ViewDF: a Flexible Framework for Incremental View Maintenance in Stream Data Warehouses

by

Yuke Yang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

Abstract

Because of the increasing data sizes and demands for low latency in modern data analysis, the traditional data warehousing technologies are greatly pushed beyond their limits. Several stream data warehouse (SDW) systems, which are warehouses that ingest append-only data feeds and support frequent refresh cycles, have been proposed including different methods to improve the responsiveness of the systems. Materialized views are critical in large-scale data warehouses due to their ability to speed up queries. Thus an SDW maintains layers of materialized views. Materialized view maintenance in SDW systems introduces new challenges. However, some of the existing SDW systems do not address the maintenance of views while others employ view maintenance techniques that are not efficient. This thesis presents ViewDF, a flexible framework for incremental maintenance of materialized views in SDW systems that generalizes existing techniques and enables new optimizations for views defined with operators that are common in stream analytics. We give a special view definition (ViewDF) to enhance the traditional way of creating views in SQL by being able to reference any partition of any table. We describe a prototype system based on this idea, which allows users to write ViewDFs directly and can automatically translate a broad class of queries into ViewDFs. Several optimizations are proposed and experiments show that our proposed system can improve view maintenance time by a factor of two or more in practical settings.

Acknowledgements

This is an opportunity for me to express my appreciation to those who have supported me during the master period. The first two persons I would like to thank deeply are my two supervisors, Dr. Tamer Ozsu and Dr. Lukasz Golab from whom I have learned a lot. They deserve a great deal of appreciation for their extensive support, both moral and financial. Their breadth and depth of knowledge, extraordinary patience, rich experience and hardworking spirit have always inspired me a lot. It is them that guided me to the right direction and overcome the obstacles that I met during the master research. In addition, they not only help me a lot at the academic level, but are also very helpful and understanding with my personal issues.

I would also like to thank Dr. Grant Weddell and Dr. Khuzaima Daudjee for spending their precious time on reviewing my thesis and for their comments.

A special thank goes to my boy friend, Huangdong Meng, for his support both on work and life. I would say without his support and his so much confidence in me, I cannot have such a happy and successful master period.

Finally I would like to thank my parents and my sister. Without their support and encouragement, I could not achieve success in my life and work.

Dedication

This is dedicated to the one I love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Traditional data warehouses are updated during downtimes, e.g., every night or on weekends. However, recently, many streaming data sources have emerged, such as sensor and RFID readings, financial transactions, network traffic traces, smart power grids and social media. Users and organizations are increasingly interested in doing data analytics over these types of data streams in nearly real time, with the goal of exploiting business opportunities or rapidly detecting system and infrastructure problems. These requirements are pushing traditional data warehousing technologies beyond their limits due to the increasing data volumes and demands for low latency in the stream-data analytics.

Stream processing has attracted attention for over a decade. In particular, some academic Data Stream Management Systems (DSMS) prototypes, such as Aurora [12], STREAM [10], as well as commercial DSMSs such as StreamBase [5] and StreamInsight [1], have been developed to process streaming data. DSMSes focus on processing data directly as it arrives without storing it first. As such, they offer high-performance and low-latency continuous processing, but they don't have access to history. However, there are many scenarios where historical information is required. As an example, in a network monitoring application, an administrator would receive alerts when there are continuous high packet loss for a very long time. In order to quickly determine the cause, the administrator may want to see the historical data to find the solutions for the similar alerts that occurred in the past. Another example is, in a stock market, a user may be interested in checking whether the incoming event, combined with the historical data, satisfies some pattern, e.g., a lasting rise or drop of price. In this scenario, the user needs to access the historical data to find the pattern.

A Stream Data Warehouse (SDW) combines traditional data warehouse and a DSMS to enable queries that seamlessly range from real-time processing and long-term historical data mining. Some academic and industrial SDW systems have been provided such as DataDepot [30], Moirae [13], TelegraphCQ [20], Truviso [27], Deja Vu [25] and so on. However, queries in SDW against a growing data archive quickly slow down, in turn slowing down real-time processing or producing results long after they are needed. Therefore, a direct problem for SDW is how to improve the responsiveness of the system as data archive grows, which is the topic addressed in this thesis.

1.2 Problem Definition

Materialized views are critical in large-scale data warehouses due to their ability to speed up queries. Thus an SDW maintains layers of complex materialized views to improve system's responsiveness. A materialized view[1] becomes out-of-date when the underlying base relations are modified. Since it is not possible to get up-to-date results if out-of-date materialized views are used, keeping the materialized views up-to-date becomes a critical problem. Although the current SDW systems employ many different techniques to improve the responsiveness of the system, and some of them consider view materialization [30, 13], they either do not address maintenance of materialized views [13], or maintain the materialized views in a very inefficient way [30].

This thesis explores using Incremental materialized View Maintenance (IVM) to do efficient materialized view maintenance in SDWs to keep the materialized views up-to-date, thereby improving the responsiveness of the system. This topic is important because of the following reasons:

- Although there has been much work on IVM for a number of years, the solutions are insufficient for today's stream applications. This is because most of the early IVM work, as well as recent developments, focus on views with relational operators [50, 7], while stream analytics involve operators beyond standard relational algebra, such as pattern matching [6, 19, 31] and sliding window aggregation [45]. In addition, stream analytics requires much larger data volumes and much higher data processing speed.

[1]Note that in the later thesis, whenever we say "view", we mean "materialized view".

- Although there have been some algorithms proposed in event processing systems and pub/sub systems to find a series of events that satisfy a specific pattern on the fly [6, 25], or to maintain aggregates over sliding windows (see, e.g., [9, 41]), these are stand-alone algorithms. In addition, they don't maintain the results as materialized views, which prevent revisiting the maintained results afterwards.

Consequently, a general SDW system for stream analytics is needed to maintain the materialized views in an incremental way. This thesis presents ViewDF, which is a flexible framework for IVM in SDWs that generalizes existing techniques and enables new optimizations for views defined with operators that are frequently used in stream analytics. The idea behind the proposed framework is as follows:

- A specific data model: We assume the inputs to our system are typically append-only data feeds whose schemas include a timestamp attribute. Each source table and materialized view is logically (and perhaps also physically) partitioned on a timestamp attribute, with partition sizes ranging from minutes to hours in practice. Source tables and views are refreshed at the granularity of partitions. Every time a new source partition arrives, the SDWs generate a new view partition.

- A special view definition (ViewDF) to enhance the traditional way of creating views in SQL: Given that views in an SDW evolve at a granularity of partitions, we should be able to declaratively specify the contents of a new partition using an SQL_like query, which we call ViewDF. A ViewDF[2] extends SQL by being able to reference any partition of any table, including a previous partition of the view itself. This extension, combined with the new data model, enables ViewDF to easily express IVM for views defined with operators beyond traditional relational algebra which we will explain in section 1.3.

1.3 Motivation: Sliding Window Aggregation

We motivate our approach with an example drawn from network monitoring [31]. Large ISPs usually periodically measure packet loss from each server to a set of destinations to monitor the network's situation so that they can quickly conduct troubleshooting if there's any trouble. Suppose we have a table S that collects measured packet loss between various

[2]Note that in the later part of the thesis, we use the term "ViewDF" for both the framework and the language.

Table 1.1: Partition of S at 9:00

timestamp	src	dest	loss
9:00	a	b	5
9:00	b	c	15

Table 1.2: Partition of S at 9:01

timestamp	src	dest	loss
9:01	a	b	9
9:01	b	c	12

source-destination pairs, with the schema: timestamp, src, dest, loss. Assume that new measurements arrive every minute and S is partitioned by minute, meaning that a new partition is created for every batch of new measurements every minute. Tables 1.1 and 1.2 show an example of source table S's two partitions at 9:00 and 9:01, assuming, for simplicity, there are two source-destination pairs in the ISP network.

