ViewDF: Declarative Incremental View Maintenance for Streaming Data

Yuke Yang, Lukasz Golab and M. Tamer Ozsu

University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

{*y274yang, lgolab, tozsu*}*@uwaterloo.ca*

Abstract

We present ViewDF: a flexible and declarative framework for incremental maintenance of materialized views (i.e., results of continuous queries) over streaming data. The main component of the proposed framework is the View Delta Function (ViewDF), which declaratively specifies how to update a materialized view when a new batch of data arrives. We describe and experimentally evaluate a prototype system based on this idea, which allows users to write ViewDFs directly and automatically translates common classes of streaming queries into ViewDFs. Our approach generalizes existing work on incremental view maintenance and enables new optimizations for views that are common in stream analytics, including those with pattern matching and sliding windows.

Keywords: Data stream management, View maintenance, Pattern matching

1. Introduction

Traditional database management systems include On-Line Analytical Processing (OLAP) systems, focusing on deep analytics over read-only data, and On-Line Transactional Processing (OLTP) systems, optimized for frequent updates, insertions and deletions. Recently, new data management solutions have been proposed to handle the three Vs of big data: volume, velocity and variety. In particular, a new application area has appeared in response to high velocity: *append-only* data stream management [24]. Data streams naturally arise in many applications such as the Internet-of-Things (IoT), and include sensor measurements, GPS fixes, system logs, Web clicks, social media interactions, financial transactions and results of scientific experiments. In these applications, new data are continuously produced, while existing data such as past measurements are not modified.

Early stream processing systems have operated in stream-in-stream-out-mode: data are processed sequentially as they arrive, without being stored permanently, and results are streamed out for consumption by other systems and applications. To enable stream-in-stream-out processing, early research on data stream systems focused on

non-blocking, single-pass execution of lightweight SQL-like operators (see, e.g., the STREAM system [3]).

On the other hand, modern stream systems tend to ingest data per-batch rather than per-tuple, and they allow persistent storage of histories of streams and results of continuous queries (i.e., materialized views). Since raw data usually need to be processed before they are suitable for analysis, applications typically create materialized views containing aggregates, joins, or higher-level events and entities computed from individual streaming records. Furthermore, adding persistent storage to stream processing systems enables complex analytics over new and historical data without the computational and administrative overhead of separate systems. For instance, infrastructure monitoring applications such as network or datacenter monitoring often correlate the current state of the entities being monitored with their past behaviour, e.g., to determine if and why a similar problem has occurred in the past and to suggest a solution before the problem spreads [7, 20]. Thus, historical data are required for real-time alerting and change detection as they provide context and a baseline for new data.

Storage-enabled stream systems based on relational database systems, sometimes referred to as Data Stream Warehouses, include Data Depot [21], DBStream [6], PipelineDB [33], Tidal Race [21] and Truviso [27]. Additionally, Flink [9] and Spark Streaming [41] are two distributed data processing systems designed to support streaming and offline analytics. In this paper, we study a fundamental technical problem faced by these types of systems: materialized view maintenance over streaming data to enable (nearly) real-time analytics.

View maintenance—in particular incremental view maintenance (IVM)—is not a new problem [13, 25]. However, early work and recent developments (e.g., DBToaster [2]) focus on views with standard relational operators, possibly including group-by aggregation and subqueries with semijoins or antijoins [38]. In contrast, stream analytics involve new operators such as sliding window aggregation and pattern matching [1, 12, 22]. Incrementally maintaining materialized views with these operators introduces new challenges.

To address these challenges, various solutions have been proposed for sliding window aggregation (see, e.g., [4, 30, 32, 36]), and event processing systems have been proposed for pattern matching on streaming data (see, e.g., [1, 15, 16, 39]). The solution we present in this paper, named ViewDF, is a flexible and *declarative* framework for IVM over data streams that generalizes existing techniques and enables new optimizations for stream analytics.

In traditional data warehouses, materialized views are specified as SQL queries over their source tables via a CREATE MATERIALIZED VIEW statement. The idea behind ViewDF is simple: we augment view definition statements with View Delta Functions (ViewDFs) that *declaratively* specify how to update views when a batch of new data arrives. An SQL-like declarative specification is desirable because it can be directly optimized and executed by an underlying database system. In addition to allowing users to write ViewDFs directly, we want to automatically translate queries to incremental ViewDFs whenever possible.

To exploit temporal locality in the context of view maintenance, ViewDF relies on *temporal partitioning*. For a very simple example, when computing an aggregate function over a sliding window of length 60 minutes, we never need to access parts of

tables containing data more than 60 minutes old.

The contributions of this paper are as follows.

1. We present the ViewDF framework for declarative incremental view maintenance over streaming data. The proposed framework exploits the append-only nature of data streams and the temporal locality of view maintenance. ViewDF allows users and applications to specify, using SQL, how a batch of new data affects the view.
2. We present algorithms for automatically translating two useful classes of streaming queries into incremental ViewDFs: event processing queries and sliding window aggregates. (However, ViewDF is a flexible framework and can be extended to other types of incrementally-computable queries.)
3. We implemented a prototype ViewDF framework using PostgreSQL, and we experimentally show its effectiveness.

The remainder of this paper is organized as follows. Section 2 introduces the ViewDF approach using a pattern matching query as an example; Section 3 discusses previous work; Section 4 gives the details of the ViewDF framework Section 5 presents translations of streaming queries to ViewDFs; Section 6 classifies the types of views that fit or do not fit the ViewDF framework; Section 7 presents experimental results; and Section 8 concludes the paper with directions for future work.

2. Example and Solution Preview

We introduce the ViewDF approach using an example drawn from network monitoring, which has been a popular motivating application for data stream analytics [6, 21, 22]. Suppose we have a data stream containing quality-of-service measurements obtained from the network, such as packet loss between various source-destination pairs (measured by sending control packets and checking how many arrive at their destination). Each record contains a timestamp, the source (`src`) and destination (`dest`) IP addresses, followed by the measurements taken at that time. Suppose that every minute, a batch of new records arrives with one record for each source-destination pair being monitored, containing the number of lost packets for the given pair over the past minute.

Let us store the stream in table M. We partition M it into one-minute parts, each corresponding to one batch of data. Let $M[i]$ be the ith part of M and let $M[i..j]$ denote the union of the ith through jth parts for $i < j$. We assume that each part is a separate logical (and perhaps also physical) table that can be accessed directly. For concreteness, assume $M[i]$ corresponds to a logical table named M_i, and $M[i..j]$ corresponds to `UNION ALL` of M_i through M_j.

We now explain the partition subscripts. The idea is to divide the Unix timestamp of the ith part by its time span. For instance, a part storing data from January 1 2015 at 0:00 hours has the subscript of 1420070400 (the Unix timestamp at that time) divided by 60 (the number of seconds in one minute), i.e., $M[23667840]$. The next part, storing data from January 1 2015 at 0:01 hours, is therefore $M[23667841]$. Similarly, for a table partitioned by hour, the subscript of the part starting at January 1 2015 at 0 hours

is 1420070400 divided by 3600 (the number of seconds in an hour), which is 394464. Note that the subscript of the oldest part reflects the timestamp of the oldest batch of data, and is not necessarily zero unless the given table or view has been collecting data since Unix time zero.

Suppose we want to identify, at any point in time, all the source-destination pairs that have reported high packet loss (say, at least ten lost packets) for at least four consecutive measurements; i.e., for at least four consecutive minutes. Additionally, for each such pair, suppose we want to report the number of consecutive measurements with high loss and the total number of lost packets during this interval. One way to define this view, call it `View1`, is shown below, assuming a syntax similar to that of event processing languages such as SASE [1]. The attributes `src` and `dest` define a source-destination pair being monitored, and `loss` measures the number of packets lost per-minute. The `PATTERN` expression `[a, b, c, d+]` indicates that we are looking for four or more consecutive tuples; a plus symbol means one or more tuple. The first three tuples will be bound to variables `a`, `b` and `c`, respectively, while the remaining tuples will be bound to `d`. The `WHERE` condition specifies that each tuple satisfying the pattern must have `loss>10`. The `GROUP BY` clause states that we are separately looking for patterns in each sequence corresponding to a particular source-destination pair. For each result tuple, `ct` counts the number of tuples satisfying the pattern, i.e., the number of consecutive measurements with at least ten lost packets, and `sum_loss` is the sum of the loss values over the tuples satisfying the pattern.

```
CREATE VIEW View1 AS
  SELECT src, dest, count(*) AS ct,
         sum(loss) AS sum_loss
  FROM M PATTERN [a, b, c, d+]
  WHERE a.loss>10 AND b.loss>10
    AND c.loss>10 AND d.loss>10
  GROUP BY src, dest
```

For example, consider the following sequence of timestamp and packet loss measurements for a particular source-destination pair. This pair is in the output at time 10:04, with `ct=4` and `sum_loss=71`. It is also reported at 10:05 with `ct=5` and `sum_loss=87`. It is no longer reported at 10:06 because at that time, it has not produced at least four consecutive measurements of at least 10 lost packets.

```
2015-01-01 10:00, 6
2015-01-01 10:01, 12
2015-01-01 10:02, 15
2015-01-01 10:03, 24
2015-01-01 10:04, 20
2015-01-01 10:05, 16
2015-01-01 10:06, 7
```

The contents of `View1` change over time. As was the case with M, we partition `View1` into one-minute parts. When a new $M[i]$ part is created for a new batch of packet loss measurements, a corresponding `View1[i]` part is created to store the

source-destination pairs reporting at least four consecutive high packet loss measurements as of that minute. Old parts of `View1` may be deleted over time if they are no longer needed.

