DelXid and the InsXid columns are used in order to identify the time that each tuple was inserted or deleted into each relation. In this way it is easy to find the tuples inserted or deleted between two refreshes.

2.3 The FK-Graph

In order to describe the foreign key constraints used by the joins of a materialized view, we define the FK-Graph [2, 20, 28]. The FK-Graph G = (N, E) is a directed Graph where N represents the set of the base tables and E represents the set of foreign key constraints between these tables. We consider as *base tables* relations which are used by the *FROM* clause of the SQL definition of the materialized view. We consider references to the same relation in the materialized view's *FROM* clause as different base tables. So N keeps the set of tuple variables. For each table in the FROM clause, a tuple variable is automatically created: if not given explicitly, the variable will have the name of the relation.

A directed edge $(x, y) \in E$, represents a foreign key constraint between base table x and base table y, enabled by the join predicates inside the materialized view's query. Base table x is the base table with the foreign key that references the primary key of base table y. For instance, the FK-Graph describing the materialized view of the Example 1.3 consists of three nodes (ss, i, dd) and two directed edges. These two edges connect the "ss" node to the nodes representing the base tables "i" and "dd".

3 BASE TABLES FORMULATIONS

In this section, we describe the different states from which a base table passes during the period between two refreshes, and we deconstruct each of those states into "primary" sets of tuples. In this work we studied materialized views that have relations as base tables and don't recursively use other materialized views as base tables. However, in order to verify the correctness of the equations described in the rest of this work, it is necessary to verify that the same states also apply to base tables which are themselves materialized views, with the constraint that they are composed only by joins between two or more relations without "group by" and aggregations.

Set	Description
$Unc(R)$	The set of tuples inside a relation R, which is used as a base table in a materialized view V, that weren't affected by insertions or deletions since the last refresh of the view.
$Del(R)$	The set of tuples inside a relation R, which is used as a base table in a materialized view V, that were deleted since the last refresh of the view.
$Ins(R)$	The set of tuples inside a relation R, which is used as a base table in a materialized view V, that were inserted since the last refresh of the view.

Table 2. Relation's disjoint sets.

3.1 Relations as Base Tables

We assume that the changes that can occur in a relation are inserts, deletes and updates. We consider *"updates"* as two different operations: an insertion and a deletion. Tuples were inserted and then deleted during the period between two refreshes are easily detectable, and ignored during the IVM process. Keeping that in mind, we can separate a relation R, that is used as base table into the materialized view V, into three disjoint sets of tuples: the unchanged ($Unc(R)$), the inserted ($Ins(R)$) and the deleted ($Del(R)$). The descriptions of all three sets expressed in Table 2.

Using the three sets defined in Table 2, we can now express the two states that the base tables can be in: the *pre-state* and the *post-state*. These two states correspond to the instances of a base table right after a refresh of the materialized view it belongs to and just before the next one. In contrast to the previously defined sets, these states refer to base table instances and they include derivation counts. These derivation counts are always equal to one, because they belong to the relations where the tuples were born. Next we express the definitions of these two states using the disjoint sets defined in Table 2. We use the '+' exponent to explicitly declare sets with counts equals to (+1):

Definition 3.1. The *pre-state* of a base table of a materialized view V corresponds to the set of tuples that existed in the base table immediately after the last refresh of V. In the event that a relation R is used as a base table, this set of tuples consists of the *unchanged* and the *deleted* part of R with regard to V.

$$Pre(R) = Unc^{+}(R) \uplus Del^{+}(R).$$

Definition 3.2. The *post-state* of a base table of a materialized view V corresponds to the set of "visible" tuples that the base table currently has. In the event that a relation R is used as a base table, this set of tuples consists of the *unchanged* and the *inserted* part of R with regard to V.

$$Post(R) = Unc^{+}(R) \uplus Ins^{+}(R).$$

Next we are going to define the summary delta for relations used as base tables. We use the minus exponent '-' to explicitly declare sets with counts equal to -1:

Definition 3.3. The *summary delta* of a base table consists of the newly inserted and newly deleted tuples into/from the base table. In the event that a relation R is used as a base table:

$$\mathfrak{D}(R) = Del^{-}(R) \uplus Ins^{+}(R).$$

We can observe that, if we apply the set of changes defined in the $\mathfrak{D}(R)$ to the $Pre(R)$, we get the $Post(R) = Pre(R) \uplus \mathfrak{D}(R)$.

3.2 Materialized Views as Base Tables

Previously, we defined the states and the deltas of base tables provided that they are relations. Now, we are going to prove that the same states are followed by base tables that are materialized views, provided that they are defined as inner joins of two or more relations without "group by" and aggregations.

For a materialized view V of this type we know that:

(1) Each newly inserted tuple will take a new identifier that didn't exist in the pre-state of V: When two or more tuples join, the id of the resulting tuple is the set of the RowIds of the joined tuples. Each newly inserted tuple into the relations of the view takes a new identifier, so the newly inserted tuple in the materialized view will also have a new identifier.
(2) The tuples that are present in the materialized view V immediately after the last refresh have counts equal to one: When two or more tuples join, the count of the resulting tuple is a product of the counts of the joined tuples. By definition relations only store tuples with counts equal to one, so that is also the case for the tuples of materialized views without aggregations.

From these two observations, we conclude that each tuple inside the pre-state of V has count equal to one and that there can not exist a newly inserted tuple with a preexisting identifier. This means that each tuple of the pre-state of V can only be deleted or remain as is. So, we conclude that all three Definitions expressed in the previous section are valid for this type of base tables, too.

4 THE CURRENT STATE OF IVM

In this section, we are going to define the computations needed in order to produce the *summary delta* of an materialized view described only by products of two or more relations. For this purpose, we use the *summary delta* computations as these were defined by Gupta et al. [12]. We are going to redefine the proof for these computations using the relation's primary sets. With the term "primary sets" we refer to the *inserted*, *deleted* and the *unchanged* sets of tuples as these were defined in section 3. The methodology we follow here will help us understand the methodology followed in the rest of this work.

4.1 The Case of Cartesian Product

In this section, we define the *summary delta* computations for materialized views which are described by a Cartesian product of two relations. Consider a materialized view V joining tables R and S defined as $V = R \times S$. Suppose that we refresh the materialized view V and after that we insert and delete some tuples into both base tables. The following Lemma describes the computations needed in order to produce the $\mathfrak{D}(V)$ expressed only by the relation's primary sets:

Lemma 4.1. *The set of tuples that we should insert into the materialized view* $V = R \times S$ *is:*

$$\mathfrak{D}^+(V) = (Ins^+(R) \otimes Ins^+(S)) \uplus (Ins^+(R) \otimes Unc^+(S)) \uplus (Unc^+(R) \otimes Ins^+(S))$$

while, the set of tuples that we should delete from the materialized view is:

$$\mathfrak{D}^-(V) = (Del^-(R) \otimes Del^+(S)) \uplus (Del^-(R) \otimes Unc^+(S)) \uplus (Unc^+(R) \otimes Del^-(S))$$

Finally, the set of changes $\mathfrak{D}(V)$ *is defined as* $\mathfrak{D}(V) = \mathfrak{D}^+(V) \uplus \mathfrak{D}^-(V)$.