Suppose we want to materialize a view ($View1$) containing the total packet loss for each source-destination pair over a sliding window of 60 minutes, updating the materialized view every minute to reflect the windowed total as of that minute. A naive way to update $View1$ when new packet loss measurements arrive is to access the most recent 60 minutes of data in S and recompute the total loss numbers for each source-destination pair in the whole window. A good optimization is to take the total packet loss computed over the previous window, add the loss from the current minute, and subtract the loss from the expired data from the previous window but outside the scope of the current window. In this way, we can avoid re-accessing and re-computing from all the partitions in the window. However, it is not possible to express this strategy in traditional SQL because the "create view" statement in the traditional SQL does not support referring to the view itself when defining the view. There are other optimizations for the sliding window aggregation proposed in [9, 41], which will be illustrated in Chapter 3.

However, the above optimization can be easily expressed in our proposed data model and ViewDF framework. In our specific data model, each source table and materialized view is partitioned by timestamp, and refreshed at a granularity of partitions. Therefore, in this example, every time a source partition arrives, we only need to generate a corresponding view partition instead of the whole materialized view. Specifically, in ViewDF framework, both S and $View1$ are partitioned by minute, with each partition of $View1$ containing the sliding window aggregates as of that minute. Let $View1[i]$ denote the ith partition of $View1$. Because ViewDF can reference any partition of any table, including a previous partition of the view itself, we can write ViewDF to express the above optimization by referring to the previous partition of $View1$ (denoted as $View1[i-1]$), and to the ith and $(i-60)$th partitions of S (denoted as $S[i]$ and $S[i-60]$). Note that $S[i-60]$ is the expired partition from the previous window that is outside the scope of the current window. In this way, we can easily express the above optimization by ViewDF. The core statements for the corresponding ViewDF are as follows.

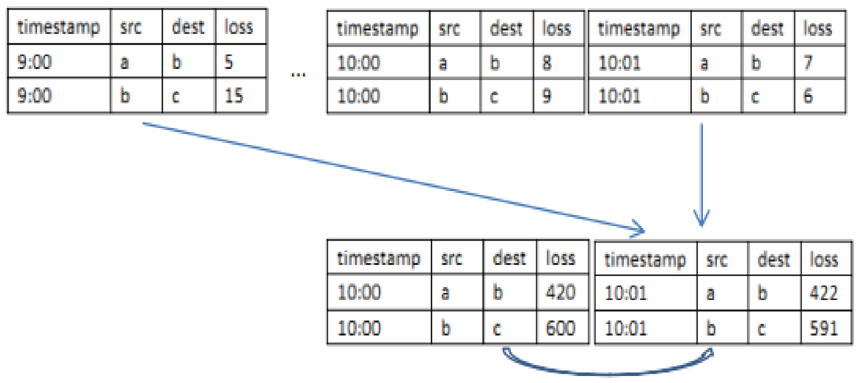
```
SELECT * FROM
(
   SELECT src,dest,sum(loss) as sum_loss FROM(
       SELECT src,dest,sum(loss) as loss FROM S[i] GROUP BY src,dest
       UNION ALL
       SELECT src,dest,sum(loss)*(-1) as loss FROM S[i-60] GROUP BY src,dest
       UNION ALL
       SELECT * from View1[i-1]) as X
   GROUP BY src,dest) as Y
WHERE sum_loss>0;
```

More details about ViewDF are discussed in Chapter 3. The reason that we add "where $sum\_loss > 0$" in the UPDATE statement of $View4$ is because we want to exclude the tuples that exist in the expired window, but not exist in the new window.

Figure 1.1 points out the data required to compute a new partition of $View1$ using the optimized approach, where the source partitions are at the top and the materialized view partitions are at the bottom. Each time a new source partition $S[i]$ arrives, we maintain a corresponding $View1[i]$ for the sum of the packet loss. When partition at timestamp 10:01 arrives, the new View1 partition is generated based on source partition at timestamp 9:00 which is the oldest partition in the last sliding window, source partition at timestamp 10:01 and the View1 partition at 10:00. For example, the first loss value in the new View1 partition at 10:01 is computed by: $420 + 7 - 5 = 422$. However, the naive way requires accessing all of the most recent 60 source partitions which is very inefficient.

1.4 Contributions and Organization

From the above motivating example about sliding window aggregation, we summarize ViewDF's advantages as follows:



timestamp	src	dest	loss
9:00	a	b	5
9:00	b	c	15

...

timestamp	src	dest	loss
10:00	a	b	8
10:00	b	c	9

timestamp	src	dest	loss
10:01	a	b	7
10:01	b	c	6

timestamp	src	dest	loss
10:00	a	b	420
10:00	b	c	600

timestamp	src	dest	loss
10:01	a	b	422
10:01	b	c	591

Figure 1.1: Illustration of the partitions required to create a new partition of View1 using an incremental strategy defined by a ViewDF.

1. Compared with the naive way of maintaining materialized views, ViewDF is very efficient and can achieve the IVM easily.
2. We give users freedom to express the materialized view maintenance by providing ViewDF, a language which is SQL-like and very easy for users to learn.

The specific contributions of this thesis are as follows:

1. We present a flexible framework for incremental maintenance of materialized views in SDW systems that generalizes existing techniques and enables new optimizations for views defined with operators that are common in stream analytics, including pattern matching and sliding window aggregation.
2. We give a special view definition (ViewDF) to enhance the traditional way of creating views in SQL. ViewDF is able to reference any partition of any table, including any partition of source tables, auxiliary tables and even the previous partitions of the view itself. In addition to allowing users to specify ViewDFs directly, we also develop algorithms to automatically translate a broad class of users' normal queries into ViewDFs that enable IVM.
3. We implement a prototype system for the proposed framework.
4. We experimentally evaluate the system, showing the speed-up of view maintenance by a factor of two or more as compared to existing approaches.

The remainder of this thesis is organized as follows. Chapter 2 is an overview of the related work. Chapter 3 presents an overview of the ViewDF framework. We talk about the system model and ViewDF specification in this chapter. Chapter 4 presents detailed algorithms for IVM in SDWs. We discuss the experiments and their results in Chapter 5. Finally, Chapter 6 presents our conclusions, as well as directions for future work.

Chapter 2

Related Work

In this chapter, we divide the related work into 4 parts: Incremental View Maintenance, Sequential Query Processing, Stream Data Warehouse, User Defined Functions (UDFs) and User Defined Aggregates (UDAs).

2.1 Incremental View Maintenance

A materialized view is one that is materialized by storing the tuples of the view in the database [36]. Materialized views provide fast access to data since the view is computed once and stored. Then, any query can access the stored results without recomputing the view. Materialized views have been widely used in query optimization, since answering queries using an existing view yields more efficient query execution plans.

A materialized view becomes out-of-date when the underlying base relations are modified. Hence, view maintenance is the process of updating the view in response to changes in the underlying relations. In most cases, it is wasteful to maintain a view by recomputing it from scratch [36]. Thus, it is usually less expensive to compute only changes in the view to update its materialization. Algorithms that compute changes to a view are called IVM algorithms. IVM has been studied extensively in the literature, e.g., [14, 46, 36, 47], for various view definition languages, e.g., Select-Project-Join (SPJ) views and views with grouping/aggregation, and for various types of updates, e.g., insertions, deletions, modifications to the database base tables. For a detailed survey related to the IVM, see [36].