A naive way to update `View1` when a new batch of packet loss measurements arrives is to re-run the above pattern query on all of M—there may be a source-destination pair that has reported over 10 lost packets since the beginning of the stream, so without scanning all of M we could not compute the correct `ct` and `sum_loss` values for it. However, this strategy will not only find those source-destination pairs which have reported at least four consecutive high-loss measurements as of the current time, but it will also recompute all such patterns that happened in the past! Clearly, we need an incremental maintenance strategy.

In this paper, we advocate expressing view updates directly in a declarative SQL-like manner. Observe that `View1` is append-only: when a new batch of data creates a new part of M, a corresponding new part of `View1` is created, and other parts of `View1` do not change. As we will show below, ViewDF can directly specify the contents of a new part of a view, by referring to one or more parts of the source table(s) as well as one or more previous parts of the view itself.

To produce an IVM strategy for `View1`, we define a `Helper` view that keeps track of `ct` and `sum_loss` for each source-destination pair each minute. When new data arrive, we incrementally compute a new part of the `Helper` view by referring to its previous part and to the new part of M. To compute the final answer, we select from `Helper` the source-destination pairs with `ct` at least 4.

The View Delta Function (ViewDF) for `Helper` is shown below. There are three main components (more details in Section 4):

```
CREATE VIEW Helper AS
INITIALIZE Helper[i] AS
  SELECT src, dest, 1 AS ct, loss AS sum_loss
  FROM M[i]
  WHERE loss>10
UPDATE Helper[j] AS
  SELECT src, dest, ct+1, sum_loss+loss
  FROM (
    SELECT New.src AS src,
           New.dest AS dest,
           Prev.ct AS ct,
           Prev.sum_loss AS sum_loss,
           New.loss AS loss
    FROM M[j] AS New
    LEFT OUTER JOIN Helper[j-1] AS Prev
    WHERE New.loss>10 )
PARTITION LENGTH 60
```

1. The `INITIALIZE` statement contains a query that defines the initial part (and, implicitly, the schema) of the view. Here, the first part of `Helper` contains the source-destination pairs that reported over 10 lost packets at that time. When the ViewDF is initially created, the `INITIALIZE` query is executed with references to parts resolved to their logical table names. The subscript i is set to that of

the newest part of M that currently exists, and the results are loaded into the corresponding `Helper[i]` part. This is the first non-empty part of `Helper`.

2. The `UPDATE` statement contains a query that defines the contents of the ith view part. Here, whenever a new $M[j]$ part is created for a new batch of data, the `UPDATE` query is executed and its results are loaded into a new `Helper[j]` part. Since both M and `Helper` are partitioned by minute, new parts will be created for them at the same pace. The `UPDATE` query performs a left outer join of the new part of M with the *previous* part of `Helper`, and updates `ct` (by incrementing it) and `sum_loss` (by adding to it the number of lost packets over the most recent minute) for the source-destination pairs that continue reporting over 10 lost packets. We need a left outer join because we want to examine all the source-destination pairs currently appearing in $M[j]$, even if they do not have a record in `Helper[j-1]`, i.e., they have not reported at least 10 lost packets in the previous minute.
3. The `PARTITION LENGTH` statement specifies the time span of each part (`Helper` is partitioned by 60 seconds, and so is M).

It is now easy to define `View2`, which is an incremental ViewDF version of `View1`, as a simple selection query over the above `Helper` view.

```
CREATE VIEW View2 AS
INITIALIZE View2[i] AS
  SELECT src, dest, ct, sum_loss
  FROM Helper[i]
  WHERE ct>=4
UPDATE View2[j] AS
  SELECT src, dest, ct, sum_loss
  FROM Helper[j]
  WHERE ct>=4
PARTITION LENGTH 60
```

Figure 1 illustrates the data required to compute a new part of `View2`; each table is divided into rectangles, with each rectangle corresponding to one part storing one minute of data. On the left, the naive approach needs to scan all of M in the worst case, as we explained earlier. On the right, the optimized approach, implemented in the ViewDF for `View2`, only needs data from the latest (rightmost) part of M and the previous part of the `Helper` view to create a new part of the `Helper` view; then, this new part suffices to update `View2`.

As the above example shows, the proposed ViewDF framework enables a declarative specification of incremental view maintenance over streaming data. ViewDF exploits the append-only nature of data streams and is applicable to views that themselves evolve in an append-only manner. To exploit temporal locality, ViewDF expresses view maintenance operations at a granularity of temporal parts, which avoids scanning large tables. This is critical for streaming queries, whose results can often be refreshed only by accessing recently-arrived data. Furthermore, the SQL-like foundation of ViewDF makes it compatible with, and leverages the query optimizer and engine of, any underlying database system. In Section 5, we will show how to automatically translate

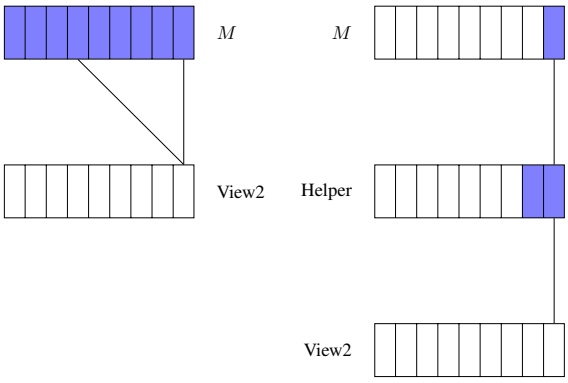

Figure 1: Computing a new part of View2 using a naive view maintenance approach (left) vs. a ViewDF incremental strategy (right).

queries such as `View1` into incremental ViewDFs; additionally, our framework allows users to specify ViewDFs directly.

3. Related Work

This paper is an extended version of a workshop paper presented at the 2015 VLDB workshop on Business Intelligence for the Real Time Enterprise (BIRTE) [40]. There is new material on sliding window aggregation using the ViewDF approach (Section 5.2 and 7.2), new material on dealing with out-of-order data (Section 4.3), a significant extension of the ViewDF framework description (Section 4.1), a new classification of materialized views that naturally fit the ViewDF framework (Section 6), and additional explanations and citations throughout the paper.

ViewDF is related to previous work on incremental view maintenance (IVM). However, previous work on IVM focuses on standard relational operators, possibly including group-by aggregation and subqueries with semijoins or antijoins [2, 13, 25, 38]. ViewDF targets materialized views over append-only data streams that include temporal and sequential operators such as sliding windows and pattern matching. While many stand-alone algorithms and optimizations have been proposed for these types of queries [1, 4, 15, 16, 30, 32, 36, 39], to the best of our knowledge ViewDF is the first general and declarative framework for incremental view maintenance over streaming data.

As we showed in Section 2, the ViewDF approach may require additional "Helper" views to make the final view incrementally maintainable. This is similar to the traditional data warehouse notion of maintaining auxiliary views to make the final view self-maintainable without accessing raw data; see, e.g., [34]. Again, previous work in this space addresses standard relational operators rather than streaming operators.

In addition to the classical view maintenance literature, there has been work that directly targets view maintenance over streams [19]. However, this work addressed the problem of computing a stream of insertions and deletions to a view over time,

whereas ViewDF addresses a different problem of incrementally propagating a batch of new data to materialized views.

Conceptually, perhaps the closest work to ViewDF is ATLaS [37], which is a declarative system for user-defined functions (UDFs). A UDF specified in ATLaS has an INITIALIZE and ITERATE component, similar to a ViewDF (and possibly also a TERMINATE component). However, an ATLaS UDF specifies how to compute a function over a stream one tuple at a time, whereas a ViewDF specifies how to maintain a temporally-partitioned materialized view one batch (part) at a time. Furthermore, ATLaS does not use the notion of temporal partitioning, which is a critical component of ViewDF as it enables efficient IVM. Finally, ATLaS did not consider translating queries to UDFs.

Temporal partitioning is a common technique for storing append-only data in data warehouses [6, 17, 21, 26]. Partition relationships have been used in previous work for view maintenance; for example, to compute monthly aggregates, it suffices to scan only those parts of the base table which contain data for that month. In ViewDF, we go a step further and allow view maintenance queries to reference parts of source tables as well as previous parts of the view itself. This enables new IVM strategies and further reduces the amount of data that need to be accessed during view maintenance.