Proof. The contents of our materialized view after the last changes are $V = Post(R) \otimes Post(S)$. Using that the post-state of a base table B defined as $Post(B) = Pre(B) \uplus \mathfrak{D}(B)$ and distributing products over unions we 've got:

$$V = (Pre(R) \otimes Pre(S)) \uplus (Pre(R) \otimes \mathfrak{D}(S)) \uplus (\mathfrak{D}(R) \otimes Pre(S)) \uplus (\mathfrak{D}(R) \otimes \mathfrak{D}(S))$$

The term $Pre(R) \otimes Pre(S)$ define the $Pre(V)$, so the $\mathfrak{D}(V) = (Pre(R) \otimes \mathfrak{D}(S)) \uplus (\mathfrak{D}(R) \otimes Pre(S)) \uplus (\mathfrak{D}(R) \otimes \mathfrak{D}(S))$. Using that $Pre(B) = Unc^+(B) \uplus Del^+(B)$ and $\mathfrak{D}(B) = Del^-(B) \uplus Ins^+(B)$, and distributing products over unions we 've got:

$$\begin{aligned}
\mathfrak{D}(V) = {} & (Unc^+(R) \otimes Del^-(S)) \uplus (Unc^+(R) \otimes Ins^+(S)) \uplus \\
& \cancel{(Del^+(R) \otimes Del^-(S))} \uplus \cancel{(Del^+(R) \otimes Ins^+(S))} \uplus \\
& (Del^-(R) \otimes Unc^+(S)) \uplus (Del^-(R) \otimes Del^+(S)) \uplus \\
& (Ins^+(R) \otimes Unc^+(S)) \uplus \cancel{(Ins^+(R) \otimes Del^+(S))} \uplus \\
& \cancel{(Del^-(R) \otimes Del^-(S))} \uplus \cancel{(Del^-(R) \otimes Ins^+(S))} \uplus \\
& \cancel{(Ins^+(R) \otimes Del^-(S))} \uplus (Ins^+(R) \otimes Ins^+(S))
\end{aligned}$$

As it is shown, some pairs of terms of the above equation can be eliminated because their union produces the empty set. After eliminating all the unions that produce the empty set, we end up with the equations defined in this Lemma. □

The computations defined by Lemma 4.1 weren't optimal due to the number of union branches included to them. However, Lemma 4.1 deconstructs summary delta computations into products between relation's primary sets, which constitutes the smallest particles of the IVM computations. That means that every defined computation that calculates summary deltas should be capable to deconstructed into the computations defined in Lemma 4.1. In the following Theorem, we use these computations in order to proof the correctness of the state of the art computations defined by Gupta et all:

Theorem 4.2. *The set of changes $\mathfrak{D}(V)$ for the materialized view $V = R \times S$ can be computed by:*

$$\mathfrak{D}(V) = (\mathfrak{D}(R) \otimes Post(S)) \uplus (Pre(R) \otimes \mathfrak{D}(S))$$

Proof. Given that for a base table B: $Post(B) = Unc^+(B) \uplus Ins^+(B)$, $Pre(B) = Unc^+(B) \uplus Del^+(B)$ and $\mathfrak{D}(B) = Del^-(B) \uplus Ins^+(B)$ and distributing products over unions, the above rule can alternatively be written as:

$$\begin{aligned}
\mathfrak{D}(V) = {} & (Del^-(R) \otimes Unc^+(S)) \uplus \cancel{(Del^-(R) \otimes Ins^+(S))} \uplus \\
& (Ins^+(R) \otimes Unc^+(S)) \uplus (Ins^+(R) \otimes Unc^+(S)) \uplus \\
& (Unc^+(R) \otimes Del^-(S)) \uplus (Unc^+(R) \otimes Ins^+(S)) \uplus \\
& (Del^+(R) \otimes Del^-(S)) \uplus \cancel{(Del^+(R) \otimes Ins^+(s))}
\end{aligned}$$

However, $(Del^-(R) \otimes Ins^+(S)) \uplus (Del^+(R) \otimes Ins^+(S)) = \emptyset$. So, the computations defined in this Theorem are equal to the computations defined in Lemma 4.1. □

4.2 The Case of Multi-way Products

Consider now a materialized view V defined as $V = R_1 \times \cdots \times R_N$. We can treat this multi-way product as a product between the materialized view V' = $(R_1 \times \cdots \times R_{N-1})$ and relation R_N and

recursively compute the $\mathfrak{D}(V)$ as described by the following algorithm:

$$\mathfrak{D}(R_1 \times \cdots \times R_N) = \begin{cases} \mathfrak{D}(R_1), & \text{N = 1} \\ (\mathfrak{D}(R_1 \times \cdots \times R_{N-1}) \otimes Post(R_N)) \uplus & \\ \qquad (Pre(R_1 \times \cdots \times R_{N-1}) \otimes \mathfrak{D}(R_N)), & \text{N > 1} \end{cases}$$

If we unfold the above equation and distribute the products over the unions, we end up with exactly N union branches. The i-th leg can be defined as:

$$\mathfrak{D}_i(V) = Pre(R_1) \otimes \cdots \otimes Pre(R_{i-1}) \otimes \mathfrak{D}(R_i) \otimes Post(R_{i+1}) \otimes \cdots \otimes Post(R_n)$$

5 THE FK OPTIMIZATION

In this section, we are going to discuss the way that we incrementally update materialized views that use joins between columns following foreign key constraints. As we will see, we can simplify the computations needed to calculate the *summary deltas* for this kind of views. First, we will discuss the case of views defined as a simple foreign key-primary key join. After that, we will discuss how we can use the "FK-Optimization" in more complex join schemas (star, snowflake). Finally, we are going to define a general algorithm that handles materialized views with arbitrary FK-Graphs.

5.1 The Case of Single Join

Consider a materialized view V joining tables R and S, defined as $V = \sigma_{R.f=S.p}(R \times S)$. We assume that column $R.f$ is a foreign key that references column $S.p$, which is the primary key of relation S. Table R is the fact table, while table S is the dimension table of the materialized view V.

When the changes applied into the dimension table S of the materialized view V consisting only by insertions and/or deletions without updates, then we can use the Properties 1.1 and 1.2 to simplify the "deconstructed" computations defined by Lemma 4.1. More specifically, for the materialized view in our example we can use the following "deconstructed" computations:

Lemma 5.1. *Given a materialized view $V = \sigma_{R.f=S.p}(R \times S)$ where a) the set of changes in S are constituted only by insertions and/or deletions and b) the column R.f is a foreign key that references column S.p, then the set of changes $\mathfrak{D}(V)$ for the materialized view V, can be calculated from:*

$$\mathfrak{D}(V) = \sigma_{R.f=S.p}(Ins^+(R) \otimes Ins^+(S)) \uplus \sigma_{R.f=S.p}(Ins^+(R) \otimes Unc^+(S)) \uplus \\ \sigma_{R.f=S.p}(Del^-(R) \otimes Del^+(S)) \uplus \sigma_{R.f=S.p}(Del^-(R) \otimes Unc^+(S))$$

Proof. As we described, Lemma 4.1 defines the computations needed in order to calculate the summary deltas of the product between two base tables. In order to produce the summary delta computations for the materialized view V we can use the computations defined by Lemma 4.1 wrapping them with the join condition: $\mathfrak{D}(V) = \sigma_{R.f=S.p}(\mathfrak{D}(R \times S))$. Pushing the join condition in each union branch of the $\mathfrak{D}(R \times S)$, we can notice that some union branches can be eliminated. More specifically, the term $\sigma_{R.f=S.p}(Unc^+(R) \otimes Ins^+(S))$ can be eliminated due to Property 1.1 (there can not exist any unchanged tuples of the fact table that join with newly inserted tuples of the dimension table). Additionally, following the Property 1.2, we can eliminate the term $\sigma_{R.f=S.p}(Unc^+(R) \otimes Del^-(S))$. The result of this process is the summary delta computations defined in this Lemma. □

Lemma 5.1 deconstructs the *summary delta* computations into joins between relation's *primary* sets, for the case that the changes in the dimensions table S doesn't have updates. In the case of

updates, Lemma 5.1 can not be used since Properties 1.1 and 1.2 are not applicable. As we mentioned previously, we consider updates as two different operations: a deletion and an insertion. Let us look at the case where we update the contents of a tuple located in the dimension table. In this case, the updated tuple will be deleted and then re-inserted as a new one containing the updated values. It is easy to see how this process violates Property 1.1: The newly inserted tuple will be matched with tuples located in the unchanged part of the fact table. It also violates Property 1.2: The newly deleted tuple will be joined with tuples located in the unchanged part of the fact table. A useful observation is that updates into the fact table don't affect either of the two Properties. So, in order to be able to use Lemma 5.1 it suffices to check if there are any updates into the dimension table.