Blakeley et al [14] propose a method in which all database updates to base tables are first filtered to remove from consideration those that cannot possibly affect the view. For the remaining database updates, a differential algorithm can be applied to compute the updated view by adding the changes instead of recomputing the materialized view from scratch. However, they only consider SPJ queries, which is very limited. Palpanas et al [46] consider views with aggregation and divided the behavior of an aggregate function into three categories: distributive (SUM, COUNT), algebraic (AVG) and holistic (MIN, MAX). While distributive and algebraic aggregation can be maintained incrementally based on base table and changes, holistic function needs to recompute the base tables to get the updated materialized views. A general IVM mechanism is proposed that applies to all aggregate functions, including both distributive and non-distributive over the insert, delete and modification operations. This work enhances the IVM framework with a selective recomputation step for non-distributive aggregation.

Ross et al [47] optimize IVM by providing a fine-grained approach to materialize the subqueries. Given a materialized view V, there are several possible views that can be additionally materialized and used for incremental maintenance of V. They formulate the problem of determining what additional views to materialize as an optimization problem, and develop an exhaustive memoing algorithm to solve the problem, using the expression DAG representation.

The above works only perform first-order maintenance and do not consider higher-order deltas. Ahmad et al [7] present the concept of higher-order IVM by means of support *viewlet transforms*, which is a recursive finite differencing technique applied to queries. The viewlet transform materializes a query and a set of its higher-order deltas as views. These views can support each other's incremental maintenance, leading to a reduced overall view maintenance cost. This can lead to that DBToaster can support applications ranging from algorithmic trading to scientific data analysis which require realtime analytics based on views over databases that change at very high rates. However, DBToaster only generalizes non-windowed SQL semantics, while the environment considered in this thesis requires IVM in windowed SQL semantics.

Witkowski et al [50] describe continuous queries (CQ) in Oracle DBMS, a feature that incorporates stream and complex event processing into a RDBMS, which is similar to our work in that it incorporates stream and sequential/temporal processing into a relational DBMS (RDBMS). They monitor real time changes to the query as the result of changes to its underlying tables and record the changes in auxiliary tables called *clogs*, in which they maintain the preimage, postimage and the operation that caused the changes. Their maintained clogs are similar to our maintained auxiliary tables. However, they only incrementally compute continuous join queries (CQJ), continuous queries with aggregations and window functions (CQW) while we also consider other sequence queries including pattern matching. In addition, they focus on transactional changes, which not only include insert, but also delete and update, while our work is focused on append-only data. Meanwhile, they focus on providing concurrency control support and not letting CQ refresh slow down the transaction traffic, which is different from our work.

Habich et al [38] exploit materialized views to speed up query processing in the presence of reporting functions. Given some existing materialized views and the user's query, through extending the relational query graph model and then rewriting query plans with the reporting function as well as aggregation queries, they show how and when to exploit the existing materialized views to make the query execution as fast as possible. They give a broad spectrum of derivation rules for different situations with reporting functions in materialized views, e.g., ordering reduction, partitioning reduction and aggregation queries. However, they focus on when and how to exploit materialized view, while ours focus on view maintenance.

In addition, there are other interesting works about materialized view maintenance in the temporal databases and the MapReduce framework. Yang et al [53] consider efficiently maintaining materialized views in a temporal database, in which they focus on temporal aggregate views. They introduce a new index structure called SB-tree, which incorporates features from both segment-trees and B-trees, to support fast lookup of aggregate results based on time. SB-tree can be maintained efficiently when the data changes. Chandramouli et al [19] propose a novel framework called TiMR that combines a time-oriented data processing system with a MapReduce framework, to perform temporal analytics. TiMR writes its own reducer code to maintain the necessary in-memory data structures to process the query, which is similar to the materialized view. No clear materialized view maintenance technique is described.

Our work performs IVM efficiently especially in the stream environment. Unlike [14], our work can support aggregation operations. Unlike [19], our work maintains materialized views in SDW for later query processing. In addition, unlike [7], our work can deal with windowed SQL semantics, as well as other sequence-oriented queries.

2.2 Sequence-Oriented Query Processing

Sequence-oriented query is a large class of queries that our system can support. It is difficult to express these queries using standard SQL. Golab et al [31] propose a generic extension to SQL to enable a new class of sequence-oriented queries to be easily expressed and optimized. There is much work on sequential operators in database systems, event processing systems, publish/subscribe systems and SDWs. In the following paragraphs, we focus on two sequential operators: pattern matching and sliding window operation.

2.2.1 Pattern Matching

Pattern matching is a sequential operator to identify the specific pattern in the gradually incoming data. For example, in the stock market, it is very useful to identify the increasing trend for a stock's price. How to express and support the search for complex sequential patterns has been explored in many projects. Sadri et al [48] introduce SQL-TS, an extension of SQL to process queries for identifying complex sequential patterns in database systems. Agrawal et al [6] provide an efficient pattern matching framework over event stream based on finite state machine (FSM). Demers et al [22] consider pattern matching in publish/subscribe systems based on FSM, and Dindar et al [25] develop an SDW in which they support pattern matching queries. Like our system's pattern matching queries, [6, 22, 25, 48] can let the user specify the states and state transition predicates in the pattern matching queries. In addition, all of them support Kleene Closure state and complex state transition predicates. However, the pattern language proposed in [48, 22] and [25] are SQL_like languages while [6] is based on a new language [51], which is shown to be richer than other languages [37]. We summarize these systems' difference for pattern matching in different aspects in Table 2.1 and we will elaborate different aspects in details in the following paragraphs.

Agrawal et al [6] can support four event selection strategies while others can only support two or three strategies. Event selection strategy addresses how to select the relevant events from an input stream mixing relevant and irrelevant events. We explain these using an example pattern query: suppose the input stream has schema (Company_name, price) and is partitioned by Company_name; we want to find a series of events from the input stream which can satisfy pattern (a,b+) with the state transition predicates: $a.price > 10$ and $b[i].price > b[i-1].price$. That is, we want to find a series of events where the beginning event's price larger than 10 and the later event's price is larger than the previous event's. In the following comparison, we use "a[Company_name](int price)" and "b[Company_name](int price)" to represent an event that satisfies the a or b matching part with the value in the '()' representing the value of the price and the value in '[]' representing the company name. If we don't list the '[Company_name]' part, it means that Company_name doesn't affect the final result.

1. Strict contiguity: In the most stringent event selection strategy, two contiguously selected events must also be contiguous in the input stream. For example, if a(11), b(5), b(6) is a series of events that matches the pattern (a,b+), the three events must also be consecutive in the input stream.

2. Partition contiguity: This is a relaxation of strict contiguity. In this selection

strategy, two contiguously selected events don't have to be contiguous in the input stream. If the events are conceptually partitioned based on a condition, the next relevant event must be contiguous to the previous one in the same partition. For example, although sequence "a[IBM](11), b[Dell](7), b[IBM](5), b[IBM](6)" doesn't satisfy strict contiguity because $b[IBM](5).price < b[Dell](7).price$, "a[IBM](11), b[IBM](5), b[IBM](6)" as a single partition is contiguous and can satisfy the matching under the partition contiguity selection strategy because b[Dell](7) belongs to the "Dell" partition and should not be affect "IBM" partition's matching results in this strategy.

3. Skip till next match: This is a further relaxation that completely removes the contiguity requirements: all irrelevant events will be skipped until the next relevant event is read. For example, although "a[IBM](11), b[Dell], b[IBM](5), b[IBM](4), b[IBM](6)" doesnt satisfy partition contiguity because $b[IBM](5).price > b[IBM](4).price$, it satisfies matching under this strategy, because we can skip b[Dell] and b[IBM](4) since they are not relevant events. After the skips, remaining events are all relevant.

4. Skip till any match: This strategy relaxes the previous one further, by allowing different actions on relevant events. For example, for sequence "a[IBM](11), b[Dell], b[IBM](5), b[IBM](4), b[IBM](6), b[IBM](7)", the skip till next match strategy must accept b[IBM](6) as this is the first matching event after b[IBM](4). However, skip till any match strategy can have two choices: accept b[IBM](6) or not. Therefore, this strategy introduces uncertainty in the matching process, and thus there can be more than one matching results from the same input stream.