Finally, there has been recent work on incremental distributed computation; see, e.g., [11, 14, 28, 31]. However, this body of work focuses mainly on extending the map/reduce processing model to incremental computation rather than ViewDF's goal of declarative specification of IVM over streaming data.

4. The ViewDF Framework

We now build on the material presented in Section 2 and discuss the details of ViewDF.

4.1. System Description

Figure 2 outlines the architecture of the ViewDF framework; in the remainder of the paper, we abuse terminology and refer to both an individual view definition and the framework as ViewDF. The bottom box represents an underlying database system, which stores base tables and (hierarchies of) materialized views. The database system also runs ad-hoc user queries and periodically runs the view update queries specified in the ViewDFs. The ViewDF layer is illustrated in the top box.

The ViewDF layer accepts ViewDF definitions directly and also includes a translation layer. In the latter, users enter CREATE VIEW statements containing selected classes of queries (e.g., `View1` from Section 2), and these are automatically translated into ViewDF expressions for incremental maintenance. In Section 5, we describe the translation algorithms for event processing queries and sliding window aggregates.

Recall the motivating example from Section 2. To exploit the temporal locality of streaming queries, ViewDF requires direct access into individual parts of tables and views. Thus, every streaming base table and view must be logically (and perhaps also physically) partitioned by time; however, there may be some dimension tables that store slowly-changing data and are not partitioned. Aside from logical temporal partitioning, ViewDF does not require any particular physical data layout. For example,

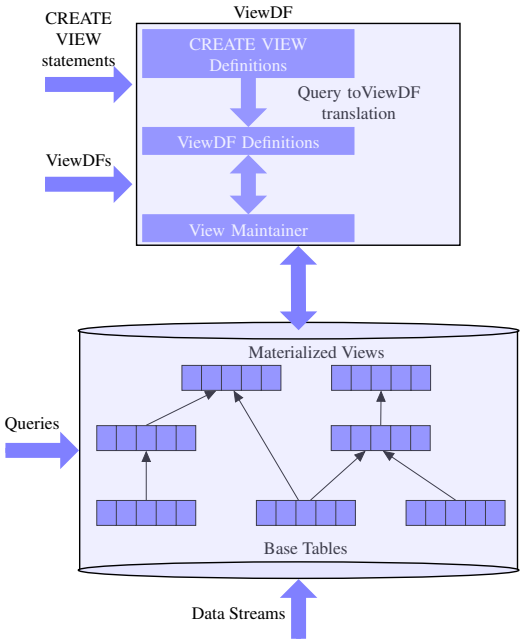

Figure 2: ViewDF architecture.

tables may be stored in row-oriented format and horizontally partitioned by time, or in matrix-based format, with one row per time series, vertically partitioned by time. Thus, ViewDF can use any underlying data management system that supports partitioning; e.g., PostgreSQL. Technically, even if partitioning is not supported, ViewDF can still be used if each part is defined as a separate table.

Data streams are assumed to arrive in batches, e.g., every minute. For now, assume that data arrive in batch order (we will discuss out-of-order arrivals in Section 4.3). For example, a one-minute batch arriving at 10:00 is assumed to contain data with timestamps between 9:59:00 and 9:59:59. Base tables are typically partitioned such that each part corresponds to one batch. The time span of each part of a materialized view typically equals the maximum part size of the tables or views over which it is defined. Over time, small parts may be *rolled-up* into larger ones, as in, e.g., [21]. For example, the most recent day may be partitioned by minute and older data may be partitioned by day. This way, historical queries spanning many days do not have to access thousands of small parts. Very old parts (of base tables and views) may be archived or deleted if they are no longer needed.

The View Maintainer module loads new data and propagates changes throughout the views as per the ViewDF expressions. When a batch of new data arrives, a new part of the corresponding base table is created for it. This may trigger the creation of new parts of materialized views that depend on this base table.

As we explained in Section 2, a ViewDF expression specifies an INITIALIZE query, and UPDATE query and the PARTITION LENGTH (time span of each part).

The queries can be arbitrary SQL queries supported by the underlying database system, and they may contain partition references of the form Table$[part]$. As mentioned earlier, partition references allow us to exploit the temporal locality of materialized views over streaming data: rather than scanning all the source tables (or views), it often suffices to access a small set of parts when refreshing the results. For concreteness, assume that partition references resolve at runtime to logical table names of the form $Table\_part$.

As we discussed in Section 2, partition subscripts are consecutive integers computed by dividing the Unix time of a batch of data by the PARTITION LENGTH. We store an additional catalog table that maintains, for each base table and materialized view, its PARTITION LENGTH and a list of its parts that have already been created and loaded with data.

An INITIALIZE query can reference one or more source tables, and one or more of their parts. At the time of initializing the view, we query the catalog table for the subscript of the most recent part of each source table and we take the minimum of these[1]. This becomes the value of the partition subscript i, i.e., the part which will store the output of the INITIALIZE query. We then translate partition references to logical table names, run the query, and insert its output into the ith part of the view. Finally, we add a row for this view to the catalog table and add i to the list of its parts.

We now describe the UPDATE step. At any point in time, the catalog table can tell us the subscript of the newest part of each base table and materialized view. Adding one to these subscripts gives the next part that will be created for each table and view, i.e., the j in the next UPDATE query. For example, recall the `Helper` view from Section 2 and say $j = 1000$ at the current time. Now, a simple examination of the UDPATE query tells us which source table parts are required for `Helper[1000]`, namely `S[1000]` and `Helper[999]`. And, now a simple query against the catalog table tells us if all of these required parts already exist. If so, we launch an update of `Helper` which will create `Helper[1000]`.

Generalizing the above example, the view maintainer works as follows. For each view V registered in the system, we maintain the subscript of the next part that is to be created, call it NEXT[V]. For each view V, we also maintain a bitmap with one entry for each part of each source table/view required to compute the next part of V, call it B[V]. When a new base table or view part is created, we scan through all the B[V]s and turn on all the bits corresponding to this new part. We then execute the UPDATE queries of all the views which have all their bits set to one. In other words, we run the UPDATE query of a view V as soon as all the required parts of its source tables/views have been created. When an UPDATE query of V terminates, we increment NEXT[V], the next part subscript, and we recalculate B[V] to see which parts required for the *next* part of V already exist.

Returning to the above example, suppose NEXT=1000 for the `Helper` view, i.e., we have just created `Helper[999]`. Its bitmap consists of two entries; at this point in time, one is for `S[1000]` and one is for `Helper[999]`. As soon as `S[1000]`

[1]For instance, if the INITIALIZE query is a join of the newest parts of two base tables, S and T, and S has parts up to 394464 but T only has parts up to 394462, then we take $i = 394462$.

is created, we will have all the data we need to create `Helper[1000]` as per its UPDATE query.

Our discussion thus far assumes that views are updated immediately after all the data they need becomes available. However, the ViewDF framework is compatible with other methods for ordering view updates, e.g., those which prioritize out-of-date views to improve freshness (see, e.g., [23]) or those which schedule related views together to minimize cache misses (see, e.g., [5]).

4.2. Additional Examples

So far, we assumed that views have the same PARTITION LENGTH and therefore the same refresh frequency as their source table(s). Below, we give an example of a view partitioned by hour (3600 seconds) over a network measurement table `M` partitioned by minute. For brevity, we only show the INITIALIZE query. Every hour, this view computes hourly sums of lost packets for each source-destination pair. Suppose we are at the beginning of time. The first part of `View3`, storing data for the first hour, requires the first 60 minutes of time, i.e., the first 60 parts of `M`. Thus, `View3[1]` requires `M[1..60]`, and, more generally, `View3[i]` requires `M[i*60-59 .. i*60]`. For `View3`, at initialization time we take the largest part subscript i of `M`, divide it by 60 (which is the ratio of the partition length of `View3` and `M`) and round down. This gives us the subscript for the first part of `View3`.

```
CREATE VIEW View3 AS
INITIALIZE View3[i] AS
  SELECT src, dest, sum(loss) as sum_loss
  FROM M[i*60-59 .. i*60]
  GROUP BY src, dest
...
PARTITION LENGTH 3600
```

Next, we give an example of a view that joins multiple streams, which is also supported in ViewDF. Suppose we have two base tables, M and N, both partitioned into one-minute parts. Suppose we want to materialize a *band join* of M and N into a view named J; in a band join, a tuple from M joins with a tuple from N if the join predicate is satisfied and both tuples have timestamps belonging to the same one-minute part. A corresponding ViewDF is shown below, assuming a natural join. Note that J is also partitioned into one-minute parts.

```
CREATE VIEW J AS
INITIALIZE J[i] AS
  SELECT *
  FROM M[i] NATURAL JOIN N[i]
UPDATE J[j] AS
  SELECT *
  FROM M[j] NATURAL JOIN N[j]
PARTITION LENGTH 60
```

Finally, we explain how an ad-hoc query can access an old part of a materialized view. Recall `View2` from Section 2 and suppose we want to find all the source-destination pairs that have reported four consecutive high packet-loss measurements

as of January 1 2015 at 0:00 hours. We could divide the Unix timestamp of this date by 60 and directly query the logical table `View2_23667841`, but this is cumbersome. Instead, in ViewDF we expose a `PART_TIMESTAMP` attribute for each view. The above query can be written as:

```
SELECT *
FROM View2
WHERE PART_TIMESTAMP="2015/01/01 0:00"
```

4.3. Dealing with Out-of-Order Data

Stream processing systems usually assume that the inputs are ordered by arrival time, in which case the notion of delayed or out-of-order data is not relevant. However, some storage-enabled stream systems and traditional data warehouses support two timestamps: *system time*, denoting the time of arrival or insertion of the data, and *application time*, denoting when the given tuple was actually created or valid in real life [8, 18]. In this case, tuples may be late or out-of-order with respect to their application timestamps.