Despite the simplified computations defined in the previous Lemma, they still use more union branches than the "non-optimized" ones defined in Theorem 4.2. In order to reduce the number of union branches even more, we define the pre-delete state:

Definition 5.2. The *pre-delete* state of a relation R used as base table in the materialized view V is the whole set of tuples that the base table has. This set of tuples consists of the unchanged, the inserted and the deleted part of the relation R:

$$PreDel(R) = Unc^{+}(R) \uplus Del^{+}(R) \uplus Ins^{+}(R).$$

We can use the "pre-delete" state of the dimension table S, in order to remove the union operator from the *summary delta* scripts, as shown in the following theorem:

Theorem 5.3. *The set of changes $\mathfrak{D}(V)$ for the materialized view $V = \sigma_{R.f=S.p}(R \times S)$, where the set of changes in S constituted only by insertions and/or deletions and the column $R.f$ is a foreign key that references the column $S.p$, can be calculated from:*

$$\mathfrak{D}(V) = \sigma_{R.f=S.p}(\mathfrak{D}(R) \otimes PreDel(S)),$$

If $\Pi_{S.p}(Del(S)) \cap \Pi_{S.p}(Ins(S)) = \emptyset$.

Proof. Using $PreDel(S) = Unc^{+}(S) \uplus Ins^{+}(S) \uplus Del^{+}(S)$ and $\mathfrak{D}(R) = Del^{-}(R) \uplus Ins^{+}(R)$ and distributing products over unions, the computations defined in this Theorem can be deconstructed into:

$$\begin{aligned}\mathfrak{D}(V) = {} & \sigma_{R.f=S.p}(Del^{-}(R) \otimes Unc^{+}(S)) \uplus \\ & \sigma_{R.f=S.p}(Del^{-}(R) \otimes Ins^{+}(S)) \uplus \\ & \sigma_{R.f=S.p}(Del^{-}(R) \otimes Del^{+}(S)) \uplus \\ & \sigma_{R.f=S.p}(Ins^{+}(R) \otimes Unc^{+}(S)) \uplus \\ & \sigma_{R.f=S.p}(Ins^{+}(R) \otimes Ins^{+}(S)) \uplus \\ & \sigma_{R.f=S.p}(Ins^{+}(R) \otimes Del^{+}(S))\end{aligned}$$

We observe that the above "deconstructed" computation differs from the one defined in Lemma 5.1. More specifically, the "pre-delete" solution is going to join the deleted part of the fact table with the inserted part of the dimension table ($\sigma_{R.f=S.p}(Del^{-}(R) \otimes Ins^{+}(S))$), and vice versa ($\sigma_{R.f=S.p}(Ins^{+}(R) \otimes Del^{+}(S))$). If there is even one match between these joins, then the "pre-delete" solution will produce wrong results.

In order to avoid unwanted results, we use the computations defined in this Theorem only when there aren't any pairs of tuples (one deleted and one inserted) with the same primary key inside the dimension table ($\Pi_{S.p}(Del(S)) \cap \Pi_{S.p}(Ins(S)) = \emptyset$). Having in mind this constraint, let's enumerate all possible cases when the terms ($\sigma_{R.f=S.p}(Del^{-}(R) \otimes Ins^{+}(S))$, and ($\sigma_{R.f=S.p}(Ins^{+}(R) \otimes Del^{+}(S))$) produce at least one tuple, and prove that something like that can't happen:

1st Case: We insert a tuple into the fact table and then we delete its referenced tuple from the dimension table. Because of the foreign key constraint, the newly inserted tuple in the fact table should be deleted before the deletion of its referenced tuple from the dimension table. This inserted and then deleted tuple is going to be ignored during the delta computations of the fact table. So, this case produces no results.

2nd Case: We delete a tuple from the fact table and then we insert a tuple into the dimension table referenced by the deleted tuple. If such a tuple existed in the fact table then its reference tuple in the dimension table must have existed before that. In order to reinsert this tuple into the dimension table we should first delete the already existing tuple. If something like this happens, the set $\Pi_{S.p}(Del(S)) \cap \Pi_{S.p}(Ins(S)) \neq \emptyset$. So, this case can't happen.

3rd Case: We insert a tuple into the dimension table and then we delete a tuple from the fact table that references the previous one. To be able to delete the tuple from the fact table, we must have first inserted it. This tuple is going to be ignored from the deltas because it is inserted and then deleted in the same delta. So, this case produces no results.

4th Case: We delete a tuple from the dimension table and then we insert a tuple into the fact table referencing the previous tuple. In order to insert this tuple into the fact table, we must first reinsert the referenced tuple into the dimension table. If something like this happens, the set $\Pi_{S.p}(Del(S)) \cap \Pi_{S.p}(Ins(S)) \neq \emptyset$. So, this case can't happen. □

As we described, in order to use the delta computations defined in Theorem 5.3, we should make sure that a) the condition $\Pi_{S.p}(Del(S)) \cap \Pi_{S.p}(Ins(S))$ produces zero results and b) the changes applied into the dimension table S don't have updates. If these two conditions are not satisfied then we should use the "unoptimized" delta computations defined in Theorem 4.2. However, the condition $\Pi_{S.p}(Del(S)) \cap \Pi_{S.p}(Ins(S))$ includes the one performed when looking for updates into dimension tables. In our model, an update is composed of two tuples - one marked as deleted and the other as inserted - that share the same primary key and are applied via the same transaction. The described condition does exactly the same without the constraint about the transaction id. From now on we will refer to this condition as the "FK-Guard".

5.2 Multiple Joins

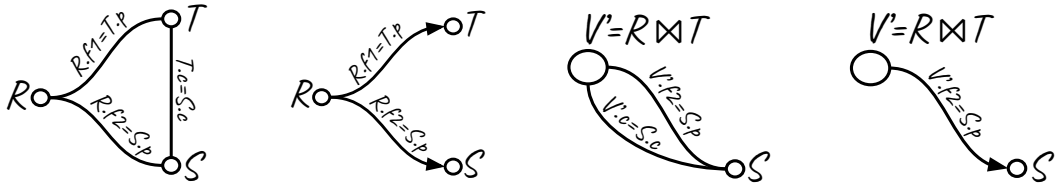
In this section, we are going to discuss how we can take advantage of the foreign key constraints used by materialized views, in more complex join schemas. First, we define a special type of graph, the rooted directed graph, and after that we describe the summary delta computations for materialized views that can be represented as rooted directed FK-Graphs. Finally, we define a general algorithm that handles materialized views represented by arbitrary FK-Graphs.

5.2.1 Rooted directed FK-Graph. A rooted directed graph is a special kind of graph constructed in order to describe snowflake, star or even cycle join schemas. The following definition describes such kind of graphs:

Definition 5.4. A *rooted directed* graph $G = (N, E, R)$ is a directed graph $G = (N, E)$ that has at least one node R such that there is a directed path from R to any node other than R.

As we have stated, rooted directed graphs can describe queries that follow star, snowflake join schemas. As a result, the delta computations for such graphs cover a big set of materialized views used by Data Warehouses. Before we describe the delta computations for such materialized views, let's define the following Property that will help us in this direction:

Property 5.5. *The table created by the inner join of two or more tables keeps the foreign key constraints of its joined tables.*



(a) Initial Join Graph. **(b)** Initial Fk-Graph. **(c)** Rewritten join graph. **(d)** Rewritten FK-Graph.

Fig. 2. Rewritten steps of a "Rooted directed" MV.

According to Property 5.5, we can use the optimized delta computations defined by Theorem 5.3, for joins between a materialized view V and a relation R provided that: a) the materialized view defined as inner joins of two or more relations without "group by" and aggregations b) the column from the relation R that participates in the join is also the primary key of R and c) the column from the materialized view V that participates in the join was used as a foreign key from one of the base tables of V pointing to primary key of R. We illustrate the importance of Property 5.5 using the following example:

Example 5.6. Consider a materialized view V, the join graph which is displayed in Figure 2a. As shown by its FK-Graph in Figure 2b, this materialized view was described by two joins following the foreign key - primary key relationship and a "simple" one. In our scheme, we assume that the FK-Guard is followed by both dimension tables (S, T).

