

# 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

2 Traditional database management systems include On-Line Analytical Processing  
3 (OLAP) systems, focusing on deep analytics over read-only data, and On-Line Trans-  
4 actional Processing (OLTP) systems, optimized for frequent updates, insertions and  
5 deletions. Recently, new data management solutions have been proposed to handle the  
6 three Vs of big data: volume, velocity and variety. In particular, a new application  
7 area has appeared in response to high velocity: *append-only* data stream management  
8 [24]. Data streams naturally arise in many applications such as the Internet-of-Things  
9 (IoT), and include sensor measurements, GPS fixes, system logs, Web clicks, social  
10 media interactions, financial transactions and results of scientific experiments. In these  
11 applications, new data are continuously produced, while existing data such as past  
12 measurements are not modified.

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

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

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

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

39 View maintenance—in particular incremental view maintenance (IVM)—is not a  
40 new problem [13, 25]. However, early work and recent developments (e.g., DBToaster  
41 [2]) focus on views with standard relational operators, possibly including group-by  
42 aggregation and subqueries with semijoins or antijoins [38]. In contrast, stream ana-  
43 lytics involve new operators such as sliding window aggregation and pattern matching  
44 [1, 12, 22]. Incrementally maintaining materialized views with these operators intro-  
45 duces new challenges.

46 To address these challenges, various solutions have been proposed for sliding win-  
47 dow aggregation (see, e.g., [4, 30, 32, 36]), and event processing systems have been  
48 proposed for pattern matching on streaming data (see, e.g., [1, 15, 16, 39]). The solu-  
49 tion we present in this paper, named ViewDF, is a flexible and *declarative* framework  
50 for IVM over data streams that generalizes existing techniques and enables new opti-  
51 mizations for stream analytics.

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

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

63 tables containing data more than 60 minutes old.

64 The contributions of this paper are as follows.

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

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

## 82 2. Example and Solution Preview

83 We introduce the ViewDF approach using an example drawn from network mon-  
84 itoring, which has been a popular motivating application for data stream analytics  
85 [6, 21, 22]. Suppose we have a data stream containing quality-of-service measurements  
86 obtained from the network, such as packet loss between various source-destination  
87 pairs (measured by sending control packets and checking how many arrive at their des-  
88 tination). Each record contains a timestamp, the source (`src`) and destination (`dest`)  
89 IP addresses, followed by the measurements taken at that time. Suppose that every  
90 minute, a batch of new records arrives with one record for each source-destination pair  
91 being monitored, containing the number of lost packets for the given pair over the past  
92 minute.

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

99 We now explain the partition subscripts. The idea is to divide the Unix timestamp  
100 of the  $i$ th part by its time span. For instance, a part storing data from January 1 2015 at  
101 0:00 hours has the subscript of 1420070400 (the Unix timestamp at that time) divided  
102 by 60 (the number of seconds in one minute), i.e.,  $M[23667840]$ . The next part, storing  
103 data from January 1 2015 at 0:01 hours, is therefore  $M[23667841]$ . Similarly, for a  
104 table partitioned by hour, the subscript of the part starting at January 1 2015 at 0 hours

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

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

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

133 For example, consider the following sequence of timestamp and packet loss mea-  
134 surements for a particular source-destination pair. This pair is in the output at time  
135 10:04, with `ct=4` and `sum_loss=71`. It is also reported at 10:05 with `ct=5` and  
136 `sum_loss=87`. It is no longer reported at 10:06 because at that time, it has not pro-  
137 duced at least four consecutive measurements of at least 10 lost packets.

```
138 2015-01-01 10:00, 6
139 2015-01-01 10:01, 12
140 2015-01-01 10:02, 15
141 2015-01-01 10:03, 24
142 2015-01-01 10:04, 20
143 2015-01-01 10:05, 16
144 2015-01-01 10:06, 7
```

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

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

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

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

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

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

```
172 CREATE VIEW Helper AS
173 INITIALIZE Helper[i] AS
174   SELECT src, dest, 1 AS ct, loss AS sum_loss
175   FROM M[i]
176   WHERE loss>10
177 UPDATE Helper[j] AS
178   SELECT src, dest, ct+1, sum_loss+loss
179   FROM (
180     SELECT New.src AS src,
181           New.dest AS dest,
182           Prev.ct AS ct,
183           Prev.sum_loss AS sum_loss,
184           New.loss AS loss
185     FROM M[j] AS New
186     LEFT OUTER JOIN Helper[j-1] AS Prev
187     WHERE New.loss>10 )
188 PARTITION LENGTH 60
```

