# ViewDF: Declarative Incremental View Maintenance for Streaming Data

Yuke Yang, Lukasz Golab and M. Tamer Özsu

University of Waterloo, Canada
{y274yang,lgolab,tozsu}@uwaterloo.ca

**Abstract.** In this paper, we present ViewDF: a declarative framework for incremental maintenance of materialized views over append-only streaming data. The main component of the proposed framework is the View Delta Function (ViewDF), which declaratively specifies how to update a 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 selected 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 window aggregation.

## 1 Introduction

We study the problem of materialized view maintenance over append-only data streams. In our context, motivated by applications such as network and infrastructure monitoring [4, 13, 14, 16], a database system collects data feeds containing measurements such as sensors readings or location reports. Data feeds continuously arrive in batches, e.g., every minute or five minutes. Since raw data usually need to be transformed and processed before they are suitable for analysis, users and applications define materialized views containing aggregates, joins, or higher-level events and entities computed from individual records. In order to enable (nearly)real-time analytics, views must be maintained as new data arrive.

View maintenance—in particular incremental view maintenance (IVM)—is not a new problem [7, 15]. However, early work and recent developments (e.g., DBToaster [2]) focus on views with standard relational operators. In contrast, real-time analytics involve new operators such as sliding window aggregation and pattern matching [1, 6, 14]. Maintaining materialized views with these operators introduces new challenges. To address these challenges, ad-hoc algorithms have been proposed for sliding window aggregation (see, e.g., [3, 19, 21, 23]), and event processing systems have been proposed for pattern matching on streaming data (see, e.g., [1, 9, 10, 26]). In this paper, we propose ViewDF: a flexible and *declarative* framework for IVM over append-only data streams that generalizes existing techniques and enables new optimizations.

The main idea is simple. Rather than defining views using SQL queries inside CREATE MATERIALIZED VIEW statements, we declaratively specify how to update a view via a View Delta Function (ViewDF). An SQL-like declarative specification is desirable because a ViewDF can be optimized and executed by an underlying database system. To enable IVM, we partition tables and views by time, and allow ViewDFs to reference individual partitions of their source table(s) as well as previous partitions of the view itself. In addition to writing ViewDFs directly, we also want to automatically translate view definition queries to incremental ViewDFs whenever possible.

The contributions of this paper are as follows. First, 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 allows users to specify, using SQL, how a batch of new data affects the view. Second, we present an algorithm for automatically translating a useful class of streaming queries, namely event processing queries, into incremental ViewDFs. Third, we implemented a prototype ViewDF framework using PostgreSQL, and we experimentally show its effectiveness.

ViewDF is a flexible framework: as new IVM techniques are proposed for other types of stream and time series analytics, the corresponding query-to-ViewDF translations may be added to the proposed system. For example, in addition to event processing, we also implemented existing techniques for incremental maintenance of sliding window aggregates from [3, 19, 21, 23] as ViewDFs. Due to space constraints, we omit the details of these and focus on event processing queries in the rest of this paper.

## 2 Example and Solution Preview

We introduce the ViewDF approach using an example drawn from network and Internet monitoring, which has been a popular motivating application for data stream analytics [4, 13, 14, 16]. 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 and destination IP addresses, followed by the measurements taken at that time. 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 $S$. We divide $S$ into one-minute partitions, each corresponding to one batch of data. Let $S[i]$ be the $i$th partition of $S$ and let $S[i..j]$ denote the union of the $i$th through $j$th partitions for $i < j$. We assume that each partition is a separate logical table that can be accessed directly; for instance, $S[i]$ corresponds to a logical table named $S\_i$, and $S[i..j]$ corresponds to UNION ALL of $S\_i$ through $S\_j$.

We now define the partition subscripts. The idea is to divide the Unix timestamp of the partition by its time span. For instance, a partition 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), e.g., $S[23667840]$[1]. The next partition of $S$, storing data from January 1 2015 at 0:01 hours, is therefore $S[23667841]$. Note that the subscript of the oldest partition reflects the timestamp of the oldest batch of data, and is not necessarily zero unless we have been receiving data since Unix time zero.

[1] Similarly, for a table partitioned by hour, the partition subscript for January 1 2015 at 0:00 hours is 1420070400 divided by 3600 (the number of seconds in an hour), which is 394464.