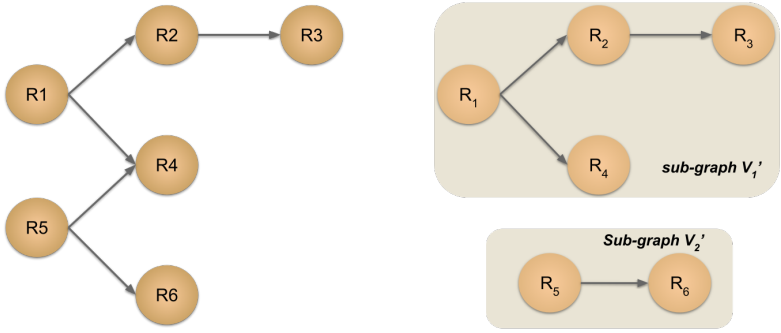
In order to calculate the summary deltas of the materialized view V, we can follow a similar approach as the one used to describe the summary deltas for materialized views defined by multi-way products (see section 4.2). More specifically, we can rewrite V as $V = \sigma_C(V' \times R')$, where V' is a materialized view which is described by the join defined between two of the three V's base tables, R' is a reference to the remaining base table and C describes the remaining join conditions. In order to compute the summary delta of V we should first calculate the summary delta of V'. If V' can be described by a join that follows the foreign key - primary key relationship then we can use the optimized scripts defined by Theorem 5.3 in order to calculate its summary deltas. As a result, the join between the tables R, T or R, S is a good selection for V'. Let's assume that $V' = R \bowtie S$. Figure 2c shows the join graph of V as this described after rewriting it using V'.

The question, now, is if we can use the "FK-Optimized" scripts to calculate the deltas of the join between the materialized view V' and relation T, or if we should use the "unoptimized" ones. According to Property 5.5, V' has inherited R's foreign key constraints. The resulting FK-Graph is shown in Figure 2d. The delta computations for our example can be expressed as:

$$\begin{aligned}\mathfrak{D}(V) &= \sigma_C(\mathfrak{D}(V') \otimes PreDel(S)) \\ &= \sigma_C(\sigma_{R.f1=T.p}(\mathfrak{D}(R) \otimes PreDel(T)) \otimes PreDel(S)) \qquad \square\end{aligned}$$

In the above example, we can observe that the delta of the materialized view V can be calculated using a query without any *union* operator. On the other hand, if we used the "unoptimized" scripts we would need a *union* operator with three branches to calculate the deltas. We can also notice that the computations follow a pattern: we join the pre-delete state of the dimension tables with the delta of the fact table. The computations needed, in order to produce the deltas of materialized views that can be represented by a rooted directed FK-Graph, are described in the following theorem:

Theorem 5.7. *Consider a materialized view V described by the rooted directed Fk-Graph $G = (N, E, R)$. Also consider the set S described by all base tables of V except R ($S = N - R$). In order to*



(a) FK-Graph G. (b) Set S'.

Fig. 3. Arbitrary FK-Graph.

compute $\mathfrak{D}(V)$ *we can use the following computations:*

$$\mathfrak{D}(V) = \mathfrak{D}(R) \otimes PreDel(S_1) \otimes \cdots \otimes PreDel(S_{|V|-1}) \tag{1}$$

If $(\Pi_{a_i}(Del(S_i)) \cap \Pi_{a_i}(Ins(S_i))) = \emptyset, \forall i \in [1, |V| - 1]$, *where* a_i *is the primary key of relation* S_i.

Proof. Let's assume the order of G's nodes $\{v_1, v_2, \ldots, v_{|V|}\}$ where v_1 refers to R and each $v_i \neq v_1 \in S$. We assume that in this order, v_i has at least one incoming edge from a node in the set $\{v_1, v_2 \ldots, v_{i-1}\}$, the nodes of which form a rooted directed sub-graph. An order with these properties can occur from the sequence that the Breadth-First Search algorithm visits the nodes of G. For such an order we can define the following recursive formula:

$$\mathfrak{D}(v_1 \bowtie .. \bowtie v_N) = \begin{cases} \mathfrak{D}(u_1), & \text{N = 1} \\ \mathfrak{D}(u_1 \bowtie .. \bowtie u_{N-1}) \otimes PreDel(u_N) & \text{N > 1} \end{cases}$$

We know that in each depth of recursion, we can use the "FK-Optimized" computations because it is certain that there is an outgoing edge from nodes $\{v_1, v_2 \ldots, v_{i-1}\}$ to node v_i. Also, we know that $\Pi_{a_i}(Del(v_i)) \cap \Pi_{\hat{a_i}}(Ins(v_i))) = \emptyset$. Unfolding the equations we get:

$$\mathfrak{D}(V) = \mathfrak{D}(u_1) \otimes PreDel(u_2) \otimes \cdots \otimes PreDel(u_N)$$

Taking advantage of the commutative property of joins, we only need to know which node can reach all other nodes in order to produce the summary delta computations. □

5.2.2 Arbitrary Fk-Graph. Consider now the materialized view represented by the FK-Graph in Figure 3a. Consider also the scenario where we want to refresh this materialized view, and all dimension tables comply with the FK-Guard. We consider as dimension tables the base tables whose node representation inside the materialized view's FK-Graph has at least one incoming edge. Unfortunately, for this materialized view, we can't use the delta computations described by Theorem 5.7 because this graph isn't rooted directed (there isn't a node from which we can reach all other nodes).

Algorithm 1 describes how we can produce the summary delta computations for all kinds of FK-Graphs. The intuition behind this algorithm is that we can detect rooted directed sub-graphs and use the "FK-Optimized" delta computations defined by Theorem 5.7 in order to compute their summary deltas. From these sub-graphs we can create a new set, each object of which represents a materialized view consisting of the joins and relations defined inside the sub-graphs. This new set contains objects without any connections (there isn't any join between the materialized views that

Algorithm 1: Summary deltas for arbitrary FK-Graphs.

input : The FK-Graph G = (N, E) of the materialized view V.
input : The set C of join and filter conditions of V.
output: $\mathfrak{D}(V)$ computations.

1 Create an empty set S'
2 **while** *N is not empty* **do**
3 $BC := \emptyset$ // Biggest Rooted Directed Component
4 **for** *each node v in V* **do**
5 CC := Rooted Directed component using v as root.
6 **if** $BC = \emptyset$ ***OR*** $|CC.N| > |BC.N|$ **then**
7 BC := CC
8 **end**
9 **end**
10 Remove from G all edges located inside BC.
11 Remove from G all nodes located inside BC.
12 Remove from G all edges pointing to a removed node.
13 Remove from C all joins described by an edge in BC.
/* Use the "FK-Optimized" computations. */
14 v := Vertex annotated with $\mathfrak{D}^{FK}(BC)$, Pre(BC), Post(BC).
15 $S' := S' + v$
16 **end**
/* Use the "unoptimized" computations. */
17 **return** $\sigma_C(\mathfrak{D}^{UN}(S'))$

follows the foreign key - primary key relationship). For this kind of sets, we can compute the final delta using the "unoptimized" delta computations described in Section 4.2.