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

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

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

```
212 CREATE VIEW View2 AS
213 INITIALIZE View2[i] AS
214   SELECT src, dest, ct, sum_loss
215   FROM Helper[i]
216   WHERE ct >= 4
217 UPDATE View2[j] AS
218   SELECT src, dest, ct, sum_loss
219   FROM Helper[j]
220   WHERE ct >= 4
221 PARTITION LENGTH 60
```

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

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

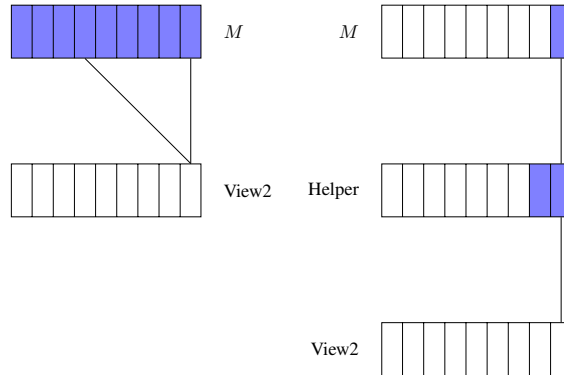


Figure 1: Computing a new part of View2 using a naive view maintenance approach (left) vs. a ViewDF incremental strategy (right).

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

### 240 3. Related Work

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

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

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

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

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

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

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

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

## 286 4. The ViewDF Framework

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

### 289 4.1. System Description

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

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

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



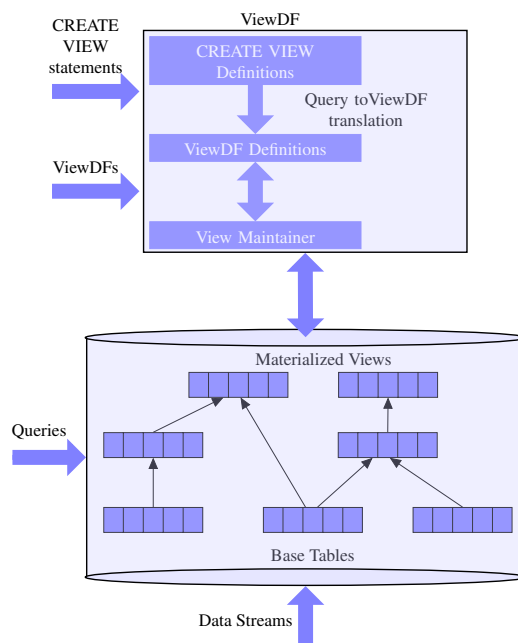


Figure 2: ViewDF architecture.

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

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

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

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

329 The queries can be arbitrary SQL queries supported by the underlying database sys-  
330 tem, and they may contain partition references of the form `Table[part]`. As mentioned  
331 earlier, partition references allow us to exploit the temporal locality of materialized  
332 views over streaming data: rather than scanning all the source tables (or views), it of-  
333 ten suffices to access a small set of parts when refreshing the results. For concreteness,  
334 assume that partition references resolve at runtime to logical table names of the form  
335 `Table_part`.

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

341 An `INITIALIZE` query can reference one or more source tables, and one or more  
342 of their parts. At the time of initializing the view, we query the catalog table for the  
343 subscript of the most recent part of each source table and we take the minimum of  
344 these<sup>1</sup>. This becomes the value of the partition subscript  $i$ , i.e., the part which will  
345 store the output of the `INITIALIZE` query. We then translate partition references to  
346 logical table names, run the query, and insert its output into the  $i$ th part of the view.  
347 Finally, we add a row for this view to the catalog table and add  $i$  to the list of its parts.

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

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

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

---

<sup>1</sup>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  $i = 394462$ .