Agrawal et al [6] achieves the skip till next match strategy by introducing an ignore edge on the corresponding state, it achieves the skip till any match by generating different series of events based on the uncertainty of the match. For example, for sequence "a[IBM](11), b[IBM](5), b[IBM](4), b[IBM](6)", we can generate at least two series: "a[IBM](11), b[IBM](5), b[IBM](6)" and "a[IBM](11), b[IBM](4), b[IBM](6)". Then [6] uses a buffer implementation to store these series and can finally construct these series, while other systems do not support constructing all the matching series of events.

However, a problem this approach has is that it only supports finding a sequence of events that match the specific pattern, doesn't let the users to select the specific attribute value or performing aggregation over the matching events. This is because there is no SELECT clause provided in its pattern matching language. In other words, a default "SELECT *" is assumed. From Table 2.1, we can see other systems support selecting different contents in the SELECT clause. For other systems, "first", "last" means the first and last occurrence of the Kleene Closure state, and "previous" means immediately preceding value. For aggregation, SQL_TS [48] and Golab et al [31] can support aggregation over each matching part, while others, except Agrawal et al [6] and Dindar et al [25], only support aggregation over the entire matching part. For example, for pattern (a+, b+), $a+$ is a matching part and $b+$ is another matching part. SQL_TS [48] and Golab et al [31] support aggregation over the entire pattern or part $a+$ or part $b+$. Deja Vu [25] provides a declarative pattern matching language to support getting aggregation values for the events that satisfy a single matching part, it does not consider supporting aggregation over the events that satisfy the entire matching query. Our work can easily support aggregation both over the entire matching events and over a single matching part by maintaining necessary intermediate variables in the Helper table[1]. For the comparison, see Table 2.1.

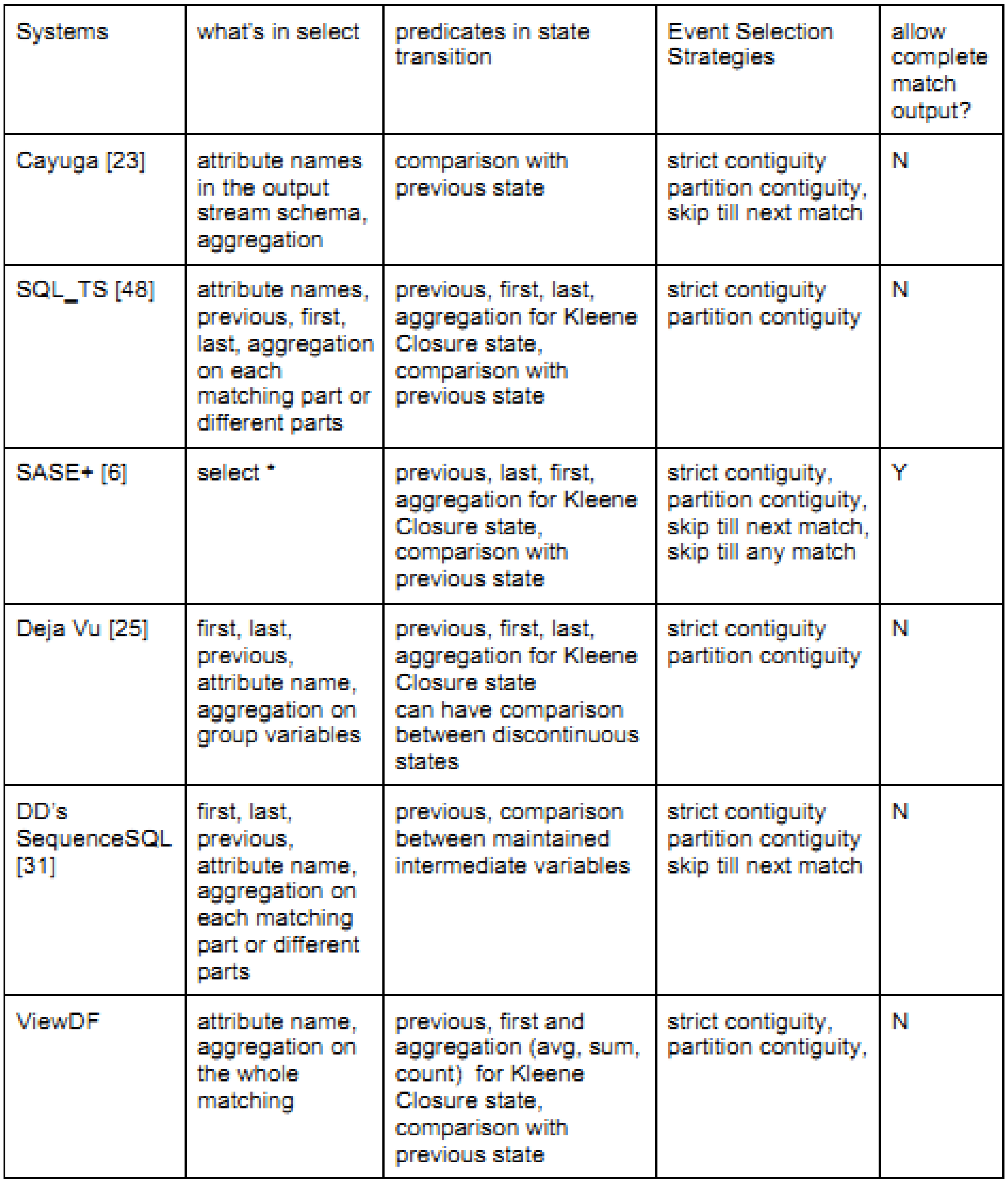
Agrawal et al [6] trade off a little space by creating a small data structure to maintain the computation state necessary for the pattern matching to improve pattern matching performance. The data structure includes version number of the run, current automaton state that the run is in, the start time of the run and other information. This is quite similar to our approach where we maintain auxiliary and materialized views to record the necessary intermediate values to improve the processing speed. However, since Agrawal et al [6] only address event systems, the structure is maintained only for necessary intermediate values, and once this is no longer necessary, it is abandoned. So, compared to materialized views in our approach, their maintained values cannot be used for future processing.

2.2.2 Sliding Window Operation

Sliding window operation is another non-relational operation that our system can support. There have been some work about maintaining views for sliding window operation in an efficient way. Arasu et al [9] and Chandrasekaran et al [20] propose several types of data structures, called *synopses*, to store enough states to efficiently re-compute various sliding window aggregates. We will use the same idea and discuss the *prefix synopses* and *interval synopses* in Section 4.2.

There are additional work on operator-level [41, 28] and plan-level [45]. For operator-level incremental processing, Jin et al [41] divide overlapping windows into disjoint panes, compute sub-aggregates over each pane, and "roll up" the pane-aggregates to compute window-aggregates. They explore the optimization of different types of aggregates, holistic

[1]In case that the Helper table may grow too big, for simplicity, we only implement aggregation over the entire pattern.