A simple solution to deal with some late arrivals is to buffer the input and re-order it in the buffer if necessary (see, e.g., [10]). However, the larger the buffer the longer the latency. Instead, most storage-enabled stream systems compute answers based on the data that have arrived so far and may recompute some answers in response to late data. Below, we show an example of how ViewDF can deal with this issue.

Recall the example from Section 2, in which `View2` maintains the source-destination pairs having high packet loss. Suppose a new batch of data has arrived and was loaded into the 9th part of the base table: `M[9]`. As shown in Figure 3, the new batch of data will trigger the creation of a new `Hepler[9]` part, which will be computed using `M[9]` and `Helper[8]`. Finally, `View2[9]` will be computed using `Helper[9]`. Additionally, suppose that some late data, i.e., older measurements, have arrived into `M[5]`. This means that `Helper[5]` may have to be recomputed (by running its ViewDF UPDATE query, which requires `M[5]` and `Helper[4]`). Furthermore, since `Helper[5]` may have changed, `View2[5]`, shown in red in Figure 3, may have to be recomputed.

Fortunately, the ViewDF UPDATE statements contain all the information we need to determine how late data arriving into some base table affect any views defined over this table. For instance, since `Helper[j]` is computed by joining `M[j]` with `Helper[j-1]`, we know that late data in `M[5]` may affect `Helper[5]`. In turn, this may affect `View2[5]` since its UPDATE statement references `Helper`, and so on. Similar partition-based reasoning has been used in other work on storage-enabled stream processing [26, 27] (however, no prior system allows views to reference older parts of themselves, as ViewDF does).

One consequence of propagating late data to materialized views is that previously computed view partitions, e.g., `View2[5]` from Section 2, may change in the future. In ViewDF, we maintain a *last updated* timestamp for each part of each view. This way, users and applications can identify parts that may have changed since they last accessed them.

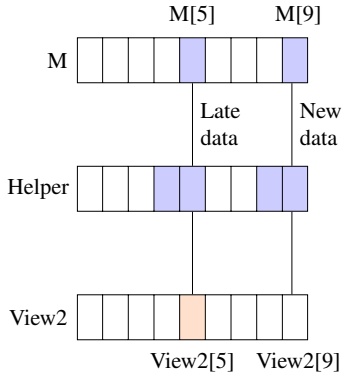

Figure 3: An example of late-arriving data and its consequences on materialized views.

5. Translating Queries to ViewDFs

While users may write ViewDFs directly, useful classes of streaming queries can be automatically rewritten into incremental ViewDFs. We show how to do this for event processing queries (similar to `View1` from Section 2) and sliding window aggregates in Sections 5.1 and 5.2, respectively. These incremental ViewDFs can then be optimized and executed by the underlying database management system.

5.1. Event Processing Queries

The event processing queries we support have the following format.

- The SELECT clause must include all the GROUP BY attributes, may include a COUNT(*) expression, and may include SUM() aggregates over any attributes. Aggregates are computed over all tuples that match the pattern specified in the query (recall `ct` and `sum_loss` in `View1`).
- The FROM clause can only include a single input stream, call it InputStream (but the input to the event processing query can be a materialized view that joins multiple streams).
- The PATTERN clause may contain an arbitrary number of variables, including those with plus symbols (denoting one or more tuple).
- The WHERE clause may contain simple arithmetic predicates on any attributes such as `a.loss>10` as well as predicates referencing two variables such as `b.loss>a.loss` (which can express patterns such as sequences of increasing loss values).
- The GROUP BY clause must include one or more attributes from the input stream if the input stream is composed of multiple sequences (such as measurements from different source-destination pairs).

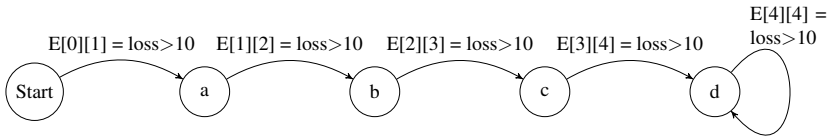

Figure 4: FSM for View1.

The first step is to covert the pattern query into a Finite State Machine (FSM). The second step is to translate the FSM into two ViewDFs: a Helper view and a final view, similar to those shown in Section 2. We discuss these two steps below.

5.1.1. Query to FSM Conversion

An FSM $F = (S[], E[][])$ consists of a set of states S and a set of directed edges and their state transition predicates E. Figure 4 shows the FSM for `View1`. The set of states follows directly from the PATTERN clause; additionally, we include a begin state $S[0]$. The last state, $S[4]$, is the accepting state.

An edge $E[i][j]$ connects states $S[i]$ and $S[j]$ and includes a state transition predicate that determines when to move $S[i]$ to $S[j]$. These predicates correspond to the WHERE predicates in the query. In `View1`, to move from $S[0]$ to $S[1]$, we need a tuple with $loss > 10$; to move from $S[1]$ to $S[2]$, we need the next tuple in the sequence to also have $loss > 10$, and so on. If the state transition predicate is not satisfied at any point, e.g., the next loss measurement is below ten, we go back to the begin state.

We refer to a state with a plus symbol, denoting one or more occurrences of a pattern, as a KleeneClosure state. These states have self-edges $E[i][i]$. In `View1`, we will remain in the accepting state S[4] as long as the next tuple has $loss > 10$.

Algorithm 1 translates a pattern query Q into a FSM and a list $auxlist$ that contains additional variables we will have to maintain in the Helper view. The set of states $S[]$ is computed in lines 1-6 based on the PATTERN clause. The state transition predicates are computed in lines 7-19 by iterating through each WHERE predicate $pred$.

If $pred$ references a single pattern variable (state) $S[p]$, then it can be one of two transition predicates. For a KleeneClosure state, it is $E[p][p]$; otherwise it is $E[p-1][p]$. In either case, we add the predicate to the corresponding edge label (lines 10 and 12, respectively). The *add* function also performs some string editing: it replaces a PATTERN variable with "New"; e.g., `a.loss>10` becomes `New.loss>10`. The add function also adds "AND" between different predicates for the same edge. For instance, if we have `a.loss>10 AND a.loss<20`, then $E[0][1].add$ will be called twice and will result in `New.loss>10 AND New.loss<20`. These edits simplify the next step of converting the FSM into a ViewDF.

Otherwise, if $pred$ references two pattern variables $S[p-1]$ and $S[p]$, e.g., $b.loss > a.loss$, then this is a transition predicate $E[p-1][p]$. In this case, the *add* function on line 16 replaces the $S[p-1]$ variable with "Prev" and the $S[p]$ one with "New"; e.g., `b.loss > a.loss` becomes `New.loss>Prev.loss`. As before, an "AND" is added if there are multiple predicates for this edge. Furthermore, to evaluate such a predicate, we need to store some additional information in the Helper view, namely

Algorithm 1: GENERATE_FSM

Input: A pattern query Q
Output: a FSM $F(S[], E[][]), auxlist$

```
1: S[0] := begin state;
2: j := 1;
3: for all variables v in the PATTERN clause do
4:    S[j] := v;
5:    j++;
6: end for
7: for all predicates pred in WHERE clause do
8:    if pred references a single state S[p] then
9:       if S[p].isKleeneClosure then
10:         E[p][p].add(pred);
11:      else
12:         E[p − 1][p].add(pred);
13:      end if
14:   end if
15:   if pred references two states S[p] and S[p − 1] then
16:      E[p − 1][p].add(pred);
17:      auxlist.addAttr(pred);
18:   end if
19: end for
```

$a.loss$. In line 17, the *addAttr* function retrieves the attribute name referenced with the $S[p-1]$ variable and appends "New" to it, i.e., for `b.loss > a.loss`, $auxlist$ will contain `New.loss`.