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

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

#### 378 4.2. Additional Examples

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

```
390 CREATE VIEW View3 AS
391 INITIALIZE View3[i] AS
392     SELECT src, dest, sum(loss) as sum_loss
393     FROM M[i*60-59 .. i*60]
394     GROUP BY src, dest
395     ...
396 PARTITION LENGTH 3600
```

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

```
404 CREATE VIEW J AS
405 INITIALIZE J[i] AS
406     SELECT *
407     FROM M[i] NATURAL JOIN N[i]
408 UPDATE J[j] AS
409     SELECT *
410     FROM M[j] NATURAL JOIN N[j]
411 PARTITION LENGTH 60
```

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

415 as of January 1 2015 at 0:00 hours. We could divide the Unix timestamp of this date by  
416 60 and directly query the logical table `View2_23667841`, but this is cumbersome.  
417 Instead, in ViewDF we expose a `PART_TIMESTAMP` attribute for each view. The  
418 above query can be written as:

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

### 422 4.3. Dealing with Out-of-Order Data

423 Stream processing systems usually assume that the inputs are ordered by arrival  
424 time, in which case the notion of delayed or out-of-order data is not relevant. How-  
425 ever, some storage-enabled stream systems and traditional data warehouses support  
426 two timestamps: *system time*, denoting the time of arrival or insertion of the data, and  
427 *application time*, denoting when the given tuple was actually created or valid in real life  
428 [8, 18]. In this case, tuples may be late or out-of-order with respect to their application  
429 timestamps.

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

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

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

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

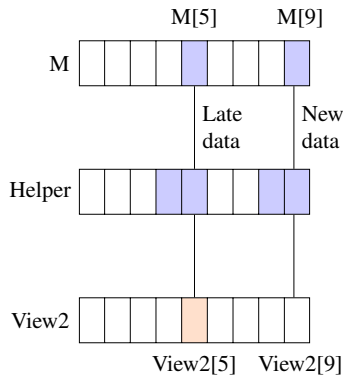


Figure 3: An example of late-arriving data and its consequences on materialized views.

## 458 5. Translating Queries to ViewDFs

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

### 464 5.1. Event Processing Queries

465 The event processing queries we support have the following format.

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

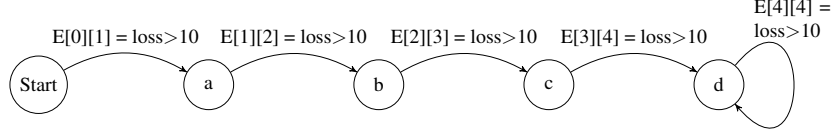


Figure 4: FSM for View1.

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

#### 485 5.1.1. Query to FSM Conversion

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

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

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

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

503 If *pred* references a single pattern variable (state)  $S[p]$ , then it can be one of two  
 504 transition predicates. For a KleeneClosure state, it is  $E[p][p]$ ; otherwise it is  $E[p -$   
 505  $1][p]$ . In either case, we add the predicate to the corresponding edge label (lines 10  
 506 and 12, respectively). The *add* function also performs some string editing: it replaces  
 507 a `PATTERN` variable with “New”; e.g., `a.loss > 10` becomes `New.loss > 10`. The  
 508 *add* function also adds “AND” between different predicates for the same edge. For  
 509 instance, if we have `a.loss > 10 AND a.loss < 20`, then  $E[0][1].add$  will be called  
 510 twice and will result in `New.loss > 10 AND New.loss < 20`. These edits simplify  
 511 the next step of converting the FSM into a ViewDF.

512 Otherwise, if *pred* references two pattern variables  $S[p-1]$  and  $S[p]$ , e.g., `b.loss >`  
 513 `a.loss`, then this is a transition predicate  $E[p-1][p]$ . In this case, the *add* function on  
 514 line 16 replaces the  $S[p-1]$  variable with “Prev” and the  $S[p]$  one with “New”; e.g.,  
 515 `b.loss > a.loss` becomes `New.loss > Prev.loss`. As before, an “AND” is  
 516 added if there are multiple predicates for this edge. Furthermore, to evaluate such a  
 517 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**

---

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

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

#### 524 5.1.2. FSM to ViewDF Conversion

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

529 First, we discuss the Helper view. The output of Algorithm 2 given *View1* and the  
530 corresponding FSM is shown below. Notice that the Helper ViewDF below is different  
531 (more general) than the one shown in Section 2, which was a hand-crafted ViewDF for  
532 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  $L$ ");

533

---

534       Here is how Algorithm 2 generated the above ViewDF for the Helper view. Lines 3-  
535 16 generate the INITIALIZE query. The SELECT statement, i.e., schema of the view,  
536 contains four parts: the non-aggregate attributes from the SELECT statement of the  
537 original query  $Q$  (lines 4-6), the attributes whose values we want to sum up (lines 7-9),  
538 a counter in case the original query includes count(\*) (lines 10-12), and the attributes  
539 in  $auxlist$  (line 13; there are none for View1). Notice that  $ct$  is initialized to one and  
540  $sum\_loss$  to the current value of loss. The Helper view also includes a  $state$  attribute