To illustrate Algorithm 1, let us use as an example the FK-Graph of Figure 3a. Algorithm 1 takes as input the FK-Graph G of a materialized view V and all of its join conditions C. In the first run of the *while* loop (line 2), the algorithm will choose the rooted directed sub-graph V_1' defined by nodes $\{R_1,R_2,R_3,R_4\}$ (lines 3-9). Next (lines 10-11), the algorithm will remove the nodes and the edges of the chosen sub-graph from the initial FK-Graph G. Also, it will remove the edge directed from node R_5 to node R_4 (line 12) since this edge will point to a removed node. Finally (line 13), it will remove from the initial set of join/filter conditions C the joins used inside the rooted directed sub-graph V_1'. At the end of the first run of the while loop (lines 14 - 15), the algorithm will add a new node V_1' in set S' annotated with $Pre(V_1')$, $Post(V_1')$ and the "FK-Optimized" summary delta computations $\mathfrak{D}^{FK}(V_1')$. In the second run of the *while* loop, the algorithm will do the same for the sub-graph V_2' = $\{R_5,R_6\}$. Finally, Algorithm 1 will calculate the summary delta computations for the product of V_1' and V_2' in S' ($\mathfrak{D}^{UN}(S')$) and it will apply the remaining join/filter conditions C to the result:

$$\mathfrak{D}(V) = \sigma_c((\mathfrak{D}(V_1') \otimes Post(V_2')) \uplus (Pre(V_1') \otimes \mathfrak{D}(V_2')))$$

where $\mathfrak{D}(V_1') = \mathfrak{D}(R_1) \otimes PreDel(R_2) \otimes PreDel(R_3) \otimes PreDel(R_4)$ and $\mathfrak{D}(V_2') = \mathfrak{D}(R_5) \otimes PreDel(R_6)$. Note that we can still push down the filter conditions inside C since the operators $\uplus$ and $\otimes$ have the same properties as their counterparts.

6 IMPLEMENTATION DETAILS

So far, we have seen how we produce the "FK-Optimized" delta computations for materialized views described by any form of FK-Graphs. In order to use these computations, the FK-Guard must be satisfied by all the dimension tables. In this section, we are going to discuss two implementation approaches which can support this constraint: the static and the dynamic one. Both approaches can be applied in systems that follow the lazy approach to refresh their materialized views [7, 14, 31]. This means that they refresh the materialized views periodically when the system is under low load.

In the static approach, the delta scripts are created during the genesis of the materialized view. These delta scripts are going to be executed each time the materialized view is refreshed. These delta scripts contain a conditional statement, which checks during the refresh process if the FK-Guard is followed by all dimension tables. If such a case does not exist, then the conditional statement will chose to use the "unoptimized" delta computations, described in Section 4.2. Otherwise, we execute the computations produced by Algorithm 1.

In the dynamic approach, the delta scripts are produced from scratch during each refresh process. More specifically, we first create the FK-Graph of the materialized view, then we detect the dimension tables that don't follow the FK-Guard, and, finally, we remove all incoming edges from the corresponding nodes in the FK-Graph. After that, we add the modified FK-Graph as input in Algorithm 1 and dynamically create the delta computations. This approach requires the recreation of delta scripts in each refresh of the materialized view.

Each of the two approaches has its own advantages and disadvantages. The dynamic approach will use the "FK-Optimized" delta scripts more often, but it will add an extra overhead in each refresh because the creation of delta scripts is a time consuming process. On the other hand, the static approach doesn't add any overhead during the refresh time, but it is affected more when the FK-Guard isn't followed even by a single dimension table. In our experimental evaluation, we chose to implement the static approach because it better fits Redshift's needs.

7 EVALUATION

The aim of our evaluation is to examine the performance of the "FK-optimized" delta computations resulting from Algorithm 1, when compared with the "unoptimized" ones, as well as the DBToaster's variant for batch updates [22]. All experiments were performed in Redshift serverless cloud-based warehousing system. More specifically, we separated our evaluation section into three sets of experiments. The first set focuses on the comparison with the "unoptimized" delta computations. The second set of experiments shows how the "FK-Optimization for delta computations affects the execution time of the whole refresh process, including the time needed to apply the changes into the materialized view. Finally, the third set of experiments compares the performance of the "FK-optimized" delta computations and a variation of it with the "unoptimized" ones as well as with the DBToaster's approach.

For our evaluation, we used two different clusters: a small one and a big one. Both clusters were created using the AWS Cloud Service. The small cluster consisted of 4 nodes of type ds2.8xl (32 vCPUs, 244 GiB of memory and a 2.56TB SSD disk) while the big cluster consisted of 9 nodes of type i3en.12xl (48 vCPUs, 384 GiB of memory and 4 x 7.5TB NVMe SSD disks). The small cluster was loaded with 3 TB of TPC-DS data and the big one with 30 TB of TPC-DS and 10 TB of TPC-H data. We used the small cluster for the first set of experiments and the big one for the second and third.

Redshift supports two types of indices: *dist-keys* and *sort-keys*. *Dist-keys* define how data are distributed at the computation nodes. They increase parallelism and minimize data reshuffling over

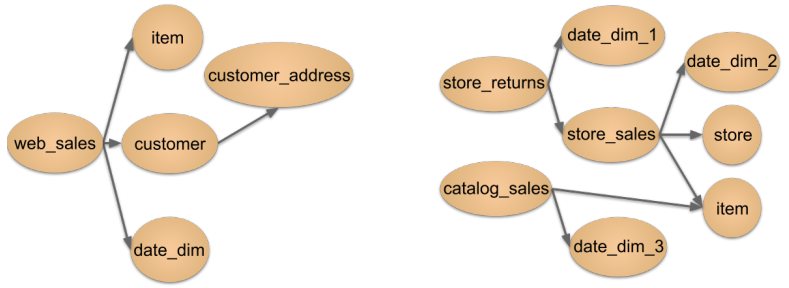
the network when the data distribution required by the query matches the physical distribution of the underlying tables. *Sort-keys* define which columns are used for sorting, thus improving the efficiency of sort-based operations like merge join. During the experimentation all base tables had the optimal sort and distribution keys for both TPC-H and TPC-DS benchmarks, the majority of which are built around foreign keys and primary keys. To reduce the number of blocks that need to be scanned, Redshift evaluates query predicates over zone maps i.e., small hash tables that contain the min/max values per block, and leverages late materialization. Additionally, it uses Bloom Filters to improve the performance of joins of large tables. Rows that do not match the join relation are efficiently filtered out at the source, thus reducing the amount of data transferred over the network. Finally, for newly inserted tuples, Redshift uses a mechanism similar to the "Incremental Data" as this is described in [30]: new tuples are not added to the *sort-keys* but are rather placed at the end of the relation. For a detailed description of Redshift's optimizations the reader is referred to [**?**]. All these built-in optimizations were kept active during all of the experiments. As a consequence, even in cases such as the existence of empty UNION ALL branches that could result in an unfair comparison to the proposed optimization only a minimum overhead is added. It is worth mentioning that the presence of main memory row-level indices would accelerate the check for empty results and would improve the results for the other tested techniques but, for the time being, no cloud warehousing engine supports them.

In order to evaluate the execution time of the delta computations, we first had to choose the update policy of the base tables used by the materialized views. We chose to evaluate each materialized view separately in order to have better control over the changes applied to the base tables. More specifically, we used the following approach: First, we extracted a small part of tuples from each relation (node) that didn't have any outgoing edges in the FK-Graph representation. We marked half of these tuples as "inserts" and the other half as "deletes". We then moved to the immediate neighbors of those nodes and we extracted all tuples referencing the initial tuples, marking them as "inserts" or as "deletes" according to the status of the referenced tuple. We continued with the same process, visiting the neighbors of neighbors, and extracting and marking tuples referencing the already extracted ones - until we had visited all available base tables. After the extraction of tuples was finished, we created new copies of the base tables without the set of tuples marked as "inserts" and we generated our materialized view. Then, we inserted the tuples marked as "inserts" and deleted the tuples marked as "deletes". Finally, we refreshed our materialized view and measured the execution time [1].