Suppose we want to find the source-destination pairs that have reported high packet loss (say, at least ten) for at least four consecutive measurements (i.e., for at least four consecutive minutes). Additionally, for each such pair, we want to report the number of consecutive measurements with high loss and the sum of the packets lost 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 packet loss 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 S 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, packet loss) measurements for a particular source-destination pair: (10:00, 6), (10:01, 12), (10:02, 15), (10:03, 24), (10:04, 20), (10:05, 16), (10:06, 7). 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.

The contents of View1 change over time. As was the case with $S$, we split View1 into one-minute partitions. When a new $S[i]$ partition is created for a new batch of data, a corresponding View1$[i]$ partition is created to store the source-destination pairs reporting at least four consecutive high packet loss measurements as of that minute. If users are only interested in the latest results, old partitions of View1 may be deleted over time.

A naive way to update View1 when a new batch of packet loss measurements arrives is to re-run the pattern query on all of $S$—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 $S$ we could not compute the correct `ct` and `sum_loss` values for it. However, this will not only find those source-destination pairs which have reported at least four consecutive high-loss measurements as of the current time, but also all such patterns that happened in the past. Clearly, we need an incremental maintenance strategy.

In this paper, we advocate expressing the view update operation directly in a declarative SQL-like manner. Observe that View1 is append-only: when a new batch of data creates a new partition of $S$, a corresponding new partition of View1 is created, and other partitions of View1 do not change. ViewDF allows users to directly specify the contents of a new partition of a view, by referring to one or more partitions of the source table(s) as well as one or more previous partitions of the view itself.

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

The View Delta Function (ViewDF) for the Helper view is shown below. There are three main components:

1) The INITIALIZE statement contains a query that defines the initial partition (and, implicitly, the schema) of the view. Here, the first partition of Helper contains the source-destination pairs that reported over 10 lost packets at that time. When the ViewDF is initially submitted, the INITIALIZE query is executed with references to partitions resolved to their logical table names. The subscript $i$ is set to that of the newest partition of $S$ that currently exists, and the results are loaded into the corresponding Helper$[i]$ partition. This is the first non-empty partition of Helper.

2) The UPDATE statement contains a query that defines the contents of the $i$th view partition. Here, whenever a new $S[j]$ partition is created for a new batch of data, the UPDATE query is executed and the results are inserted into a new Helper$[j]$ partition. Since both $S$ and Helper are partitioned by minute, new partitions will be created for them at the same pace. The UPDATE query performs a left outer join of the new partition of $S$ with the *previous* partition of the Helper view, 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 is because we want to account for all the source-destination pairs currently appearing in $S[j]$, 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 partition (Helper is partitioned by 60 seconds, as is $S$).