When given `View1` as input, Algorithm 1 returns the FSM illustrated in Figure 4 (however, the edge labels are actually `New.loss>10`) and an empty $auxlist$ (there are no predicates referencing two pattern variables in `View1`).

5.1.2. FSM to ViewDF Conversion

Given a pattern query Q and the output of Algorithm 1 (i.e., the FSM and *auxlist*), we can now generate ViewDFs for the Helper view (Algorithm 2) and the final view V (Algorithm 3) corresponding to Q. Again, to explain these two algorithms, we use `View1` as the input query.

First, we discuss the Helper view. The output of Algorithm 2 given `View1` and the corresponding FSM is shown below. Notice that the Helper ViewDF below is different (more general) than the one shown in Section 2, which was a hand-crafted ViewDF for a simple pattern query.

Algorithm 2: GENERATE_HELPER_VIEWDF

Input: A pattern query Q, FSM $F(S[], E[][])$, $auxlist$, partition length L

1: n := number of states in F;
2: Write("CREATE VIEW Helper AS");
3: Write("INITIALIZE Helper[i] AS SELECT");
4: **for all** non-aggregate attributes a in SELECT clause of Q **do**
5: Write("New." + a + ",");
6: **end for**
7: **for all** attributes a in SELECT clause of Q with SUM(a) **do**
8: Write("New." + a + " AS sum_" + a + ",");
9: **end for**
10: **if** SELECT clause of Q includes count(*) **then**
11: Write("1 AS ct, ");
12: **end if**
13: Write($auxlist$);
14: Write("1 AS state");
15: Write("FROM InputStream[i] AS New");
16: Write("WHERE " + $E[0][1]$");
17: Write("UPDATE Helper[j] AS SELECT");
18: **for all** non-aggregate attributes a in SELECT clause of Q **do**
19: Write("New." + a + ",");
20: **end for**
21: **for all** attributes a in SELECT clause of Q with SUM(a) **do**
22: Write("Prev.sum_" + a + "+ New." + a);
23: **end for**
24: **if** SELECT clause of Q includes count(*) **then**
25: Write("ct+1, ");
26: **end if**
27: Write($auxlist$);
28: Write("CASE WHEN state=0 then 1");
29: **for** k=0 to $n-1$ **do**
30: **if** $S[k]$.isKleeneClosure **then**
31: Write("WHEN state="+k+"AND "+$E[k][k]$+"THEN state");
32: **end if**
33: Write("WHEN state="+k+"AND "+$E[k][k$+1]+"THEN state+1");
34: **end for**
35: Write("END");
36: Write("FROM (SELECT * FROM InputStream[j] AS New");
37: Write("LEFT OUTER JOIN Helper[j-1] AS Prev");
38: Write("WHERE state <> 0");
39: Write("PARTITION LENGTH L");

Here is how Algorithm 2 generated the above ViewDF for the Helper view. Lines 3-16 generate the INITIALIZE query. The SELECT statement, i.e., schema of the view, contains four parts: the non-aggregate attributes from the SELECT statement of the original query Q (lines 4-6), the attributes whose values we want to sum up (lines 7-9), a counter in case the original query includes count(*) (lines 10-12), and the attributes in $auxlist$ (line 13; there are none for `Viewl`). Notice that ct is initialized to one and sum_loss to the current value of loss. The Helper view also includes a $state$ attribute

which keeps track of which state a source-destination pair is in (line 14). The WHERE clause is the state transition predicate $E[0][1]$ (line 15). Here, we initialize the Helper view by selecting all source-destination pairs having `loss>10`. All such pairs have $state = 1$ at this point.

```
CREATE VIEW Helper AS
INITIALIZE Helper[i] AS
  SELECT New.src AS src, New.dest AS dest,
  New.loss AS sum_loss, 1 AS ct, 1 AS state
  FROM InputStream[i] AS New WHERE New.loss>10
UPDATE Helper[j] AS
  SELECT New.src, New.dest,
  Prev.sum_loss+New.loss, 1+ct,
  CASE WHEN state=0 and New.loss>10 THEN state+1
       WHEN state=1 and New.loss>10 THEN state+1
       WHEN state=2 and New.loss>10 THEN state+1
       WHEN state=3 and New.loss>10 THEN state+1
       WHEN state=4 and New.loss>10 THEN state
  END,
  FROM (
    SELECT *
    FROM InputStream[j] AS New
    LEFT OUTER JOIN Helper[j-1] AS Prev)
  WHERE state <> 0
PARTITION LENGTH L
```

Lines 17-39 generate the UPDATE query. Its SELECT statement includes the non-aggregate attributes from the original query (lines 18-20), as in the INITIALIZE query. The attributes being summed up are incrementally maintained (`Prev.sum_loss+New.loss`; lines 21-23), as is the count of tuples matching the pattern ($ct + 1$; lines 24-26). The CASE statement (lines 28-35) updates the state of each source-destination pair. We consider all possible states and whether the state transition predicates (including self-edges for KleeneClosure states; lines 30-32) are true. If a transition predicate is true, the state variable is incremented. Notice that source-destination pairs that are already tracked in the previous partition of Helper may be in states 1 through 4. Those which are not yet tracked are in state zero and may move to state one if $E[0][1]$ is satisfied (i.e., `loss>10`). These tuples do not have matching records in `Helper[j-1]`, in which case the outer join operator (line 37) assigns zero to the integer attribute state. This is how we know that these are new source-destination pairs which have not reported `loss>10` in the last minute, i.e., they are in the begin state.

Algorithm 3 generates the final view. Given the pattern matching query from `View1` as input, the final view is shown below and named `View4`. The INITIALIZE and UPDATE queries are the same. Their SELECT statements include all the non-aggregate attributes of the original query (lines 5-7), plus all the aggregates (lines 8-13). The main difference between this automatically-generated ViewDF and View2 from Section 2 is the WHERE predicate. Here, the WHERE predicate checks to see if we are in the final (accepting) state (lines 15 and 19).

Algorithm 3: GENERATE_FINAL_VIEWDF

Input: A pattern query Q, FSM $F(S[], E[][])$, partition length L

1: n := number of states in F;
2: Write("CREATE VIEW V AS");
3: Write("INITIALIZE V[i] AS");
4: Write("SELECT");
5: **for all** non-aggregate attributes a in SELECT clause of Q **do**
6: Write(a + ",");
7: **end for**
8: **for all** attributes a in SELECT clause of Q with SUM(a) **do**
9: Write(sum_" + a + ",");
10: **end for**
11: **if** SELECT clause of Q includes count(*) **then**
12: Write("ct");
13: **end if**
14: Write("FROM Helper[i]");
15: Write("WHERE state=" + n);
16: Write("UPDATE V[j] AS");
17: Repeat lines 4-13
18: Write("FROM Helper[j]");
19: Write ("WHERE state=" + n);
20: Write("PARTITION LENGTH L");

```
CREATE VIEW View4 AS
INITIALIZE View4[i] AS
  SELECT src, dest, ct, sum_loss
  FROM Helper[i]
  WHERE state=4
UPDATE View4[j] AS
  SELECT src, dest, ct, sum_loss
  FROM Helper[j]
  WHERE state=4
PARTITION LENGTH L
```

We conclude the discussion of translating pattern queries to ViewDFs by noting that the final view stores the results of the query and the Helper view is needed only to update it. Thus, old parts of Helper may be deleted. However, as explained in Section 4.3, late-arriving data into the base table may cause some Helper parts (and some parts of the final view) to change. If we expect out-of-order data, we need to keep some recent parts of Helper. How many parts to keep depends on how late data can be.

5.2. Sliding Window Aggregates

In this section, we describe how to convert sliding window aggregates to ViewDFs. Incremental evaluation of sliding window aggregates has been studied in previous work; see, e.g., [4, 30, 32, 36]. Thus, rather than proposing a new incremental strategy, we show how to express existing optimizations in the ViewDF framework.

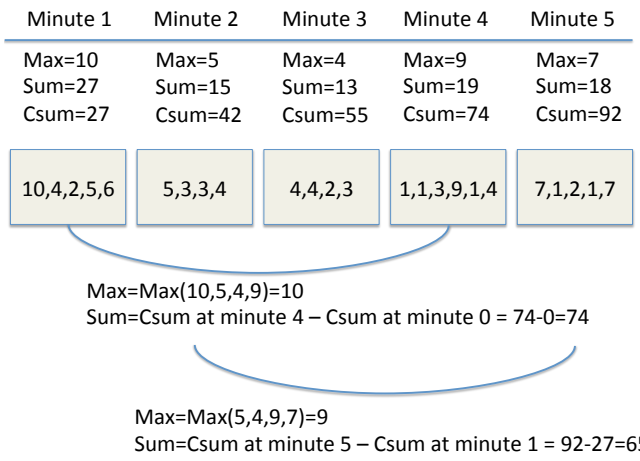

Figure 5: Computing distributive and subtractable aggregates over a sliding window.