Systems	what's in select	predicates in state transition	Event Selection Strategies	allow complete match output?
Cayuga [23]	attribute names in the output stream schema, aggregation	comparison with previous state	strict contiguity partition contiguity, skip till next match	N
SQL_TS [48]	attribute names, previous, first, last, aggregation on each matching part or different parts	previous, first, last, aggregation for Kleene Closure state, comparison with previous state	strict contiguity partition contiguity	N
SASE+ [6]	select *	previous, last, first, aggregation for Kleene Closure state, comparison with previous state	strict contiguity, partition contiguity, skip till next match, skip till any match	Y
Deja Vu [25]	first, last, previous, attribute name, aggregation on group variables	previous, first, last, aggregation for Kleene Closure state can have comparison between discontinuous states	strict contiguity partition contiguity	N
DD's SequenceSQL [31]	first, last, previous, attribute name, aggregation on each matching part or different parts	previous, comparison between maintained intermediate variables	strict contiguity partition contiguity skip till next match	N
ViewDF	attribute name, aggregation on the whole matching	previous, first and aggregation (avg, sum, count) for Kleene Closure state, comparison with previous state	strict contiguity, partition contiguity,	N

Figure 2.1: Comparison between different pattern matching systems

or bounded. Ghanem et al [28] explore the semantics and implementation for the incremental evaluation of window operators, including window select, project, join, window aggregation and so on. Jin et al [41] and Ghanem et al [28] push the incremental logic all the way down to the operators, while, Liarou et al [45] design and develop the incremental logic at the query level, leaving the lower level intact and thus being able to reuse the complete storage and execution engine of a DBMS kernel. The latter first splits the input stream into n basic windows, processes each basic window separately, and finally merges partial results. Every time the window slides, they only need to process the new basic window's result and merge with the non-expired basic windows' results. While the concept of basic window is similar to the concept of pane [41], Liarou et al [45] need to rewrite the query plan including some modifications to the query optimizer in the DBMS kernel.

Although there is much work on non-relational operators, they are stand-alone algorithms while this thesis focuses on developing a general framework to incrementally materialized views maintenance on different kinds of queries.

2.3 Stream Data Warehouse

Several stream data warehouse systems have been developed in the research community. The earliest system is perhaps TelegraphCQ [20] that was originally designed as a DSMS. Instead of building the system from the scratch, TelegraphCQ implemented its Stream Processing Engine (SPE) as an extension to the PostgreSQL relational engine. In this regard, it provided a potential platform for tighter integration between relations and streams, and this direction was pursued even further after TelegraphCQ was commercialized into Truviso [27]. Truviso is a so-called "stream-relational" system that provides an integrated query processing approach that runs SQL queries continuously and incrementally over data before it gets stored in the database. Truviso supports queries over tables, streams, and their combinations, and as such, aims at efficiently serving continuous analytics applications.

There are a few other recent stream data warehouse systems that follow TelegraphCQ/ Truviso's design principle of building a streaming engine out of a relational database engine, but in slightly different ways. Moirae [13] incorporates Borealis [52] as SPE for continuous stream processing and PostgreSQL [4] as the underlying RDBMS for the storage of historical streams. As the queries against the growing historical data archive can quickly become slow and affect the whole system's response time, Moirae proposes to produce approximate results quickly, and, if necessary, additional more precise results incrementally. Specifically, in order to improve the responsiveness of the system, Moirae partitions the historical data into three types: present chunk, recent chunks and old chunks. The present chunk stays in memory while the recent and old chunks are stored on disk. Moirae guarantees that recent chunks have up-to-date indexes and materialized views by providing *recent event materialization*, which means that Moirae materializes recent chunks for all newly defined streams when a user submits a new query. The old chunks may be accompanied by some older indexes and materialized views. At runtime, Moirae processes chunks incrementally, prioritizing recent chunks over old chunks because any access to old chunks is much slower than access to more recent chunks. In addition, Moirae allows users to tell whether they want higher quality results or whether they are no longer interested in an event. Based on the user's preference, at runtime, Moirae decides how much resources to allocate and how much historical data to access. However, Moirae only addresses recent event materialization and there are no details about efficiently maintaining the materialized views, while this thesis is focused on efficient materialized view maintenance.

DataCell [44] extends the column-oriented MonetDB relational database for stream processing [15]. Like STREAM's "stream" data type, a new data type called "basket" is introduced in addition to relational tables. Stream tuples are accumulated in baskets and are accessed by continuous queries in a periodic fashion. Baskets allow batch, out-of-order, and shared processing. The general goal of this project is to explore how much the existing relational technology can be exploited for stream processing. As such, it has the potential to naturally integrate SPE functionality with DBMS functionality as part of its future work. Since DataCell is based on MonetDB, it is focused on column store organization while our system focuses on row store organization.

DejaVu [25] extends the MySQL relational database engine and exploits its pluggable storage engine API. Both streaming and historical data sources can be easily attached into a common query engine. Deja Vu provides *Live Stream Store* and *Archived Stream Store*, that work seamlessly to achieve both stream and archive data access. *Live Stream Store* is an in-memory store to accept push-based inputs. It essentially acts like a queue, providing live events into the query processing engine as they arrive. Live input events can also be fully or selectively materialized into an *Archived Stream Store*, which additionally provides features such as data compression and efficient access for historical pattern matching queries. Similar to Moirae [13], Deja Vu also faces the problem about system responsiveness. To address this problem, Deja Vu proposes recent event buffering and query result caching [24] for pattern correlation queries. For recent event buffering, it introduces the *Recent Buffer* that is an in-memory data structure that mediates between the live and archived event stores. By caching the most recent stream tuples, it provides the "hottest" subset of the archive with the same access costs and paths as for the stream data, thereby avoiding costly disk reads on recently archived data. Furthermore, it provides the means to perform bulk inserts into the archive event store. For query result caching, it is based on

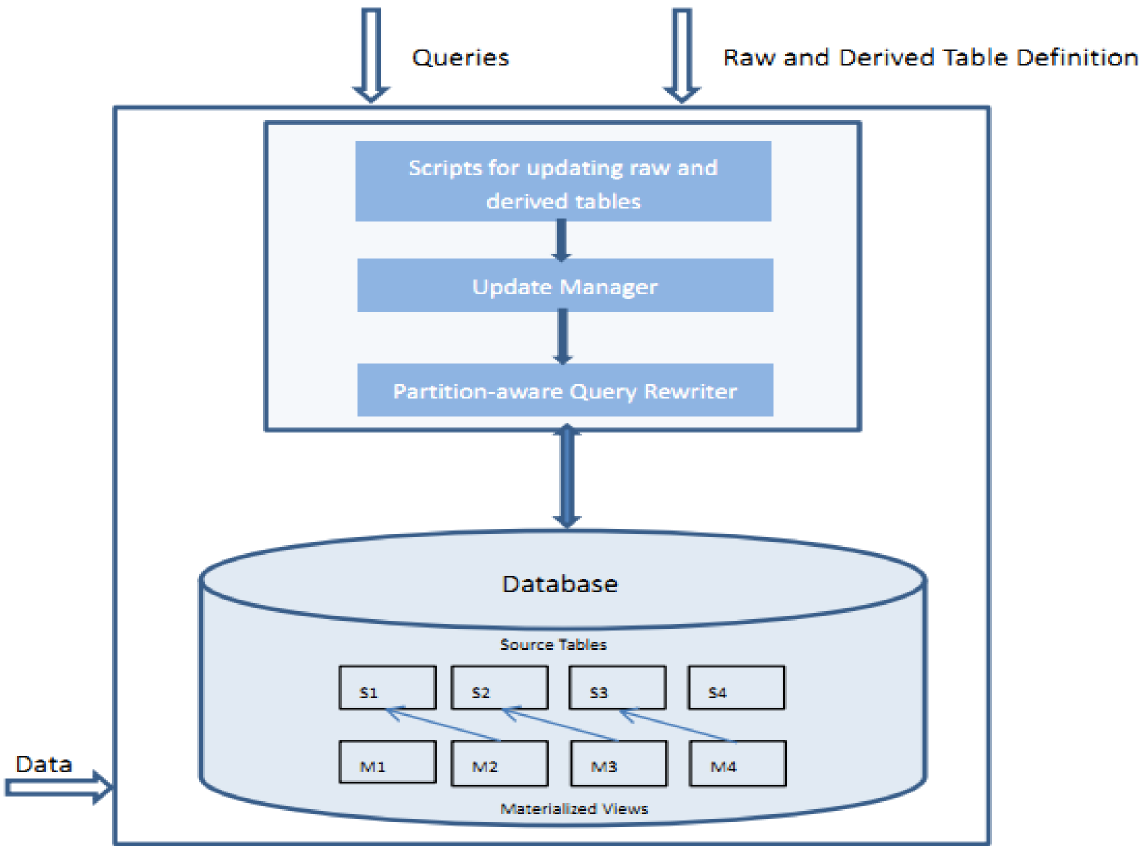

Queries
Raw and Derived Table Definition
Scripts for updating raw and derived tables
Update Manager
Partition-aware Query Rewriter
Database
Source Tables
S1
S2
S3
S4
M1
M2
M3
M4
Materialized Views
Data

Figure 2.2: DataDepot warehouse model

the observation that the recency region of a live match usually intersects with the recency regions of other live matches, so that an archive match could be used by multiple live matches. The query result caching resembles materialized views in our work. However, DejaVu only focuses on declarative pattern matching techniques over live and archived streams of events, while our work more generally consider sequential queries, temporal queries, relational queries and so on.