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

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
```

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. ViewDF expresses view maintenance operations at a granularity of temporal partitions, which avoids scanning large tables; in the above example, rather than scanning all of $S$, updating the Helper view only requires access to two partitions: its previous partition and the new partition of $S$. 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 4.2, we will show how to automatically translate queries such as View1 into incremental ViewDFs.

## 3 Related Work

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, 7, 15, 25]. 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, 3, 9, 10, 19, 21, 23, 26], to the best of our knowledge ViewDF is the first general and declarative framework for incremental view maintenance over streaming data.

Specifically, 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., [22]. Again, previous work in this space addresses standard relational operators rather than streaming operators.

There has also been work that directly targets view maintenance over streams [12]. 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 [24], 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 materialized view one batch at a time. Furthermore, ATLaS does not use the notion of temporal partitioning, which ViewDF uses for efficient IVM.

Temporal partitioning is a common technique for storing append-only data in traditional and stream data warehouses [4, 11, 13, 16]. Partition relationships have been used in previous work for view maintenance; for example, to compute monthly aggregates, it suffices to scan only those partitions of the base table which contain data for that month. In ViewDF, we go a step further and allow view maintenance queries to reference partitions of source tables as well as previous partitions of the view itself. This enables new IVM strategies for new types of queries.

Finally, there has been recent work on incremental distributed computation; see, e.g., [5, 8, 18, 20]. 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

### 4.1 System Description

Figure 1 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, and runs ad-hoc queries and view update queries specified in the ViewDFs. The ViewDF layer is the top box.

Every streaming base table and view is horizontally partitioned by time; however, there may be some dimension tables that store slowly-changing data and are not partitioned. Data streams arrive in batches, e.g., every minute. To simplify the technical discussion, we assume that data arrive in batch order. For example, a one-minute data batch arriving at 10:00 is assumed to contain data with timestamps between 9:59:00 and 9:59:59. Solutions for handling out-of-order data have been discussed in, e.g., [16, 17].

We assume that there is a data-loading process which creates new base table partitions as new data batches arrive and triggers the ViewDF layer whenever a new partition has been added. Partitions are stored as separate logical tables (and perhaps also physical tables, but this depends on the underlying database system). 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 (which is typically the arrival frequency of the stream). We assume that there is a catalog table that maintains, for each base table and materialized view, its partition length and a list of its partitions. Old partitions may be deleted depending on the available space and data retention policies.

The ViewDF layer accepts ViewDF definitions directly, and also includes a translation layer that automatically converts user-supplied CREATE VIEW statements (for certain types of queries) into ViewDF expressions for incremental maintenance. In Section 4.2, we describe the translation algorithm for event processing queries.

The view maintainer is responsible for view updates as per the ViewDF expressions. As we explained in Section 2, a ViewDF expression specifies an INITIALIZE query, and UPDATE query and a partition length. The queries can be arbitrary SQL queries supported by the underlying database system, and they may contain partition references of the form Table$[partition]$ that are resolved to logical table names at runtime.

An INITIALIZE query can reference one or more source tables, and one or more of their partitions. At the time of initializing the view, we query the catalog table for the

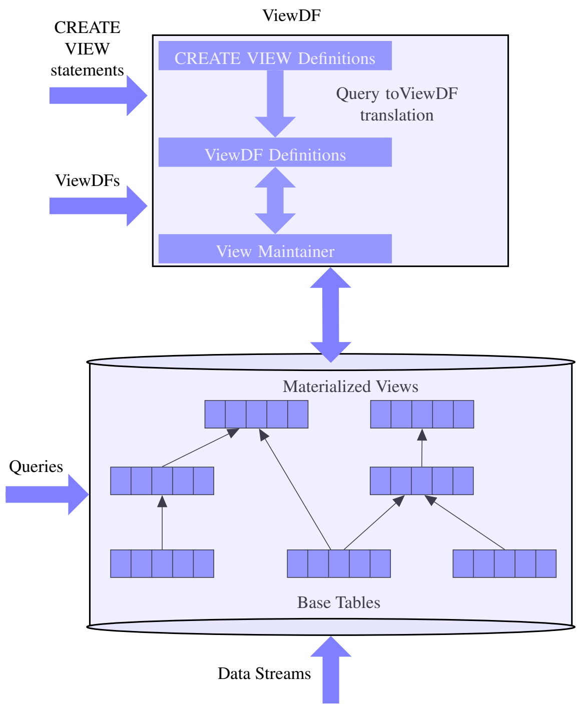

**Fig. 1.** ViewDF architecture.

newest partition of each source table and take the minimum of their subscripts[2]. This becomes the value of the partition subscript $i$, i.e., the view partition 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 $i$th partition of the view. Finally, we add a row for this view to the catalog table and add $i$ to the list of its partitions.

So far, we assumed that views have the same partition length and therefore the same refresh frequency as the source table(s). Below, we give an example of a view, labeled View3, partitioned by hour (3600 seconds) over the network measurement table $S$ 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 partition of View3, storing data for the first hour, requires the first 60 minutes of time, i.e., the first 60 partitions of $S$. Thus, View3$[1]$ requires $S[1..60]$, and, more generally, View3$[i]$ requires $S[i*60-59..i*60]$. At initialization time we take the largest partition subscript $i$ of $S$, divide it by 60 (which is the ratio of the partition length of View3 and $S$) and round down. This gives us the subscript for the first partition of View3.

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

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

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

Generalizing the above example, the view maintainer works as follows. For each view registered in the system, we maintain the partition subscript of the next partition that is to be created. For each view, we also maintain a bitmap with one entry for each source partition required to compute the next view partition. When a new base table or view partition is created, we scan through all the bitmaps and turn on all the bits corresponding to the new partition. Any view that has all of its bits set to one is scheduled for an update, and its UPDATE query is executed. Its next partition subscript is then incremented and its bitmap is recalculated.

To conclude the description of the ViewDF architecture, we explain how an ad-hoc query can access an old partition of a materialized view. Recall View2 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 View4_23667841, but this is cumbersome. Instead, in ViewDF we expose a PARTITION_TS attribute for each view. The above query can be written as:

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

## 4.2 Translating Event Processing Queries to ViewDFs

In this section, we present an algorithm for translating event-processing queries into ViewDFs that incrementally maintain histories of their results. The 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` from View1).
- The FROM clause can only include a single input stream, call it STREAM.
- The PATTERN clause may contain an arbitrary number of variables, including those with plus symbols (denoting one or more tuple).

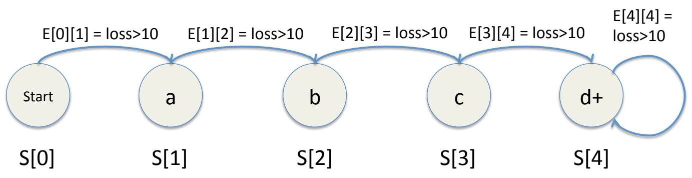

**Fig. 2.** FSM for View1.

- 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).

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.

**Query to FSM Conversion** A FSM $F = (S[], E[][])$ consists of a set of states $S$ and a set of directed edges and their state transition predicates $E$. Figure 2 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 7-18 by iterating through each WHERE predicate $pred$.

If $pred$ references a single pattern variable (state) $S[p]$, we add the corresponding transition predicate to the edge label $E[p-1][p]$ (line 9). For a KleeneClosure state, we additionally include the predicate in the self-edge $E[p][p]$ (lines 10-11). The *add* function on lines 9 and 11 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.

**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:         $E[p-1][p]$.add($pred$);
10:         if $S[p]$.isKleeneClosure then
11:             $E[p][p]$.add($pred$);
12:         end if
13:     end if
14:     if $pred$ references two states $S[p]$ and $S[p-1]$ then
15:         $E[p-1][p]$.add($pred$);
16:         $auxlist$.addAttr($pred$);
17:     end if
18: end for
```