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

```

545 CREATE VIEW Helper AS
546 INITIALIZE Helper[i] AS
547   SELECT New.src AS src, New.dest AS dest,
548   New.loss AS sum_loss, 1 AS ct, 1 AS state
549   FROM InputStream[i] AS New WHERE New.loss > 10
550 UPDATE Helper[j] AS
551   SELECT New.src, New.dest,
552   Prev.sum_loss + New.loss, 1 + ct,
553   CASE WHEN state = 0 and New.loss > 10 THEN state + 1
554         WHEN state = 1 and New.loss > 10 THEN state + 1
555         WHEN state = 2 and New.loss > 10 THEN state + 1
556         WHEN state = 3 and New.loss > 10 THEN state + 1
557         WHEN state = 4 and New.loss > 10 THEN state
558   END,
559   FROM (
560     SELECT *
561     FROM InputStream[j] AS New
562     LEFT OUTER JOIN Helper[j-1] AS Prev)
563   WHERE state <> 0
564 PARTITION LENGTH L

```

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

580 Algorithm 3 generates the final view. Given the pattern matching query from  
 581 View1 as input, the final view is shown below and named View4. The INITIAL-  
 582 IZE and UPDATE queries are the same. Their SELECT statements include all the  
 583 non-aggregate attributes of the original query (lines 5-7), plus all the aggregates (lines  
 584 8-13). The main difference between this automatically-generated ViewDF and View2  
 585 from Section 2 is the WHERE predicate. Here, the WHERE predicate checks to see if  
 586 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  $L$ ");

---

```
588 CREATE VIEW View4 AS
589 INITIALIZE View4[i] AS
590   SELECT src, dest, ct, sum_loss
591   FROM Helper[i]
592   WHERE state=4
593 UPDATE View4[j] AS
594   SELECT src, dest, ct, sum_loss
595   FROM Helper[j]
596   WHERE state=4
597 PARTITION LENGTH L
```

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

### 605 5.2. Sliding Window Aggregates

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

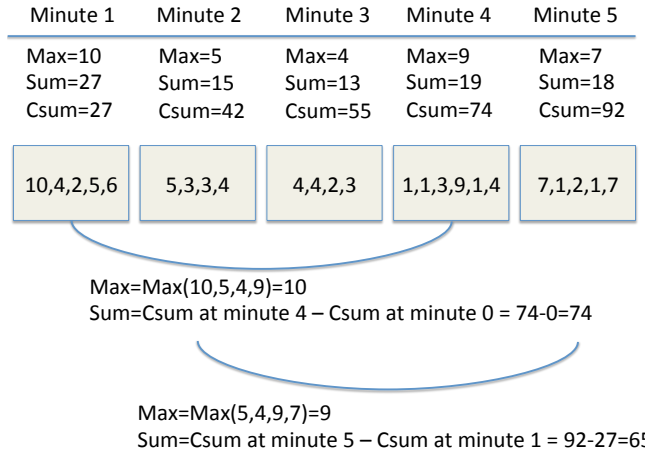


Figure 5: Computing distributive and subtractable aggregates over a sliding window.

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

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

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

### 633 5.2.1. ViewDF Translation for Distributive Aggregates

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

```

636 SELECT G, max(A)
637 FROM S [WINDOW W minutes]
638 GROUP BY G
639 HAVING H

```

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

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

```

655 CREATE VIEW Helper AS
656 INITIALIZE Helper[i] AS
657   SELECT G, max(A) as max_A
658   FROM S[i]
659   GROUP BY G
660 UPDATE Helper[j] AS
661   SELECT G, max(A)
662   FROM S[j]
663   GROUP BY G
664 PARTITION LENGTH 60

665 CREATE VIEW View5 AS
666 INITIALIZE View5[i] AS
667   SELECT G, max(max_A) as max_A
668   FROM Helper[i-W+1..i]
669   GROUP BY G
670   HAVING H
671 UPDATE View5[j] AS
672   SELECT G, max(max_A)
673   FROM Helper[j-W+1..j]
674   GROUP BY G
675   HAVING H
676 PARTITION LENGTH 60