Two types of aggregates can be optimized over sliding windows. Let E_1 and E_2 be two multi-sets of numbers. An aggregate function f is *distributive* if $f(E_1 \cup E_2)$ can be computed from $f(E_1)$ and $f(E_2)$ alone. A distributive aggregate function f is *subtractable*, also referred to in the literature as invertible or differential, if for any $E_1 \subseteq E_2$, $f(E_2 - E_1)$ can be computed from $f(E_1)$ and $f(E_2)$ alone. For example, sum, count, maximum and minimum are distributive, and sum and count are also subtractable. Average is neither distributive nor subtractable, but it can be computed from two subtractable functions, namely sum and count.

Figure 5 illustrates how to incrementally evaluate distributive and subtractable aggregates over sliding windows. Assume a new batch of data arrives every minute, and, each minute, we want to compute the aggregates over a four-minute window. In the figure, we illustrate five minutes of data in the five grey rectangles. For distributive aggregates, we pre-compute the aggregate value for each minute and then merge the pre-computed aggregates that are in the scope of the current window. Above the rectangles, we show the maximum and sum values for each minute, as well as the *cumulative sum* (Csum) since the beginning of time which we will use for subtractable aggregates. To compute the maximum over minute 1 through 4, we take the maximum of the pre-computed max values for minutes 1 through 4. When the window slides, we take the maximum of the pre-computed max values for minutes 2 through 5, and so on.

For subtractable aggregates such as sum, we can do better. As illustrated in the figure, to obtain the sum from minute 2 through 5, we take the cumulative sum at minute 5 and subtract the cumulative sum at minute 5 minus the window length, i.e., at minute 1.

5.2.1. ViewDF Translation for Distributive Aggregates

We now explain how to express the above optimization for distributive aggregates in the ViewDF framework using the following query template:

```
SELECT G, max(A)
FROM S [WINDOW W minutes]
GROUP BY G
HAVING H
```

G is a set of attributes, A is the aggregate attribute and H is a HAVING condition over the aggregate values. Assume S is an input stream that arrives in one-minute batches and that the window slides every minute. In each batch, there may be multiple tuples belonging to the same group (in contrast to the pattern matching example from Section 2, in which each batch contained exactly one record for each group, i.e., each source-destination pair).

The translation to ViewDF is simple, so we omit the translation algorithm and only show the output below. The `Helper` view pre-aggregates the maximum value for each group in each minute. The final view, labeled `View5`, then takes the maximum of the pre-aggregated maximum values over the current window, and applies the HAVING predicate. The INITIALIZE and UPDATE queries are the same. Notice that we must compute the first W parts of the `Helper` view before it can initialize `View5`. Also, as was the case with pattern-matching queries, old parts of the `Helper` view may be deleted to save space. We only need the most recent W parts to update the view, plus any additional older parts to handle out-of-order processing.

```
CREATE VIEW Helper AS
INITIALIZE Helper[i] AS
  SELECT G, max(A) as max_A
  FROM S[i]
  GROUP BY G
UDPATE Helper[j] AS
  SELECT G, max(A)
  FROM S[j]
  GROUP BY G
PARTITION LENGTH 60

CREATE VIEW View5 AS
INITIALIZE View5[i] AS
  SELECT G, max(max_A) as max_A
  FROM Helper[i-W+1..i]
  GROUP BY G
  HAVING H
UPDATE View5[j] AS
  SELECT G, max(max_A)
  FROM Helper[j-W+1..j]
  GROUP BY G
  HAVING H
PARTITION LENGTH 60
```

5.2.2. ViewDF Translation for Subtractable Aggregates

Next, we re-use the query template from above, but we replace `max` with `sum`, to illustrate the optimization for subtractable aggregates. This time, the `Helper` view

stores the cumulative sum for each group, which is updated incrementally by combining the previous part of the `Helper` view with the new part of the input stream. The final view, called `View6` subtracts the cumulative sums stored in the $(i - W)$th part of the `Helper` view from those in the ith part, and applies the HAVING predicate, which now includes a `sum_A>0` condition. This removes groups that do not exist in the current window—these groups have the same cumulative sum at the beginning and end of the current window.

```
CREATE VIEW Helper AS
INITIALIZE Helper[i] AS
  SELECT G, sum(A) AS sum_A
  FROM S[i]
  GROUP BY G
UPDATE Helper[j] AS
  SELECT G, sum(Temp_Sum) AS sum_A
  FROM (
    SELECT G, sum(A) AS Temp_Sum
    FROM S[j]
    GROUP BY G
    UNION ALL
    SELECT G, sum_A AS Temp_Sum
    FROM Helper[j-1]
    ) AS Temp
  GROUP BY G
PARTITION LENGTH 60

CREATE VIEW View6
INITIALIZE View6[i] AS
  SELECT Temp.G, sum(Temp.Temp_Sum) AS sum_A
  FROM (
    SELECT G, sum_A AS Temp_Sum
    FROM Helper[i]
    UNION ALL
    SELECT G, sum_A*(-1) AS Temp_Sum
    FROM Helper[i-W]
    ) AS Temp
  GROUP BY G
  HAVING sum_A>0 AND H

  UPDATE VIEW View6[j] AS
  SELECT Temp.G, sum(Temp.Temp_Sum) AS sum_A
  FROM (
    SELECT G, sum_A AS Temp_Sum
    FROM Helper[j]
    UNION ALL
    SELECT G, sum_A*(-1) AS Temp_Sum
    FROM Helper[j-W]
    ) AS Temp
  GROUP BY G
  HAVING sum_A>0 AND H
```

```
PARTITION LENGTH 60
```

There are two space-saving optimizations we can apply to subtractable aggregates. First, as before, we only need the W most recent parts of the `Helper` view. Second, the `Helper` view shown above keeps track of the cumulative sum for each group since the beginning of time; however, we only need the groups that appear in the current window. Thus, we can periodically remove groups that are no longer needed. These groups can be identified by the following query:

```
SELECT Temp.G, sum(Temp.Temp_Sum) AS sum_A
FROM (
  SELECT G, sum_A AS Temp_Sum
  FROM Helper[i]
  UNION ALL
  SELECT G, sum_A*(-1) AS Temp_Sum
  FROM Helper[i-W+1]
  ) AS Temp
GROUP BY G
HAVING sum_A=0
```

6. Discussion

In this section, we discuss the limitations of our approach and we develop a classification of materialized views that fit the ViewDF framework.

Recall the main assumption from Section 4.1: the input consists of one or more timestamped data streams, and each table and view is partitioned by time such that each part can be accessed directly. In general, the ViewDF framework is effective for views whose updates can be expressed in a way that requires access to a small number of parts rather than the whole base table(s).

One general class of views that fit the ViewDF approach are those with implicit or explicit timestamp predicates such as fixed or sliding windows. Such predicates naturally identify which parts store data that are required during a view update. In the worst case—for example, for non-distributive aggregates over sliding windows such as quantiles—the entire window needs to be accessed, which is still smaller than the entire table. For views over multiple streams, this class includes sliding window and band joins (recall Section 4.2).

As discussed in Section 5.2.1, the sub-class of distributive aggregates over sliding windows can be expressed as ViewDFs even more efficiently. Rather than accessing the entire window, only two parts need to be accessed for every update, which provides substantial performance benefits (details to follow in Section 7).

The second class of views compatible with the ViewDF approach are those which are incrementally maintainable by storing a constant amount of additional information. In the context of data streams, two examples of this class are subtractable aggregates (Section 5.2.2) and pattern matching (Section 5.1). In ViewDF the additional information can easily be stored in "Helper" views.

On the other hand, the ViewDF approach cannot improve the efficiency of view updates where each update requires access to the entire input. For example, suppose

we want to maintain a statistical model over an entire data stream and update the model whenever a new batch of data arrives. We may need to recompute the model from scratch and therefore we cannot isolate an update to a small number of parts.

Finally, we comment on the relationship between ViewDF and sequence and array languages such as AQuery [29] and SRQL [35]. This line of research has focused on efficient evaluation of ad-hoc queries written in such languages, which include constructs such as sliding windows and order predicates (e.g., previous or next). Below, we discuss some issues in implementing sequence views in ViewDF assuming an underlying sequence or array query engine.

Consider a stream of stock quotes, with each stream item composed of a timestamp, the stock name, and the current price. Consider the following stream fragment, illustrating stock prices for the Acme company.

```
2015-01-01 10:58, Acme, $5.00
2015-01-01 10:59, Acme, $4.95
2015-01-01 11:00, Acme, $4.97
2015-01-01 11:01, Acme, $4.98
```

Suppose we want to materialize a view which identifies the times when the price of a stock was higher than its *previous* price. In the above example, the records at time 11:00 and 11:01 would be included (and perhaps the record at 10:58, depending on the previous price of Acme stock). Assume this query can expressed in and evaluated by the underlying sequence query engine. Now suppose we write a corresponding ViewDF, partitioned by, say, hour, and specify in the UPDATE clause that the jth part (hour) of the view is to be computed over the jth part (hour) of the stream. This means that the first record of any hour will not have a previous record. Thus, the record at time 11:00 would not be included in the output. This can be fixed by referring to two parts of the stream whenever updating the view: the current hour and the previous hour, but this comes at a cost of additional processing. Generalizing this example, we observe that sequence queries which may span partition boundaries may be implemented in ViewDF, but perhaps not optimally (unless we can specify Helper views to pre-compute the information required from other parts, as in our pattern matching example from Section 5.1).