7.1 Delta Computations

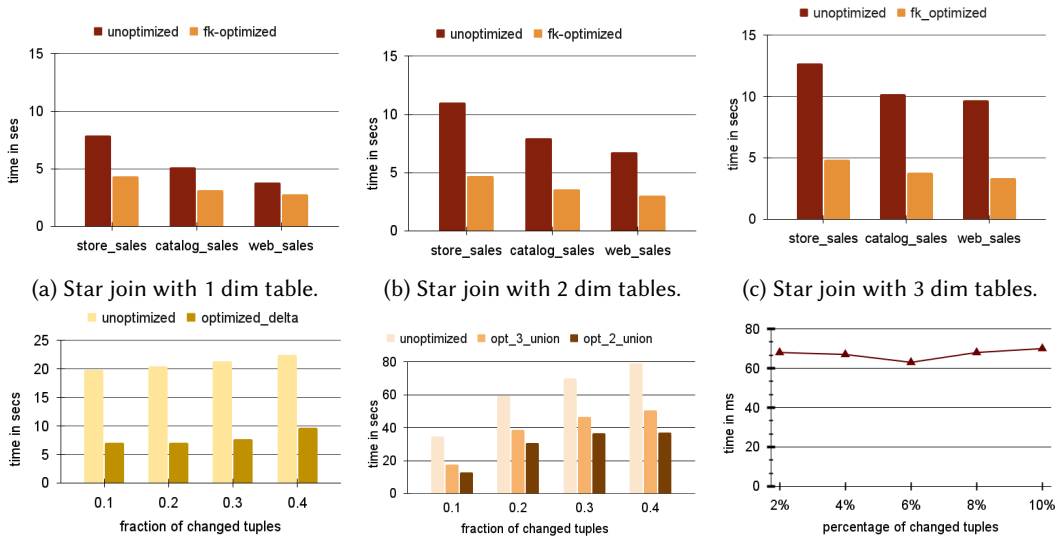
The aim of this set of experiments is to examine the performance of the "FK-optimized" delta computations, resulting from Algorithm 1, compared with the "unoptimized" ones. In order to evaluate our delta computations we have created eleven materialized views without filters or aggregations. Nine of them follow the star join schema while two of them follow the snowflake and the arbitrary join schema, respectively. The nine materialized views that follow the star join schema derived from three initial materialized views which also followed the star join schema and were selected from a set of 173 materialized views extracted from TPC-DS. These three materialized views used the same three base tables (*date _dim*, *item*, *customer_address*) as dimension tables, and they all had exactly three joins that followed the foreign key - primary key relationship. The only difference was in the fact table (*store_sales*, *web_sales* and *catalog_sales*). Two additional candidate materialized views were generated from each of the initially chosen materialized views

[1]You can find all materialized views used during the evaluation as well as the update policy followed in https://github.com/aws-samples/amazon-redshift-ivm-fk-optimization-queries



(a) Snowflake FK-Graph. (b) Arbitrary FK-Graph.

Fig. 4. FK-Graphs for Complex Join Schemata.



(a) Star join with 1 dim table. (b) Star join with 2 dim tables. (c) Star join with 3 dim tables.

(d) Snowflake join schema. (e) Arbitrary join schema. (f) FK Guard.

Fig. 5. Execution times for delta computations.

(six additional materialized views in total), containing the subsets <$fact\_table$, $customer\_address$> and <$fact\_table$, $customer\_address$, $item$>. The FK-Graph for the materialized view with snowflake schema is shown in Figure 4a, while Figure 4b shows the FK-Graph for the materialized view following the arbitrary join schema.

Figures 5a, 5b and 5c show the execution times of the delta computations for nine different materialized views which followed the star join schema. For these three diagrams we use the same bar chart colors, since all three have the same range in the time axes. To simulate changes occurring between two refreshes, .05 percent of the total tuples in each dimension table were marked as inserted and another .05 percent as deleted. Each one of the three charts shows how effective our optimization was for star joins having one, two and three dimension tables, respectively. We can observe that as the number of dimension tables increased, so did the effectiveness of the "FK-Optimization". This was expected: as the number of dimension tables increases, so does the

number of UNION ALL branches in the "unoptimized" delta computations. At the same time, the "FK-optimized" deltas don't have any union branches and are only affected by the increase of joins.

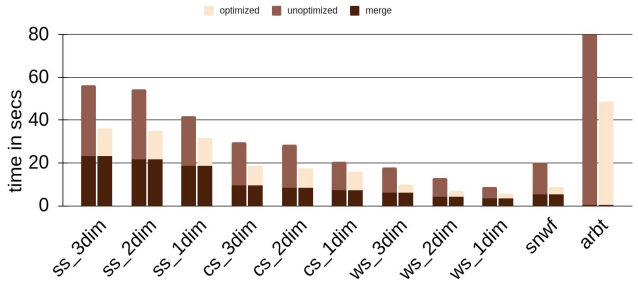
Figures 5d and 5e show the execution times of the delta computations for materialized views following the snowflake and arbitrary join schemas, respectively. In the horizontal axis, we see the percentage of tuples that were chosen from base tables without outgoing edges and were marked as inserted or deleted. More specifically, Figure 5d shows that the "FK-Optimized" delta scripts were 3 times faster in comparison with the "non-optimized" ones. This happens because snowflake schemas can be handled by the delta scripts without any union branches. On the other hand, for this specific materialized view the "non-optimized" delta scripts use a union operator with 5 branches. In this experiment, we also notice a small increase in execution times as the number of insertions and deletions increases. This was expected for our specific experimental setup because, as we increase the number of changes, we also increase the size of the results of the delta computations. The same results were noticed for the materialized view that follows an arbitrary join schema. More specifically, from the FK-Graph of this materialized view (Figure 4b), we distinguish two rooted directed sub-graphs, that have as roots the nodes *store_returns* and *catalog_sales*, respectively. The "FK-optimized" delta scripts of this FK-Graph contain a UNION ALL operator with two branches. In order to better study the behavior of arbitrary FK-Graphs, we produce another FK-Graph by removing the foreign key from the store_returns table pointing to the store_sales table. We use this FK-Graph to also produce a delta script that uses a union operator with three legs. From the execution times of this delta scripts (Figure 5e), we can notice that the "FK-Optimized" delta scripts that had two union legs were more than two times faster compared with the non-optimized ones. We can, also, notice an small increase of the execution times when we use the delta scripts that have a union operator with three legs.

Figure 5f shows the execution times of the computations that check if there are pairs of tuples (one inserted and one deleted), with the same primary key a into a relation S ($\Pi_a(Del(S)) \cap \Pi_a(Ins(S)) = \emptyset$). As we mentioned, this check is needed in order to decide if we are going to execute the "FK-optimized" delta scripts or the "unoptimized" ones. The horizontal axis shows the percentage of tuples that we chose to update from the customer table. The customer table was one of the biggest dimension tables in our dataset with 30 billion tuples. As we see, this check was executed in only a few milliseconds. This happened because the sizes of inserted and deleted parts is very small compared to the total size of the relation. Comparing these execution times with the ones needed for delta computations, we can safety assume that this check doesn't add any overhead.

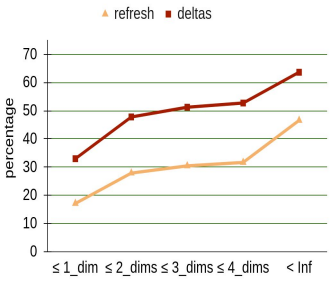
7.2 Refresh Process

We can separate the execution time of the refresh process into two disjoint execution times: the time needed for the delta calculation and the time needed to merge the calculated deltas with the materialized view. In the previous set of experiments we studied how the execution time of the delta computations is affected by the "FK-Optimization". In this subsection, we are going to study the way that the "FK-optimization" of the delta computations affects the total refresh time of a materialized view.

Figure 6a shows the execution times required for the refresh of each materialized view used in the previous subsection. In the horizontal axis, we notice the materialized views names. The first nine groups of bar charts correspond to the materialized views following the star join schema. The first two letters in each materialized view name indicate the name of the fact table that the materialized view was using, while the last four characters indicate the number of the dimension tables. The last two pairs of bar charts refer to the materialized views that follow the snowflake (snwf) and the arbitrary (arbt) join schemas. In order to not overshadow the other bar charts due to the difference in height, we prune the bar that shows the total refresh time of the "unoptimized"



(a) The execution times of the refresh of the eleven materialized views.



(b) % of improvement in execution times.

Fig. 6. Experiments related with the whole refresh process.

refresh process for the "arbt" materialized view. The total execution time of this process was 207 seconds.