```

### 677 5.2.2. ViewDF Translation for Subtractable Aggregates

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

680 stores the cumulative sum for each group, which is updated incrementally by combin-  
 681 ing the previous part of the Helper view with the new part of the input stream. The  
 682 final view, called View6 subtracts the cumulative sums stored in the  $(i - W)$ th part  
 683 of the Helper view from those in the  $i$ th part, and applies the HAVING predicate,  
 684 which now includes a  $\text{sum\_A} > 0$  condition. This removes groups that do not exist in  
 685 the current window—these groups have the same cumulative sum at the beginning and  
 686 end of the current window.

```

687 CREATE VIEW Helper AS
688 INITIALIZE Helper[i] AS
689   SELECT G, sum(A) AS sum_A
690   FROM S[i]
691   GROUP BY G
692 UPDATE Helper[j] AS
693   SELECT G, sum(Temp_Sum) AS sum_A
694   FROM (
695     SELECT G, sum(A) AS Temp_Sum
696     FROM S[j]
697     GROUP BY G
698     UNION ALL
699     SELECT G, sum_A AS Temp_Sum
700     FROM Helper[j-1]
701   ) AS Temp
702   GROUP BY G
703 PARTITION LENGTH 60

704 CREATE VIEW View6
705 INITIALIZE View6[i] AS
706   SELECT Temp.G, sum(Temp.Temp_Sum) AS sum_A
707   FROM (
708     SELECT G, sum_A AS Temp_Sum
709     FROM Helper[i]
710     UNION ALL
711     SELECT G, sum_A*(-1) AS Temp_Sum
712     FROM Helper[i-W]
713   ) AS Temp
714   GROUP BY G
715   HAVING sum_A>0 AND H

716 UPDATE VIEW View6[j] AS
717   SELECT Temp.G, sum(Temp.Temp_Sum) AS sum_A
718   FROM (
719     SELECT G, sum_A AS Temp_Sum
720     FROM Helper[j]
721     UNION ALL
722     SELECT G, sum_A*(-1) AS Temp_Sum
723     FROM Helper[j-W]
724   ) AS Temp
725   GROUP BY G
726   HAVING sum_A>0 AND H

```

727 PARTITION LENGTH 60

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

```
734 SELECT Temp.G, sum(Temp.Temp_Sum) AS sum_A
735 FROM (
736     SELECT G, sum_A AS Temp_Sum
737     FROM Helper[i]
738     UNION ALL
739     SELECT G, sum_A*(-1) AS Temp_Sum
740     FROM Helper[i-W+1]
741 ) AS Temp
742 GROUP BY G
743 HAVING sum_A=0
```

## 744 6. Discussion

745 In this section, we discuss the limitations of our approach and we develop a classi-  
746 fication of materialized views that fit the ViewDF framework.

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

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

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

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

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

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

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

779 Consider a stream of stock quotes, with each stream item composed of a times-  
780 tamp, the stock name, and the current price. Consider the following stream fragment,  
781 illustrating stock prices for the Acme company.

```
782 2015-01-01 10:58, Acme, $5.00  
783 2015-01-01 10:59, Acme, $4.95  
784 2015-01-01 11:00, Acme, $4.97  
785 2015-01-01 11:01, Acme, $4.98
```

786 Suppose we want to materialize a view which identifies the times when the price  
787 of a stock was higher than its *previous* price. In the above example, the records at time  
788 11:00 and 11:01 would be included (and perhaps the record at 10:58, depending on  
789 the previous price of Acme stock). Assume this query can be expressed in and evaluated  
790 by the underlying sequence query engine. Now suppose we write a corresponding  
791 ViewDF, partitioned by, say, hour, and specify in the UPDATE clause that the *j*th part  
792 (hour) of the view is to be computed over the *j*th part (hour) of the stream. This means  
793 that the first record of any hour will not have a previous record. Thus, the record at time  
794 11:00 would not be included in the output. This can be fixed by referring to two parts  
795 of the stream whenever updating the view: the current hour and the previous hour, but  
796 this comes at a cost of additional processing. Generalizing this example, we observe  
797 that sequence queries which may span partition boundaries may be implemented in  
798 ViewDF, but perhaps not optimally (unless we can specify Helper views to pre-compute  
799 the information required from other parts, as in our pattern matching example from  
800 Section 5.1).

## 801 7. Experiments

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

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

813 The purpose of the experiments is to show that 1) the ViewDF approach is signif-  
814 icantly faster than the naive approach, and 2) the ViewDF approach performs on par  
815 with the hard-coded approach, i.e., incurs little or no overhead. The experiments were  
816 performed on a workstation with an AMD Phenom II X4 955 3200 MHz processor and  
817 8 GB of RAM, running Ubuntu 12.10. We set the size of the shared memory used by  
818 the database server to 600MB.