7. Experiments

We implemented the ViewDF framework, including the translation layer for event-processing queries and sliding window aggregates, on top of PostgreSQL 9.1.3. In this section, we experimentally compare the performance of our system and translation layer with:

1. a naive approach that recomputes the result whenever a batch of new data arrives;
2. a "hard-coded" approach that simulates the ViewDF framework, including partition relationships and any required helper views. The hard-coded approach only uses the database to store and retrieve data (via JDBC) from the relevant partitions, using any available indexes. Query-specific incremental maintenance logic is implemented outside the database as a Java program, using similar physical operators to those of PostgreSQL such as sort-merge join and sort-groupby.

The purpose of the experiments is to show that 1) the ViewDF approach is significantly faster than the naive approach, and 2) the ViewDF approach performs on par with the hard-coded approach, i.e., incurs little or no overhead. The experiments were performed on a workstation with an AMD Phenom II X4 955 3200 MHz processor and 8 GB of RAM, running Ubuntu 12.10. We set the size of the shared memory used by the database server to 600MB.

As we will explain shortly, we generated several synthetic input streams with various distributions. These streams arrive in one-minute batches and resemble the network monitoring stream from Section 2. We wrote a PL/PGSQL program to create new parts of the base table storing the stream whenever new data arrive. We use Postgres' native table partitioning via the table inheritance mechanism. Each runtime number we report is an average of 50 runs.

7.1. Event Processing Queries

We start with `View1` from Section 2, which identifies source-destination pairs that have reported at least four consecutive measurements with high packet loss. We test

1. a naive approach that reruns the pattern query whenever a batch of new data arrives;
2. a hard-coded implementation of `Helper` and `View2` from Section 2, in which we "SELECT *" from the required `Helper` and `View2` parts but perform the outer join in Java as a sort-merge join;
3. the ViewDF approach corresponding to `Helper` and `View4` from Section 5.

Recall that the naive approach may have to access the entire stream in case there is a source-destination pair that has been reporting high loss since the beginning of the stream. We will experiment with different values of $scope$, which forces a limit on how far back the naive algorithm may scan.

In the first experiment, we measure the scalability of the tested approaches. Figure 6 plots the execution time as a function of the number of records (i.e., source-destination pairs) in each data batch. For the naive approach, we set $scope = 20$, i.e., any pair reporting high-loss measurements for more than 20 minutes will have incorrect count and sum-loss numbers. We fix the proportion of loss measurements greater than ten to 10 percent (we will investigate the impact of these parameters on running time shortly). The ViewDF approach is significantly more efficient and scalable than the naive approach, and even slightly faster than the hard-coded approach, which highlights the effectiveness of our translation algorithm. In particular, even with one million tuples per batch, ViewDF can still process the batch and update the view in under five seconds.

Next, we fix the number of tuples per part to one million and vary the $scope$ of the naive algorithm from 5 to 100. Results are shown in Figure 7. Since scope only affects the naive algorithm, Hardcoded and ViewDF have constant running time in this experiment. The naive algorithm becomes very slow as scope increases, as expected, due to having to process more and more data during view updates.

Now, we vary the proportion of tuples with loss measurements greater than ten from 10 to 100 percent while keeping the number of tuples per batch fixed at one million

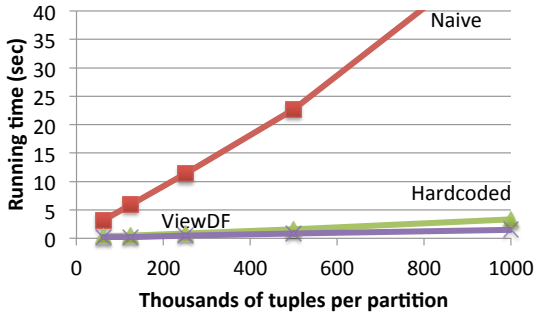

Figure 6: Running time vs. data set size for Naive, Hardcoded and ViewDF.

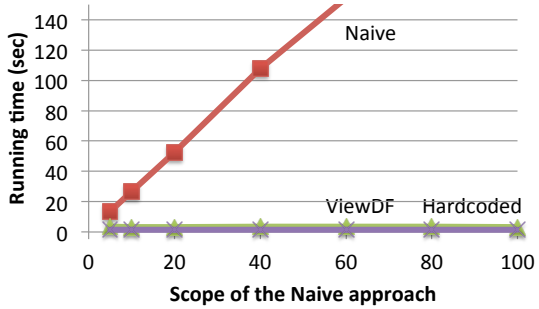

Figure 7: Running time vs. scope for Naive, Hardcoded and ViewDF.

(and, for the naive algorithm, we set scope to a very small value of 4). Results are shown in Figure 8. As the proportion of high-loss tuples increases, all three methods take longer because there are more pattern matches that need to be generated, and because more partial matches need to be tracked in the `Helper` view. However, the naive approach does not worsen much: it always has to scan four partitions anyway and does not maintain any intermediate results. Again, ViewDF is slightly faster than Hardcoded, and remains faster than Naive regardless of the proportion of high-loss values even at this low value of scope. For higher values of scope, the performance gap between Naive and ViewDF is even greater.

7.2. Sliding Window Aggregates

In the previous experiment, the input stream contained exactly one record per source-destination pair per part. In this experiment, we generate ten records per pair per part and execute a MAX (distributive) and SUM (subtractable) window aggregate query. We test

1. a naive approach that recomputes new results from scratch whenever the window slides;

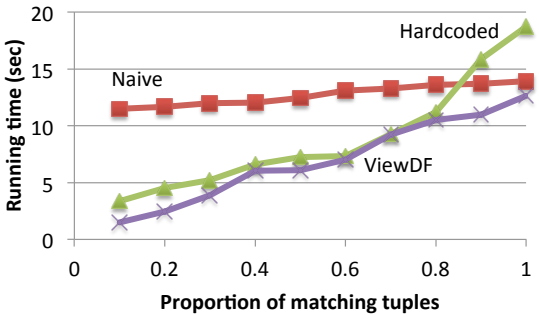

Figure 8: Running time vs. proprotion of tuples having loss>10 for Naive, Hardcoded and ViewDF.

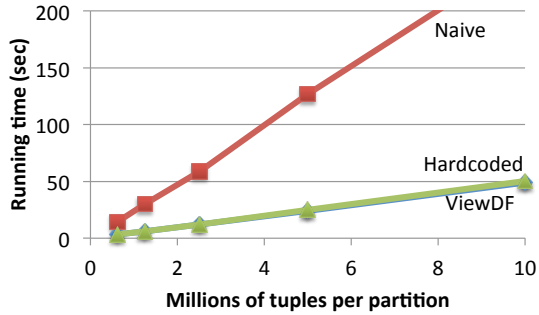

Figure 9: Running time vs. data set size for sliding window MAX executed using the Naive, Hardcoded and ViewDF approaches.

2. a hard-coded implementation that manually maintains the `Helper` view by sort-merging a new batch of data with the partial aggregates stored in the previous part of the `Helper` view;
3. the ViewDF approach corresponding to `Helper/View5` and `Helper/View6` from Section 5.2.

Below, we report two running-time experiments for each type of aggregate: scalability with the number of tuples per data batch (up to 10 million tuples) and scalability with the window size (up to 100 minutes, i.e., batches).

7.2.1. Distributive Aggregates (MAX)

For sliding window MAX, Figure 9 shows scalability with the data size (fixing the window size at ten) and Figure 10 shows scalability with the window size (at 10 million tuples per part). As the number of tuples per batch grows, all three methods take longer; however ViewDF and Hardcoded are significantly faster than Naive. Similarly, as the window size grows from 5 to 100 batches, Naive scales poorly, whereas ViewDF and Hardcoded are much faster and scale better. The performance of ViewDF and Hardcoded is virtually identical.

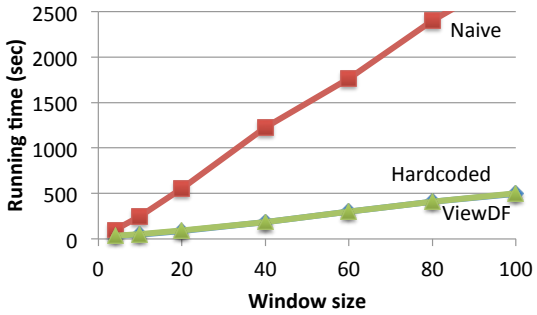

Figure 10: Running time vs. window size for sliding window MAX executed using the Naive, Hardcoded and ViewDF approaches.