Otherwise, if $pred$ references two pattern variables $S[p-1]$ and $S[p]$, e.g., $b.loss > a.loss$, then again we add the transition predicate to $E[p-1][p]$ (line 15). Here, the *add* function on line 15 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 will need to store some additional information in the Helper view, namely $a.loss$. In line 16, 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 2 (the edge labels are actually "$New.loss > 10$") and an empty $auxlist$ (there are no predicates referencing two pattern variables in View1).

**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 is different (more general) than the one shown in Section 2, which was a hand-crafted ViewDF for a simple pattern query.

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

**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 the SELECT clause of $Q$ do
 5:     Write(“New.” + $a$ + “,”);
 6: end for
 7: for all attributes $a$ in the 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 STREAM[i] AS New”);
16: Write(“WHERE ” + $E[0][1]$”);
17: Write(“UPDATE Helper[j] AS SELECT”);
18: for all non-aggregate attributes $a$ in the SELECT clause of $Q$ do
19:     Write(“New.” + $a$ + “,”);
20: end for
21: for all attributes $a$ in the 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 STREAM[j] AS New”);
37: Write(“LEFT OUTER JOIN Helper[j-1] AS Prev”);
38: Write(“WHERE state <> 0”);
39: Write(“PARTITION LENGTH $L$”);
```

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 View3). 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 STREAM[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 STREAM[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 14 and 17).

```
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
```

**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 the SELECT clause of $Q$ do
6:      Write($a$ + “,”);
7:  end for
8:  for all attributes $a$ in the 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] WHERE state=” + $n$”);
15: Write(“UPDATE V[j] AS”);
16: Repeat lines 4-13
17: Write(“FROM Helper[j] WHERE state=” + $n$”);
18: Write(“PARTITION LENGTH $L$”);
```

```
PARTITION LENGTH L
```

We conclude the discussion of translating pattern queries to ViewDFs with a simple optimization. The final view stores the results of the query and the Helper view is needed only to update it. Thus, old partitions of Helper may be deleted.

## 5 Experiments

We implemented the ViewDF framework, including the translation layer for event-processing queries, 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 reruns the pattern query whenever a batch of new data arrives;
2. a “hard-coded” approach in which the incremental maintenance logic from the Helper view is implemented as a sort-merge join in Java; the database is used only to store the output of the Helper and final view (and is accessed via JDBC);

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.