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

### 825 7.1. Event Processing Queries

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

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

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

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

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

854 Now, we vary the proportion of tuples with loss measurements greater than ten from  
855 10 to 100 percent while keeping the number of tuples per batch fixed at one million



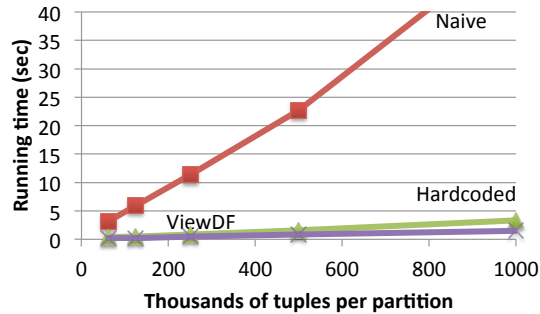


Figure 6: Running time vs. data set size for Naive, Hardcoded and ViewDF.

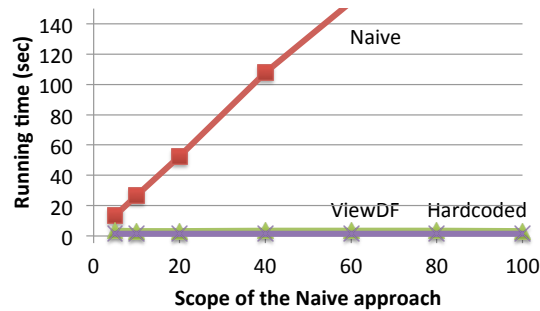


Figure 7: Running time vs. scope for Naive, Hardcoded and ViewDF.

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

## 865 7.2. Sliding Window Aggregates

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

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

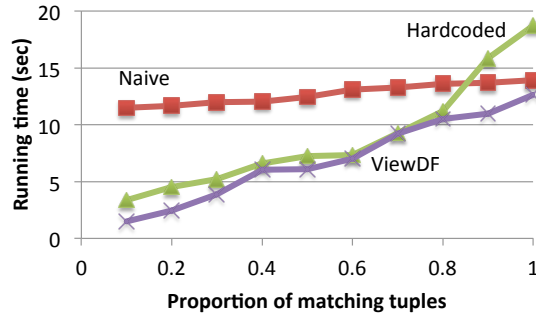


Figure 8: Running time vs. proportion of tuples having loss > 10 for Naive, Hardcoded and ViewDF.

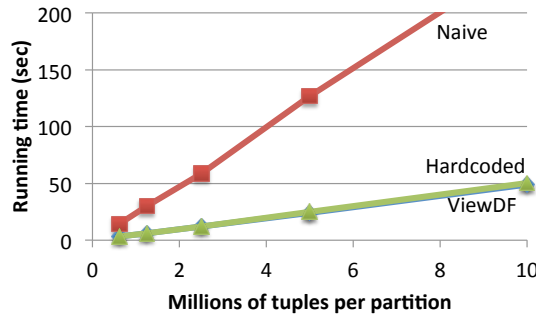


Figure 9: Running time vs. data set size for sliding window MAX executed using the Naive, Hardcoded and ViewDF approaches.

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

877 Below, we report two running-time experiments for each type of aggregate: scala-

878 bility with the number of tuples per data batch (up to 10 million tuples) and scalability

879 with the window size (up to 100 minutes, i.e., batches).

### 880 7.2.1. Distributive Aggregates (MAX)

881 For sliding window MAX, Figure 9 shows scalability with the data size (fixing

882 the window size at ten) and Figure 10 shows scalability with the window size (at 10

883 million tuples per part). As the number of tuples per batch grows, all three methods take

884 longer; however ViewDF and Hardcoded are significantly faster than Naive. Similarly,

885 as the window size grows from 5 to 100 batches, Naive scales poorly, whereas ViewDF

886 and Hardcoded are much faster and scale better. The performance of ViewDF and

887 Hardcoded is virtually identical.

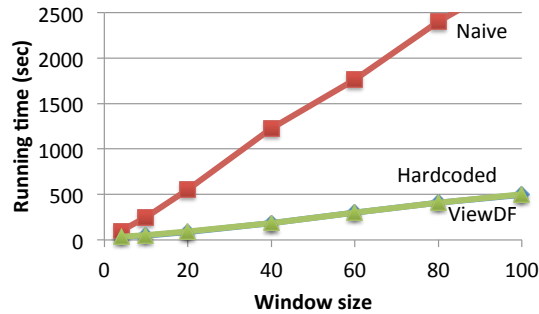


Figure 10: Running time vs. window size for sliding window MAX executed using the Naive, Hardcoded and ViewDF approaches.