DataDepot [30] is an SDW whose architecture is depicted in Figure 2.2. It uses Daytona [35] as its underlying database. The user writes a set of configuration files containing the definitions of the raw and derived tables, and every time the source table changes, the derived tables need to be updated. The update manager is responsible to schedule the derived tables' updates with the scheduling algorithm in [32]. In DataDepot, both source and derived tables are horizontally partitioned on a timestamp attribute. As newer data

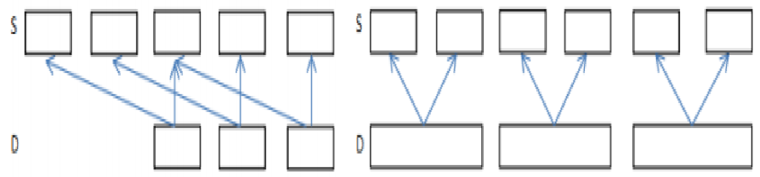


Figure 2.3: Examples for DataDepot Partition Dependencies

arrive, new partitions are created as is necessary to store the data. Similar to Moirae [13] and Deja Vu [25], DataDepot also requires efficient system responsiveness. In DataDepot, the responsiveness is mostly achieved by materialized views. To ensure efficient updates on materialized views, for each source table, DataDepot lets the user provide a lower bound and an upper bound of range of source partitions that affect data in a destination partition of the derived table. These bounds are expressed as source lower bound (SLB) and source upper bound (SUB) functions. Figure 2.3 provides two examples to explain SLB and SUB. In the first example, suppose the source table S has 5 1-hour partitions and the Derived table D is computed based on the three latest S's partitions. Therefore, SLB and SUB is expressed as $p - 2$ and p. In the second example, suppose S has 6 1-hour partitions and D has three 2-hour partitions. D's partition is computed from the relationship $SLB = 2p$ and $SUB = 2p+1$. Based on the user provided SLB and SUB, DataDepot avoids accessing the whole historical storage thus improves the system responsiveness. The importance of partition dependencies during view maintenance will be shown in the pattern matching example in Section 3.2.1, in which we will see that if the user don't provide the SLB, we have to recompute from the very beginning of the source partitions to get the materialized view. Two additional techniques are used to improve system responsiveness: variable partitioning and real-time scheduling. DataDepot uses variable partitioning to manage the historical table with the most recent portion of the table finely partitioned (say, 5 minutes) and the older part coarsely partitioned (say, one day). The second technique is a real-time scheduling algorithm [32] to minimize the weighted staleness (difference between the current time and the most recent data in a table) of a streaming warehouse.

2.4 UDFs and UDAs

Since our system is built on PostgreSQL, we explain PostgreSQL's UDFs in this section. PostgreSQL provides four kinds of UDFs to extend SQL's functionality: query language functions (written in SQL); procedural language functions (written in, for example, PL/PGSQL); internal functions, and C-language functions [4]. Compared to query language functions, procedural languages aren't built into the PostgreSQL server; they must be provided with loadable modules. In addition, PostgreSQL also provides UDAs. Aggregate functions in PostgreSQL are expressed in terms of state values and state transition functions. That is, an aggregation operates by using a state value that is updated as each successive input row is processed. To define a new aggregate function, one selects a data type for the state value, an initial value for the state, and a state transition function to express the state value's change as new rows are processed [4].

ATLaS is a language that improves the power and extensibility of traditional query languages in DBMS [49]. It aims at supporting efficient database-centric data mining applications and adds to SQL the ability to define new UDAs and Table Functions. This improves users' freedom to express their applications. ATLaS's UDAs are very similar to PostgreSQL's UDAs. ATLaS's UDAs uses a local table to maintain the state values, which is similar to PostgreSQL's state value. Each UDA consists of three blocks: INITIALIZE, ITERATE and TERMINATE, which is similar to PostgreSQL's initcond, sfunc and finalfunc. However, a big advantage of ATLaS is that its UDAs implement a stream-oriented computation model to accept a stream as input and produce a stream as output. It can support online aggregations while PostgreSQL's UDA can't. This make ATLaS suitable for online aggregations in stream applications. However, both PostgreSQL and ATLaS's UDAs are row-based (update state values as each successive input row is processed) while our system can support partition-based processing.

Chapter 3

Overview of ViewDF Framework

In this chapter, we give an overview of the ViewDF framework and system architecture. Figure 3.1 illustrates the components of the system. From Figure 3.1, we can see that the entire system is encapsulated as a box accepting data, queries and ViewDFs. The entire system consists of two large components: (1) ViewDF framework (2) Underlying database management system (DBMS). We will explain these components in the following sections.

3.1 Underlying DBMS

The whole system assumes an underlying DBMS, which answers queries and stores source tables and materialized views. The source tables are generated from the arriving data which is usually collected from append-only data feeds such as sensor, RFID, financial transactions or social media stream. We assume that the schema of each data feed includes a timestamp attribute and the data feeds arrive in order. For handling out-of-order data in stream warehouses, we can easily incorporate the solutions from [43, 40].

All of the source tables and materialized views are physically (or logically) partitioned using the timestamp attribute from the data feeds. We require that the data arrives at the system in the form of batch insertions, e.g., once a whole partition of tuples, which is different from the event systems' stream of events one by one. The partition lengths of source tables typically correspond to the data feed arrival frequencies, which in practice range from one or several minutes to several hours. We assume new data arrive periodically and existing data do not change in the future. Every time a whole partition of tuples arrives, the underlying DBMS creates a source partition and load the data in the new batch into