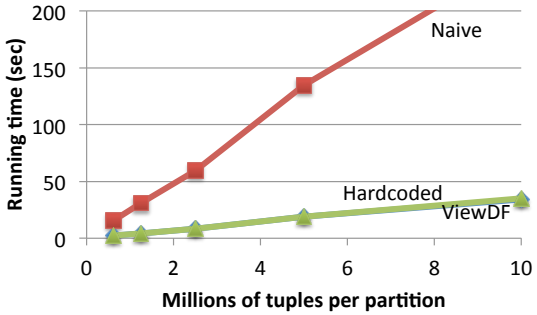

Figure 11: Running time vs. data set size for sliding window SUM executed using the Naive, Hardcoded and ViewDF approaches.

7.2.2. Subtractable Aggregates (SUM)

For sliding window SUM, Figure 11 illustrates scalability with the data size (again, fixing the window size at ten) and Figure 12 shows scalability with the window size (again, at 10 million tuples per part). The results are similar to those for MAX. Again, ViewDF (and Hardcoded) is significantly faster and scales better than Naive with the data size and window size; additionally, ViewDF and Hardcoded perform equally well.

Since subtractable aggregates are also distributive, we can incrementally compute SUM using the approach we employed for MAX (`Helper/View5`). Of course, we do not expect this to be as efficient as the specialized optimization for subtractable aggregates, but it should still be much more efficient than Naive. This was indeed the case in our experiments. Returning to Figure 11, maintaining the sliding window SUM using the strategy for MAX was roughly 50 percent less efficient than the strategy for subtractable aggregates; e.g., at 10 million tuples per part, it took 49 seconds compared to 34 for Hardcoded and ViewDF.

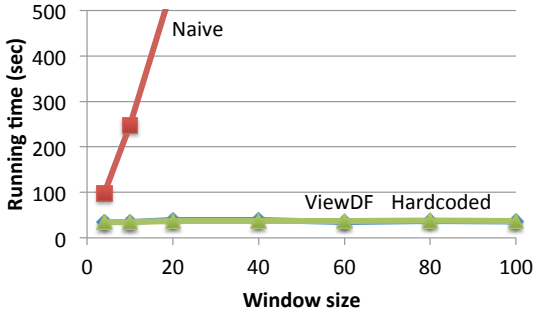

Figure 12: Running time vs. window size for sliding window SUM executed using the Naive, Hardcoded and ViewDF approaches.

8. Conclusions

In this paper, we introduced ViewDF, a framework for declaratively specifying how to incrementally update materialized views over append-only streaming data. ViewDF assumes that tables and views are partitioned by time, and leverages partition relationships to avoid scanning an entire source table when updating a view. In addition to letting users write SQL queries that directly specify how to update a view, we showed how to automatically converting event-processing queries and sliding window aggregates into incremental view maintenance expressions.

Several interesting directions for future work arise from this paper, including the following:

1. Automatic translation from view definition queries to ViewDFs for other classes of useful streaming queries besides event processing and window aggregation—for example, time series analytics, graph analytics and iterative machine-learning operations such as prediction model building and incremental clustering.
2. Cost-based selection of the most efficient incremental maintenance algorithm (ViewDF) when multiple options are available.
3. Applying multi-query optimization to ViewDFs so that similar views, such as pattern-matching queries looking for similar patterns, may be updated together.

9. Acknowledgements

This research was funded by the Natural Science and Engineering Research Council of Canada (NSERC).

10. References

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. SIGMOD 2008, 147-160.

[2] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBtoaster: higher-order delta processing for dynamic, frequently fresh views. PVLDB, 5(10):968–979, 2012.

[3] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. *Data Stream Management - Processing High-Speed Data Streams*, Springer, 2016, 317-336.

[4] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. VLDB 2004, 336-347.

[5] A. Baer, L. Golab, S. Ruehrup, M. Schiavone, and P. Casas. Cache-Oblivious Scheduling of Shared Workloads. ICDE 2015, 855-866.

[6] A. Baer, A. Finamore, P. Casas, L. Golab, and M. Mellia. Large-Scale Network Traffic Monitoring with DBStream, a System for Rolling Big Data Analysis. BigData 2014, 165-170.

[7] M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee. Moirae: History-Enhanced Monitoring. CIDR 2007, 375-386.

[8] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent Streaming Through Time: A Vision for Event Stream Processing. CIDR 2007, 363-374.

[9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin*, 38(4): 28-38, 2015.

[10] U. Cetintemel, D. J. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J.-H. Hwang, S. Madden, A. Maskey, A. Rasin, E. Ryvkina, M. Stonebraker, N. Tatbul, Y. Xing, S. Zdonik. The Aurora and Borealis Stream Processing Engines. *Data Stream Management - Processing High-Speed Data Streams*, Springer, 2016, 337-359.

[11] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. PVLDB 8(4): 401-412, 2014.

[12] B. Chandramouli, J. Goldstein, and S. Duan. Temporal analytics on big data for web advertising. ICDE 2012, 90-101.

[13] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295-405, 2012.

[14] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. NSDI 2010: 313-328

[15] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A General Purpose Event Monitoring System. CIDR 2007, 412-422.

[16] N. Dindar, B. Güç, P. Lau, A. Ozal, M. Soner, and N. Tatbul. Dejavu: declarative pattern matching over live and archived streams of events. SIGMOD 2009, 1023-1026.

[17] N. Folkert, A. Gupta, A. Witkowski, S. Subramanian, S. Bellamkonda, S. Shankar, T. Bozkaya, and L. Sheng. Optimizing refresh of a set of materialized views. VLDB 2005, 1043-1054.

[18] C. Ge, M. Kaufmann, L. Golab, P. M. Fischer and A. Goel. Indexing Bi-Temporal Windows. SSDBM 2015, 19.

[19] T. M. Ghanem, A. K. Elmagarmid, P. Larson, and W. G. Aref. Supporting views in data stream management systems. TODS 35(1), 2010.

[20] L. Golab and T. Johnson. Data stream warehousing. ICDE 2014, 1290-1293

[21] L. Golab, T. Johnson, J. Seidel, and V. Shkapenyuk. Stream Warehousing with DataDepot. SIGMOD 2009, 847-854.

[22] L. Golab, T. Johnson, S. Sen, and J. Yates. A sequence-oriented stream warehouse paradigm for network monitoring applications. PAM 2012, 53-63.

[23] L. Golab, T. Johnson, and V. Shkapenyuk. Scalable Scheduling of Updates in Streaming Data Warehouses. TKDE 24(6):1092-1105, 2012.

[24] L. Golab and M. T. Ozsu. Data Stream Management. *Synthesis Lectures on Data Management*. Morgan & Claypool Publishers, 2010.

[25] A. Gupta and I. Mumick. *Materialized views: techniques, implementations, and applications*. MIT press, 1999.

[26] T. Johnson and V. Shkapenyuk. Data Stream Warehousing in TidalRace. CIDR 2015, 4.

[27] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous Analytics Over Discontinuous Streams. SIGMOD 2010, 1081-1092.

[28] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: MapReduce-Style Processing of Fast Data. PVLDB 5(12): 1814-1825, 2012.

[29] A. Lerner and D. Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. VLDB 2003, 345-356.

[30] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.

[31] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. J. Shenoy. SCALLA: A Platform for Scalable One-Pass Analytics Using MapReduce. TODS 37(4): 27, 2012.

[32] E. Liarou, S. Idreos, S. Manegold, and M. L. Kersten. Enhanced stream processing in a DBMS kernel. EDBT 2013, 501-512.

[33] PipelineDB. https://www.pipelinedb.com/

[34] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making Views Self-Maintainable for Data Warehousing. PDIS 1996, 158-169.

[35] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer and M. Krishnaprasad. SRQL: Sorted Relational Query Language. SSDBM 1998, 84-95.

[36] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General Incremental Sliding-Window Aggregation. PVLDB 8(7): 702-713, 2015.

[37] H. Wang and C. Zaniolo. ATLaS: A Native Extension of SQL for Data Mining. SDM 2003, 130-141.

[38] A. Witkowski, S. Bellamkonda, H.-G. Li, V. Liang, L. Sheng, W. Smith, S. Subramanian, J. Terry, and T.-F. Yu. Continuous queries in Oracle. VLDB 2007, 1173-1184.

[39] E. Wu, Y. Diao, S. Rizvi. High-performance complex event processing over streams. SIGMOD 2006, 407-418.

[40] Y. Yang, L. Golab, and T. Ozsu. ViewDF: Declarative Incremental View Maintenance for Streaming Data. BIRTE 2015. Available at db.cs.pitt.edu/birte2015/files/paper5.pdf.

[41] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. HotCloud, 10, 2012.