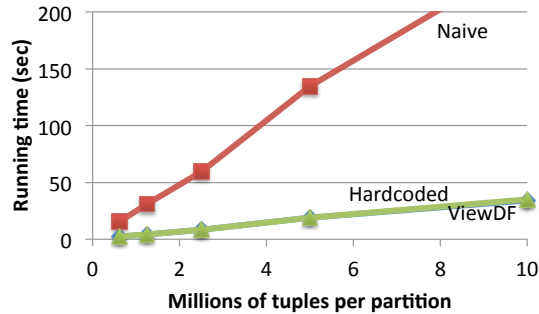


Figure 11: Running time vs. data set size for sliding window SUM executed using the Naive, Hardcoded and ViewDF approaches.

888 *7.2.2. Subtractable Aggregates (SUM)*

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

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

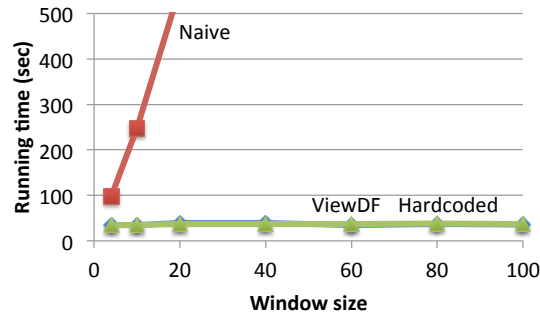


Figure 12: Running time vs. window size for sliding window SUM executed using the Naive, Hardcoded and ViewDF approaches.

## 902 8. Conclusions

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

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

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

## 920 9. Acknowledgements

921 This research was funded by the Natural Science and Engineering Research Coun-  
 922 cil of Canada (NSERC).

## 923 10. References

- 924 [1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching  
 925 over event streams. SIGMOD 2008, 147-160.

- 926 [2] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBtoaster: higher-order delta  
927 processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.
- 928 [3] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani,  
929 U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Manage-  
930 ment System. *Data Stream Management - Processing High-Speed Data Streams*,  
931 Springer, 2016, 317-336.
- 932 [4] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggre-  
933 gates. *VLDB 2004*, 336-347.
- 934 [5] A. Baer, L. Golab, S. Ruehrup, M. Schiavone, and P. Casas. Cache-Oblivious  
935 Scheduling of Shared Workloads. *ICDE 2015*, 855-866.
- 936 [6] A. Baer, A. Finamore, P. Casas, L. Golab, and M. Mellia. Large-Scale Network  
937 Traffic Monitoring with DBStream, a System for Rolling Big Data Analysis. *Big-*  
938 *Data 2014*, 165-170.
- 939 [7] M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee. Moirae: History-Enhanced  
940 Monitoring. *CIDR 2007*, 375-386.
- 941 [8] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent Streaming Through  
942 Time: A Vision for Event Stream Processing. *CIDR 2007*, 363-374.
- 943 [9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas. Apache  
944 Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering*  
945 *Bulletin*, 38(4): 28-38, 2015.
- 946 [10] U. Cetintemel, D. J. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cher-  
947 niack, J.-H. Hwang, S. Madden, A. Maskey, A. Rasin, E. Ryvkina, M. Stone-  
948 braker, N. Tatbul, Y. Xing, S. Zdonik. The Aurora and Borealis Stream Process-  
949 ing Engines. *Data Stream Management - Processing High-Speed Data Streams*,  
950 Springer, 2016, 337-359.
- 951 [11] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Ter-  
952 williger, and J. Wernsing. Trill: A High-Performance Incremental Query Pro-  
953 cessor for Diverse Analytics. *PVLDB* 8(4): 401-412, 2014.
- 954 [12] B. Chandramouli, J. Goldstein, and S. Duan. Temporal analytics on big data for  
955 web advertising. *ICDE 2012*, 90-101.
- 956 [13] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in*  
957 *Databases*, 4(4):295-405, 2012.
- 958 [14] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears.  
959 MapReduce Online. *NSDI 2010*: 313-328
- 960 [15] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White.  
961 Cayuga: A General Purpose Event Monitoring System. *CIDR 2007*, 412-422.