We separate each bar into two disjoint execution times. We use the "merge" label to refer to the execution times needed for each materialized view to merge its deltas, and the "optimized" and "unoptimized" labels in order to refer to the execution times of the "FK-Optimized" and "unoptimized" delta computations, respectively. We didn't visualize the execution times for the condition that checks if the dimension tables follow the FK-Guard, because this execution was extremely fast in comparison with the rest. The execution times of the merge process is identical for both "FK-Optimized" and "unoptimized" refresh processes. This was expected as the summary deltas produced by both methods are identical. Lastly, we can observe that the merge process of the "arbt" materialized view was extremely fast in comparison with the other materialized views. This happens because the summary deltas produced by this materialized view was extremely small.

Finally, Figure 6b illustrates how much the "FK-Optimization" improved the execution times of the delta computations and of the refresh process in total, as the number of dimension tables increased. More specifically, the horizontal axis shows the set of materialized views chosen each time in order to measure the improvement. We begin with materialized views that have one dimension table and we end up with all materialized views. In this chart, we can see that for the complete set of materialized views, the "FK-Optimizations" improve the execution times of the whole refresh process by 47%, while they improve the execution times of delta computations by 64%.

7.3 Comparison With Other Approaches

In this section we are going to examine the performance of the "FK-optimized" delta computations and a variation of it compared with the "unoptimized" ones as well as with the DBToaster's approach. DBToaster uses a technique called recursive IVM. This method uses delta queries ("first-order deltas") to incrementally maintain the view of the input query, then materializes the delta queries also as views, maintains these views using delta queries of delta queries ("second-order deltas") and, finally, continues with alternating between materializing views and deriving higher-order delta queries for maintenance. There are several variants of the DBToaster's methodology [1, 17, 22]. For our experimental evaluation we implement using SQL the recursive IVM method constructed for batch updates as explained in [22]. To speed up the execution times we also built proper indices for every auxiliary materialized view created by the algorithm.

Inspired by DBToaster and the recently published work by Qichen Wang and Ke Yi [28], we created a new variation of the FK- Optimization. In this variation we traverse the FK-Graph in post

order creating materialized views from the sub-trees as we move upwards. For example, for the FK-Graph of Figure 4a, we first create the $customer \bowtie customer\_address$ Materialized View (A) and after that, the $web\_sales \bowtie A \bowtie date\_dim \bowtie item$ Materialized View (B) which is also the final one. Since materialized view A has as a primary key the primary key of customer, we can use the "FK-Optimized" scripts in order to maintain the final materialized view. It can be easily observed that this is true for all the created Materialized Views. We call this variation "Materialized" Fk-Optimization (M-OPT).

In order to evaluate the four different approaches we used the snowflake query from the previous experiment setup, which has no aggregations or filters, and a new materialized view extracted from "query 10" of "TPC-H". This newly extracted materialized view also has no aggregations nor filters, uses four base tables (*lineitem*, *orders*, *customer* and *nation*) and follows a chain join schema. We then added filters to those two materialized views and created two more. We chose to add filters to the "date_dim" base table of the snowflake query and to the "lineitem" and the "orders" base tables of the chain query, following "query 10" of "TPC-H". According to our update policy, as defined at the Introduction of Section 7, we should have started the extraction of inserted and deleted tuples from the "nation" table of TPC-H. Since this table only has 25 rows that never change, we chose to start from the "customer" table instead.

Figure 7 shows the results of our evaluation. As you can see, our approach and the "Materialized" variation outperform the other two approaches and especially DBToaster. DBToaster produces a large number of auxiliary views, the maintenance of which takes a lot of time. More specifically, DBToaster created 10 auxiliary materialized views for the "chain" query and 17 materialized views for the snowflake one. During the refresh of the materialized view 42 refreshes were executed for the snowflake query and 16 for the chain one. This happened because DBToaster handles the updates to each table separately and, as a result, produces a large number of maintenance queries which corresponds to an equally big number of storage accesses. For the queries with filters, where the size of some auxiliary views is big since their base tables don't have any filters, the DBToaster becomes even slower than the other approaches. Especially in the case of the snowflake query with filters, it is 2 orders of magnitude slower than our approach.

Comparing the "FK-Optimization" with its "Materialized" variation we can see that the "FK-Optimization" is almost always faster than the variation. For the chain query the "Materialized" variation created three materialized views, the maintenance of which leads always to slower refreshes. In the case of the snowflake query, where we created only two materialized views, the "Materialized" variation is faster when no filters are used. However, for the "snowflake" query with filters the FK-Optimization was faster, since the join of $\mathfrak{D}(web\_sales)$ with the filtered "date_dim" produced small delta intermediate results. The "Materialized" version of the IVM algorithm must first update the $customer \bowtie orders$ materialized view and then execute the final query. As a result, the total maintenance cost of the "Materialized" approach was bigger than the regular one.

8 RELATED WORK

Classical IVM: *"Classical IVM"* is a problem studied for more than three decades giving rise to numerous influential works [4, 5, 9, 12, 13, 19, 21, 24–26]. The *"Classical IVM"* methods calculate the set of changes by executing a single delta computation and they avoid materializing auxiliary views that occur in the delta computation query tree. We can categorize such IVM methods into two approaches, the Algebraic and the procedural one. The algebraic approach was one of the most popular approaches [4, 8–10, 24, 25]. It expresses all maintenance tasks using only the standard query operators, hence it can leverage existing query execution and optimization engines. However, in practice, efficient implementation sometimes requires giving up the conceptual simplicity of the algebraic approach and specifying aspects of the maintenance in a more procedural manner. Thus,

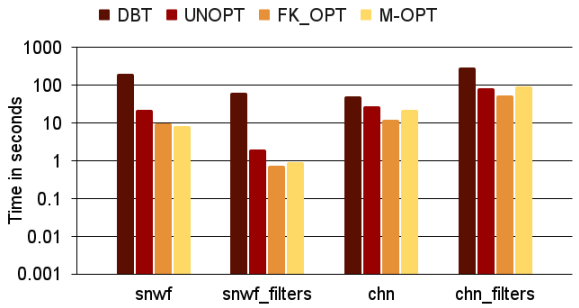


Fig. 7. Comparison with other approaches.

summary-deltas and variants [12, 13, 18, 23] have emerged as the preferred method of representing changes. One of the main advantages of this method is that the IVM scripts use delta computations that handle both insertions and deletions in one pass rather than separately computing the set of deletions and the set of insertions. For comprehensive surveys, the reader is referred to [6, 11].

Logical-level optimizations, such as the one presented here, are especially important for scalable, serverless warehouses, whose materialized view implementations (and, generally, query processing) do not have the benefit of classic, main memory-based indices that provide O(1) access to the list of ("pointers" to the) matching rows. Scalable, serverless warehouses have decoupled storage and compute servers, employing clusters when needed. Furthermore, they rely exclusively on cheap, "big block"-based cloud storage, thus delivering tremendously lower cost of ownership to customers. These architectural choices have led to query processing techniques that do not need and/or use conventional main memory indices. Furthermore, none of their IVM features immediate propagation of base table changes to the materialized views. Rather, refreshes involve relatively big batches of changes.

In that context, the presented optimization is very useful since it altogether avoids accesses, by reorganizing operations at the logical level, as opposed to relying on a main memory index to avoid accesses by physical level means.

Materialized View Maintenance: Another class of IVM methods materialize auxiliary views in order to speed up the delta computations [15–17, 22, 27, 28]. There are works using these methods for both the problem of batch updates [16, 22, 27] and the problem of per-tuple IVM [15, 17, 28]. For the problem of per-tuple IVM, where all auxiliary materialized views can be stored in main memory and the focus is on achieving low latency rather than high throughput, the “Materialized view Maintenance” approach prevailed in comparison with first order IVM. However, this is not always the case for batch updates in cloud-based warehousing systems as in such cases the cost of maintaining the auxiliary materialized views is bigger than the in-memory model adopted by the per-tuple IVM approaches. The problem of finding the best set of materialized views that reduces the total time cost of view maintenance by materializing and maintaining additional views is hard and relying on the set of updates which dynamically changes in each refresh [27]. So, materializing and maintaining part or each node of the query tree may not occur at an optimal maintenance cost as shown by our experimental evaluation. Even in the case that materializing auxiliary views is better, our technique can still accelerate their maintenance.