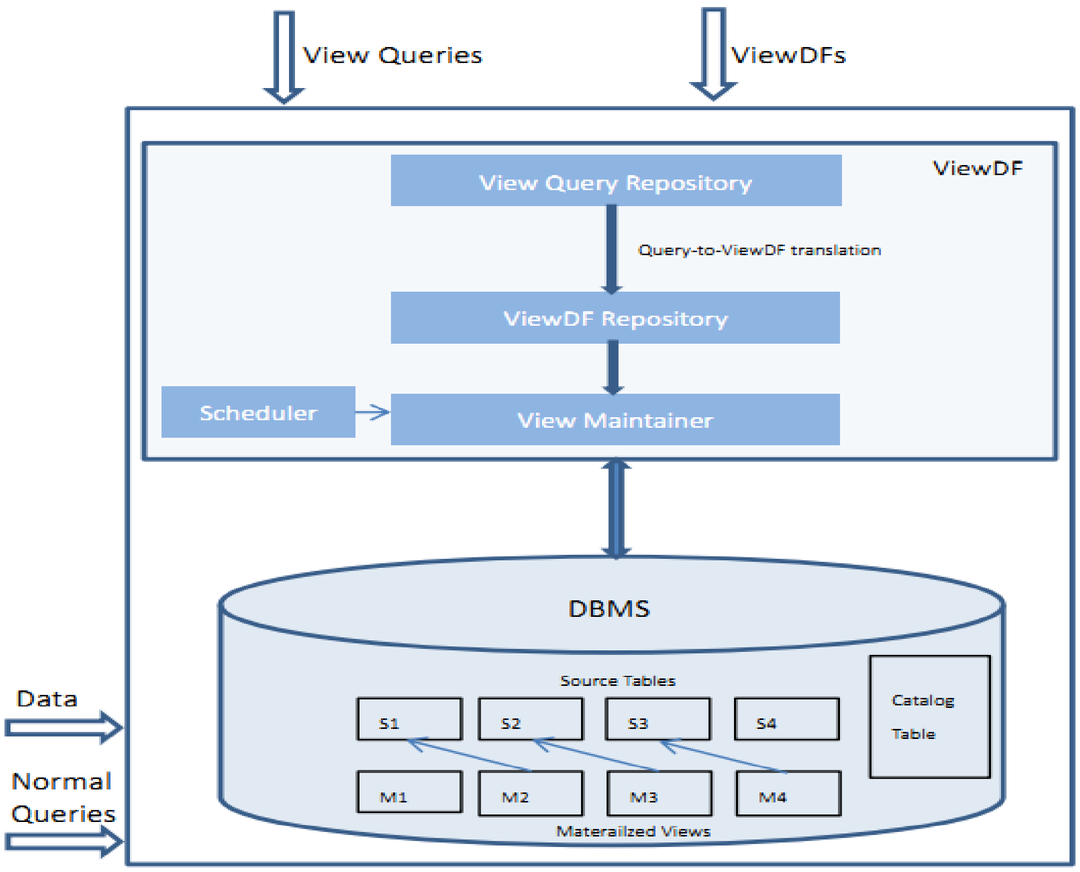


Figure 3.1: Overview of the ViewDF system

it. There are no deletions and modifications on the source partitions. The creation of one new partition (or several new partitions) of a source table will trigger the generation of a new partition of materialized view, which are handled by ViewDF framework, as we will discuss shortly. In Figure 3.1, we can see there is a relationship between the partitions of source table and materialized view. Every time a new batch of data arrives, a new source partition, say $S4$, will be generated, which in turn will trigger the generation of $M4$.

In addition, we also maintain a CATALOG table to record the name of the source tables and materialized views, their first partition's timestamp, the previous partition's subscript. The reason that we maintain the catalog table will be discussed shortly.

3.2 ViewDF Framework

As shown in Figure 3.1, ViewDF is a view specification and maintenance framework. Specifically, it contains three small components: (1) View query repository, (2) ViewDF repository, and (3) View maintainer. Users can submit specified views into the system in two ways: as view queries[1] or directly as ViewDFs that specify how to compute new partitions in an incremental way. The queries are stored in the View query repository and the ViewDFs are stored in the ViewDF repository. In addition to allowing users to specify ViewDFs directly, our system can also automatically translate a broad class of view queries into ViewDFs that enable efficient IVM. Therefore, a crucial component of the framework is the Query-to-ViewDF translator, which automatically generates ViewDFs that encode incremental view maintenance techniques based on view queries. Details about Query-to-ViewDF translator will be given in Chapter 4. In the following subsections, we will illustrate two view specifications (view queries and ViewDFs) and the function of view maintainer.

3.2.1 View Queries

This section discusses queries whose results will be automatically maintained as views. We elaborate view queries for sliding window aggregation and pattern matching in this subsection.

[1]Note that view queries are different from normal queries. View queries need to be pre-processed by ViewDF framework and then executed by the underlying DBMS, while normal queries are directly executed by the underlying DBMS.

(1) Sliding window aggregation: The user's query for the sliding window aggregation in the motivating example is as follows.

```
CREATE VIEW View1 AS
  SELECT src, dest, sum(loss) as sum_loss
  FROM S [WINDOW 60 minutes]
  GROUP BY src, dest
```

We assume an SQL-like syntax extended with a WINDOW clause, which defines the length of the sliding window as 60 minutes. The WINDOW clause is essentially a shorthand for two timestamp predicates that define the time range of data of interest.

(2) Pattern matching: We provide an example to explain the syntax for pattern matching queries. In the example, we use the same source partitions as in the motivating example. Suppose we want to maintain a materialized view over S, call it $View2$, that contains, for any given minute, all the source-destination pairs that have reported a high loss value (say, at least ten) for at least four consecutive measurements (at least four consecutive minutes if the source partition is generated once a minute). Additionally, we want to report the number of consecutive measurements with high loss and the total loss during this interval in the materialized view. We can view this example query as a patten matching query and the pattern the query wants to look for is: a high loss value for at least four consecutive measurements.

The query is shown below, assuming a syntax similar to recently-proposed event processing languages SASE+ [6]. The syntax of the query is as follows. In the first line, the query specifies the view name, $View2$. The second and the third line are the SELECT clause and FROM clause that have the same syntax as traditional SQL. In the fourth line, the PATTERN clause specifies a sequence pattern, "a,b,c,d+" in the example query for "at least four consecutive measurements". In the PATTERN clause, $d+$ indicates a Kleene plus operator to represent one or more consecutive occurrences and a, b, c represent one occurrence. The WHERE clause uses the variables in the PATTERN clause to specify predicates on individual variable as well as across multiple variables. Finally, the pattern query can also include other statements from the traditional SQL, such as GROUP BY and ORDER BY. We will provide more details about the query in Chapter 4.

```
CREATE VIEW View2 AS
  SELECT timestamp, src, dest, count(*) as count, 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[1].loss > 10
AND d[i].loss > 10
GROUP BY src, dest
```

Specifically, our system specifies the requirements on SELECT and WHERE clauses for the language as follows:

1. In the WHERE clause:

   (a) We allow the user to specify predicates on a single variable as well as across multiple variables. A predicate on a single variable only contains one variable from the SELECT clause, for example, $a.loss > 10$ in the example query. A predicate across multiple variables can contain multiple variables from the SELECT clause, for example, $b.loss > a.loss$. For simplicity, we only support two consecutive variables' comparison with the variable appears later in the PATTERN clause (b) on the left side and the earlier variable (a) on the right side.

   (b) For Kleene Closure (The variable with a Kleene plus operator, $d+$ in the example query), we support getting the first and ith occurrence of the Kleene Closure state by attaching the state name with "[1]" and "[i]" ($d[1]$ and $d[i]$ in the example). We also support getting the previous occurrence of the Kleene Closure state by attaching the state with "[i-1]" ($d[i-1]$ for example). In addition, we support getting aggregation (we only support average, sum and count) on the Kleene Closure state. For example, the user can specify the predicate "$d[i] > avg(d[..i])$" which means the ith occurrence of d state must be larger than the average of the previous occurrences of d state where "d[..i]" means all the previous occurrences of d state.

2. In the SELECT clause:

   (a) We support all the things that SQL can include in the SELECT clause.

   (b) We do not support getting the first, ith and previous occurrence that is supported in the WHERE clause for the Kleene Closure state.

(c) We only support getting aggregation (only sum, count, average) on the events that match the whole pattern. We don't support getting aggregation on each matching part. For example, assume sequence "a,b,c,d1,d2" is a sequence that satisfies the example pattern matching query, we support getting aggregation for the whole sequence but we don't support getting aggregation on the subsequence "d1,d2" which matches only the *d* part.