We generated a synthetic input stream that resembles the network monitoring stream from Section 2 and arrives in one-minute batches. We wrote a PL/PGSQL program to

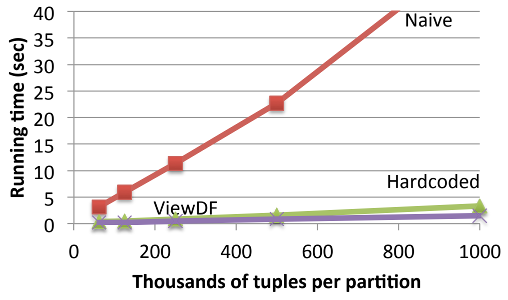

**Fig. 3.** Running time vs. data set size for Naive, Hardcoded and ViewDF.

create new partitions 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.

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 3 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 partition to one million and vary the $scope$ of the naive algorithm from 5 to 100. Results are shown in Figure 4. 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 (and, for the naive algorithm, we set scope to a very small value of 4). Results are shown in Figure 5. 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.

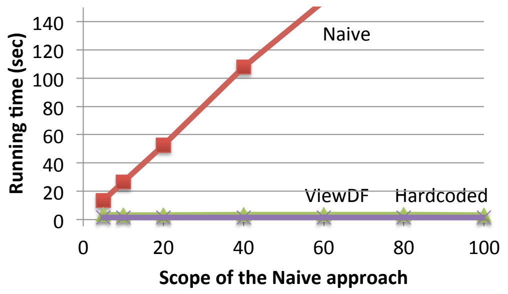

**Fig. 4.** Running time vs. scope for Naive, Hardcoded and ViewDF.

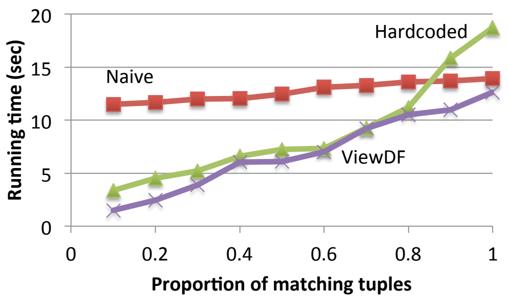

**Fig. 5.** Running time vs. proprotion of tuples having loss>10 for Naive, Hardcoded and ViewDF.

## 6 Conclusions

In this paper, we introduced and experimentally evaluated 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 proposed an algorithm for automatically converting event-processing queries into incremental view maintenance expressions. An interesting direction for future work is to develop automatic translations to ViewDFs for other classes of streaming queries—for example, time series analytics and iterative machine-learning operations such as prediction model building and incremental clustering. Another useful extension is to choose the most efficient ViewDF in a cost-based manner when multiple options are available.

## 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 and J. Widom. Resource sharing in continuous sliding-window aggregates. VLDB 2004, 336-347.
4. 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.
5. 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.
6. B. Chandramouli, J. Goldstein, and S. Duan. Temporal analytics on big data for web advertising. ICDE 2012, 90-101.
7. R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295-405, 2012.
8. T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. NSDI 2010: 313-328
9. 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.
10. 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.
11. 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.
12. T. M. Ghanem, A. K. Elmagarmid, P. Larson, and W. G. Aref. Supporting views in data stream management systems. TODS 35(1), 2010.
13. L. Golab, T. Johnson, J. Seidel, and V. Shkapenyuk. Stream Warehousing with DataDepot. SIGMOD 2009, 847-854.
14. L. Golab, T. Johnson, S. Sen, and J. Yates. A sequence-oriented stream warehouse paradigm for network monitoring applications. PAM 2012, 53-63.
15. A. Gupta and I. Mumick. *Materialized views: techniques, implementations, and applications*. MIT press, 1999.
16. T. Johnson and V. Shkapenyuk. Data Stream Warehousing in TidalRace. CIDR 2015, 4.
17. 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.
18. 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.
19. 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.
20. 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.
21. E. Liarou, S. Idreos, S. Manegold, and M. L. Kersten. Enhanced stream processing in a DBMS kernel. EDBT 2013, 501-512.
22. D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making Views Self-Maintainable for Data Warehousing. PDIS 1996, 158-169.
23. K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General Incremental Sliding-Window Aggregation. PVLDB 8(7): 702-713, 2015.
24. H. Wang and C. Zaniolo. ATLaS: A Native Extension of SQL for Data Mining. SDM 2003, 130-141.
25. 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.
26. E. Wu, Y. Diao, S. Rizvi. High-performance complex event processing over streams. SIGMOD 2006, 407-418.