- 962 [16] N. Dindar, B. Güç, P. Lau, A. Ozal, M. Soner, and N. Tatbul. Dejavu: declarative  
963 pattern matching over live and archived streams of events. SIGMOD 2009, 1023-  
964 1026.
- 965 [17] N. Folkert, A. Gupta, A. Witkowski, S. Subramanian, S. Bellamkonda, S.  
966 Shankar, T. Bozkaya, and L. Sheng. Optimizing refresh of a set of materialized  
967 views. VLDB 2005, 1043-1054.
- 968 [18] C. Ge, M. Kaufmann, L. Golab, P. M. Fischer and A. Goel. Indexing Bi-Temporal  
969 Windows. SSDBM 2015, 19.
- 970 [19] T. M. Ghanem, A. K. Elmagarmid, P. Larson, and W. G. Aref. Supporting views  
971 in data stream management systems. TODS 35(1), 2010.
- 972 [20] L. Golab and T. Johnson. Data stream warehousing. ICDE 2014, 1290-1293
- 973 [21] L. Golab, T. Johnson, J. Seidel, and V. Shkapenyuk. Stream Warehousing with  
974 DataDepot. SIGMOD 2009, 847-854.
- 975 [22] L. Golab, T. Johnson, S. Sen, and J. Yates. A sequence-oriented stream warehouse  
976 paradigm for network monitoring applications. PAM 2012, 53-63.
- 977 [23] L. Golab, T. Johnson, and V. Shkapenyuk. Scalable Scheduling of Updates in  
978 Streaming Data Warehouses. TKDE 24(6):1092-1105, 2012.
- 979 [24] L. Golab and M. T. Ozsü. Data Stream Management. *Synthesis Lectures on Data  
980 Management*. Morgan & Claypool Publishers, 2010.
- 981 [25] A. Gupta and I. Mumick. *Materialized views: techniques, implementations, and  
982 applications*. MIT press, 1999.
- 983 [26] T. Johnson and V. Shkapenyuk. Data Stream Warehousing in TidalRace. CIDR  
984 2015, 4.
- 985 [27] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N.  
986 Thombre. Continuous Analytics Over Discontinuous Streams. SIGMOD 2010,  
987 1081-1092.
- 988 [28] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet:  
989 MapReduce-Style Processing of Fast Data. PVLDB 5(12): 1814-1825, 2012.
- 990 [29] A. Lerner and D. Shasha. AQuery: Query Language for Ordered Data, Optimiza-  
991 tion Techniques, and Experiments. VLDB 2003, 345-356.
- 992 [30] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain:  
993 efficient evaluation of sliding-window aggregates over data streams. *SIGMOD  
994 Record*, 34(1):39-44, 2005.
- 995 [31] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. J. Shenoy. SCALLA: A Platform  
996 for Scalable One-Pass Analytics Using MapReduce. TODS 37(4): 27, 2012.

- 997 [32] E. Liarou, S. Idreos, S. Manegold, and M. L. Kersten. Enhanced stream process-  
998 ing in a DBMS kernel. EDBT 2013, 501-512.
- 999 [33] PipelineDB. <https://www.pipelinedb.com/>
- 1000 [34] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making Views Self-  
1001 Maintainable for Data Warehousing. PDIS 1996, 158-169.
- 1002 [35] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer and M. Krish-  
1003 naprasad. SRQL: Sorted Relational Query Language. SSDBM 1998, 84-95.
- 1004 [36] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General Incremental  
1005 Sliding-Window Aggregation. PVLDB 8(7): 702-713, 2015.
- 1006 [37] H. Wang and C. Zaniolo. ATLaS: A Native Extension of SQL for Data Mining.  
1007 SDM 2003, 130-141.
- 1008 [38] A. Witkowski, S. Bellamkonda, H.-G. Li, V. Liang, L. Sheng, W. Smith, S. Sub-  
1009 ramanian, J. Terry, and T.-F. Yu. Continuous queries in Oracle. VLDB 2007,  
1010 1173-1184.
- 1011 [39] E. Wu, Y. Diao, S. Rizvi. High-performance complex event processing over  
1012 streams. SIGMOD 2006, 407-418.
- 1013 [40] Y. Yang, L. Golab, and T. Ozsü. ViewDF: Declarative Incremen-  
1014 tal View Maintenance for Streaming Data. BIRTE 2015. Available at  
1015 [db.cs.pitt.edu/birte2015/files/paper5.pdf](http://db.cs.pitt.edu/birte2015/files/paper5.pdf).
- 1016 [41] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized Streams: An  
1017 Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. Hot-  
1018 Cloud, 10, 2012.