IVM and Integrity Constraints: Despite the numerous works related to IVM, only a few of them use the integrity constraints in order to speed up the materialized view’s refresh process. Quass et al. [26] was one of the first works that took advantage of the foreign key constraints in order to speed up the maintenance process. However, their work was developed in order to maintain

materialized views which handle data from multiple external sources. The idea of this work was to keep auxiliary data for each materialized view in order to minimize the external information needed to maintain them. One of the optimizations mentioned in this work was the simplification of maintenance expressions for joins that follow the foreign key - primary key relationship. However, it uses different maintenance expressions for updates, deletions and insertions, and the foreign key - primary key related optimization is applicable only to insertions.

Larson et al. [19] introduce an efficient incremental maintenance procedure for outer join views. Their approach was very close to the algebraic one in the sense that they use regular algebraic operators without defining new ones. As a consequence, the set of insertions and deletions are calculated using two different maintenance expressions. One of the proposed optimizations uses the foreign key constraints in order to eliminate inner joins from the delta expressions. Unfortunately, their optimizations are specific to outer join views and can't be used in the case of inner joins.

Finally, Qichen Wang and Ke Yi [28] developed an algorithm in order to incrementally maintain "foreign-key acyclic" queries, which are queries that can be described by rooted directed FK-Graphs. Their algorithm is based on the observation that each tuple in the fact table can produce at most one result tuple, so keeping and continuously updating some extra columns in each base table leads to an efficient incrementally updated multi-way join algorithm. We can consider their approach similar to the "Materialized" version of our approach described in Section 7.3. However, a) their algorithm is designed for the case of per-tuple IVM where all data and auxiliary indexes/materialized views can be stored in main memory and can be repeatedly updated without a big cost. This results in a complicated algorithm that can not be easily extended for the batch update approach. b) Their model is based on the streaming scenario and the per-tuple IVM, so there are no guarantees about the order that the updates come in. As a result, Properties 1.1 and 1.2 are not applicable so their algorithm has to execute extra computations which in our approach are eliminated. And, c) as shown by our experimental evaluation, materializing part of the FK-Graph doesn't always result in the best maintenance cost. The problem of finding the best set of materialized views that reduces the total time cost of view maintenance by materializing and maintaining additional views is hard and relies on the set of updates, which dynamically changes in each refresh [27]. Furthermore, even in the case that materializing auxiliary views is better, our technique can still accelerate their maintenance.

9 CONCLUSION

In this work, we have presented an optimization that speeds up the performance of IVM scripts for materialized views having inner joins which follow the foreign key - primary key constraint. The IVM computations defined in the bibliography and used widely by serverless cloud-based data warehousing systems suggest delta computations with a union operator having as many branches as the number of base tables of the materialized view. However, knowing that some of the joins of the materialized view follow the foreign key - primary key constraints, we can simplify the delta computations. More specifically, we saw that for materialized views that follow snowflake join schemas and, as a consequence, star join schemas, we can produce delta computations without any union operator. For materialized views with more complicated join schemas, we have described an algorithm that also simplifies the IVM scripts and produces delta computations with less union branches than the ones produced by the state-of-the-art algorithms. In the experimental section, we saw that the proposed optimization showed very encouraging results. The experiments show that for a set of materialized views that follow various join schemas the proposed optimization improved up to 2 times the total refresh process and up to 2.7 times the execution times of delta computations.

REFERENCES

[1] Supun Abeysinghe, Qiyang He, and Tiark Rompf. 2022. Efficient Incrementialization of Correlated Nested Aggregate Queries using Relative Partial Aggregate Indexes (RPAI). (2022).

[2] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. Join synopses for approximate query answering. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*. 275–286.

[3] Randall G Bello, Karl Dias, Alan Downing, James Feenan, Jim Finnerty, William D Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. 1998. Materialized views in Oracle. In *VLDB*, Vol. 98. 24–27.

[4] Jose A Blakeley, Per-Ake Larson, and Frank Wm Tompa. 1986. Efficiently updating materialized views. *ACM SIGMOD Record* 15, 2 (1986), 61–71.

[5] O Peter Buneman and Eric K Clemons. 1979. Efficiently monitoring relational databases. *ACM Transactions on Database Systems (TODS)* 4, 3 (1979), 368–382.

[6] Rada Chirkova and Jun Yang. 2011. Materialized views. *Foundations and Trends in Databases* 4, 4 (2011), 295–405.

[7] Latha S Colby, Akira Kawaguchi, Daniel F Lieuwen, Inderpal Singh Mumick, and Kenneth A Ross. 1997. Supporting multiple view maintenance policies. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. 405–416.

[8] Timothy Griffin and Bharat Kumar. 1998. Algebraic change propagation for semijoin and outerjoin queries. *ACM SIGMOD Record* 27, 3 (1998), 22–27.

[9] Timothy Griffin and Leonid Libkin. 1995. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. 328–339.

[10] Timothy Griffin, Leonid Libkin, and Howard Trickey. 1997. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering* 9, 3 (1997), 508–511.

[11] Ashish Gupta, Inderpal Singh Mumick, et al. 1995. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18.

[12] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166.

[13] Himanshu Gupta and Inderpal Singh Mumick. 2006. Incremental maintenance of aggregate and outerjoin expressions. *Information Systems* 31, 6 (2006), 435–464.

[14] Eric N Hanson. 1987. A performance analysis of view materialization strategies. *ACM SIGMOD Record* 16, 3 (1987), 440–453.

[15] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1259–1274.

[16] Yannis Katsis, Kian Win Ong, Yannis Papakonstantinou, and Kevin Keliang Zhao. 2015. Utilizing ids to accelerate incremental view maintenance. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1985–2000.

[17] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal* 23, 2 (2014), 253–278.

[18] Wilburt Labio, Jun Yang, Yingwei Cui, Hector Garcia-Molina, and Jennifer Widom. 1999. Performance issues in incremental warehouse maintenance. In *In Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00), Cairo, Egypt, September 2000*. Stanford InfoLab.

[19] Per-Ake Larson and Jingren Zhou. 2007. Efficient maintenance of materialized outer-join views. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 56–65.

[20] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal* 27, 5 (2018), 643–668.

[21] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. 1997. Maintenance of data cubes and summary tables in a warehouse. *ACM Sigmod Record* 26, 2 (1997), 100–111.

[22] Milos Nikolic, Mohammad Dashti, and Christoph Koch. 2016. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proceedings of the 2016 International Conference on Management of Data*. 511–526.

[23] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. 2002. Incremental maintenance for non-distributive aggregate functions. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 802–813.

[24] Xiaolei Qian and Gio Wiederhold. 1991. Incremental recomputation of active relational expressions. *IEEE transactions on knowledge and data engineering* 3, 3 (1991), 337–341.

[25] Dallan Quass. 1996. Maintenance expressions for views with aggregation. (1996).

[26] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. 1996. Making views self-maintainable for data warehousing. In *Fourth International Conference on Parallel and Distributed Information Systems*. IEEE, 158–169.
[27] Kenneth A Ross, Divesh Srivastava, and S Sudarshan. 1996. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. 447–458.
[28] Qichen Wang and Ke Yi. 2020. Maintaining Acyclic Foreign-Key Joins under Updates. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1225–1239.
[29] PostgreSQL wiki. 2020. Incremental View Maintenance. https://wiki.postgresql.org/wiki/Incremental_View_Maintenance. [Online; accessed 9-May-2021].
[30] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. 2019. Analyticdb: Real-time olap database system at alibaba cloud. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2059–2070.
[31] Jingren Zhou, Per-Ake Larson, and Hicham G Elmongui. 2007. Lazy maintenance of materialized views. In *Proceedings of the 33rd international conference on Very large data bases*. 231–242.