3.2.2 ViewDFs

The users can also submit view specification in ViewDFs that specify how to compute new partitions in an incremental way. The following is the ViewDF for the query in the motivating example. There are three main components in the ViewDF query: the INITIALIZE statement, which defines the first partition of the view, the UPDATE statement, which defines the content of the ith partition of the view, and the PARTITION BY statement, which specifies the partition length. The partition length is determined by the data arrival rate. Note that the CREATE VIEW statement in ViewDF is different from the traditional CREATE VIEW statement in the DBMS. The traditional CREATE VIEW statement defines a view of a query with the view not physically materialized, while the CREATE VIEW statement in ViewDF create materialized views. For the following ViewDF, the data arrives in a whole partition every 1 minute. The reason that we add "where $sum\_loss > 0$" in the UPDATE statement of $View1$ is because we want to exclude the tuples that exist in the expired window, but not in the new window.

```
CREATE VIEW View1 AS
INITIALIZE View1[k] AS
    SELECT src, dest, sum(loss) as sum_loss
    FROM S [k-60 .. k]
    GROUP BY src, dest;
UPDATE View1[i] AS
    SELECT * FROM
    (
       SELECT src,dest,sum(loss) as sum_loss FROM(
           SELECT src,dest,sum(loss) as loss FROM S[i] GROUP BY src,dest
           UNION ALL
           SELECT src,dest,sum(loss)*(-1) as loss FROM S[i-60]
           GROUP BY src,dest
           UNION ALL
```

```
            SELECT * from View1[i-1]) as X
        GROUP BY src,dest) as Y
    WHERE sum_loss>0;
PARTITION BY 1 minute
```

The generation of new partitions of source tables will trigger the generation of new partitions of materialized views. As there can be several concurrent generations of new partitions, a scheduler is needed to limit the number of concurrent updates and determine which table should be scheduled next. Our system can easily integrate the scheduler designed in [32]. When a new partition of view is scheduled to be created, the view maintainer component runs the SQL statements from the corresponding ViewDF against the database. Specifically, we will check whether this is the first time to generate $View1$ partition. If it is, the view maintainer will execute the INITIALIZE statement in the ViewDF. Or the view maintainer will execute the UPDATE statement. Note that ViewDF is different from the recursive query in the traditional DBMS. Although both maintain temporary tables and execute the same query for many times, ViewDF maintain the temporary tables as materialized views in order for later usage while recursive queries only temporarily use the intermediate table, and in each recursive step, it will replace the contents of the working table with the contents of the intermediate table, then empty the intermediate table. The partition subscripts referenced in the INITIALIZE and UPDATE statement are resolved to the physical relations based on the information provided in the catalog table before the system executes these two statements. For example, $S[i]$ means the ith partition of the source table and will be resolved to the specific name of the source partition. We will illustrate the details of checking whether it is first time generation and the partition subscript resolution in Chapter 4. The output of these two statements will be loaded into new partitions of the materialized view.

In the following paragraphs, we show that ViewDF can be used to express the algorithms for incremental materialized view maintenance for SPJ queries (queries with select, project and join operators).

For select view, a *select* view is defined by the expression $V = \sigma_{C(Y)}(R)$, where C is a boolean expression defined on $Y \subseteq R$. As we partition the source table and materialized view based on timestamp, every time a new source partition $R[i]$ arrives, instead of re-computing all the source partitions and get the final materialized view, we generate a new materialized view partition $View8[i]$ only based on $R[i]$. The ViewDF can be expressed as:

```
CREATE VIEW View8 AS
```

```
INITIALIZE View8[k] AS
    SELECT *
    FROM R[k]
    WHERE C(Y)
UPDATE View8[i] AS
    SELECT *
    FROM R[i]
    WHERE C(Y)
```

For project view, a *project* view is defined by the expression $V = \prod_X(R)$, where $X \subseteq R$. As we partition the source table and materialized view based on timestamp, every time a new source partition $R[i]$ arrives, instead of recomputing all the source partitions and get the final materialized view, we generate a new materialized view partition $View9[i]$ only based on $R[i]$. The ViewDF can be expressed as:

```
CREATE VIEW View9 AS
INITIALIZE View9[k] AS
    SELECT X
    FROM R[k]
UPDATE View9[i] AS
    SELECT X
    FROM R[i]
```

For join view, a *join* view is defined by the expression $V = R_1 \bowtie R_2$, assuming R_1 and R_2 join on attribute list $Attr$. As we partition the source table and materialized view based on timestamp, every time a new source partition $R_1[i]$ and $R_2[i]$ arrives, instead of recomputing all the source partitions and get the final materialized view, we generate a new materialized view partition $View10[i]$ only based on $R_1[i]$ $R_2[i]$. Note that the join here is different from the normal join in the database. It is like a full partition-wise join [3] because both table R_1 and R_2 are co-partitioned on the same key “timestamp” and the join can be executed in partition level with one partition join result unaffected by other partition join results. The ViewDF can be expressed as:

```
CREATE VIEW View10 AS
INITIALIZE View10[k] AS
    SELECT *
    FROM R1[k] inner join R2[k] on R1[k].Attr = R2[k].Attr;
```

```
UPDATE View10[i] AS
    SELECT *
    FROM R1[i] inner join R2[i] on R1[i].Attr = R2[i].Attr;
```

A combined SPJ query is defined by the expression $V = \prod_X(\sigma_{C(Y)}(R_1 \bowtie R_2))$, assuming R_1 and R_2 join on attribute list $Attr$. As we partition the source table and materialized view based on timestamp, every time a new source partition $R_1[i]$ and $R_2[i]$ arrives, instead of recomputing all the source partitions and get the final materialized view, we generate a new materialized view partition $View11[i]$ only based on $R_1[i]$ $R_2[i]$. The ViewDF can be expressed as:

```
CREATE VIEW View11 AS
INITIALIZE View11[k] AS
    SELECT X
    FROM R1[k] inner join R2[k] on R1[k].Attr = R2[k].Attr
    WHERE C(Y);
UPDATE View11[i] AS
    SELECT X
    FROM R1[i] inner join R2[i] on R1[i].Attr = R2[i].Attr
    WHERE C(Y);
```

3.2.3 Interfaces for Other Systems

ViewDFs can also easily express the optimizations incorporated in DBToaster [7, 9] and [45]. We will illustrate these ViewDFs in Chapter 4. For DBToaster, we will demonstrate that as long as the viewlet transforms[2] are provided, we can easily express DBToaster's idea by ViewDFs. We will also show how to use ViewDFs to express the optimizations for sliding window aggregation [9, 45].

[2]We will explain viewlet transforms in chapter 4.

Chapter 4

Query-to-ViewDF Translation

This chapter provides algorithms for the Query-to-ViewDF translation. Our algorithms can automatically translate various queries/operators including pattern matching queries and sliding window aggregation into ViewDFs that encode incremental materialized view maintenance. For pattern matching queries, we make full use of Finite State Machine(FSM) and automatically generate auxiliary views to greatly increase the processing speed of pattern matching. For sliding window aggregation, instead of recomputing from the whole window, we provide ViewDFs to express the incremental maintenance.

4.1 Pattern Matching

This section provides algorithms to automatically translate from users' pattern matching queries to ViewDFs that encode incremental view maintenance. We use the pattern matching query discussed in Section 3.2.1 as an example. For the input data, we ensure there is only one tuple for each source-destination pair in each partition. A naive way to update $View2$ when a new batch of packet loss measurements arrives is to re-run the pattern query. As $View2$ requires maintenance of at least four consecutive measurements with high loss, the naive way needs to revisit the previous source partitions to find if there are at least four consecutive measurements. Because $View2$ requires "at least" four, it is unclear if it is necessary to revisit the previous 4, or 5, or even 100 or more source partitions. Unlike sliding window aggregation in the motivating example in which we know we can revisit the previous 60 partitions, for the pattern matching example, we even do not know how many previous partitions to revisit because there is no specific number provided. In addition, another drawback for the naive way is that, it will not only find those source-destination pairs that have reported at least four consecutive high-loss measurements as of the current time, but also all such intervals that happened in the past!

For incremental maintenance strategy, we define a "helper" view $Helper$ that keeps track of all source-destination pairs with at least one high-loss measurement, as well as the number of consecutive high-loss measurements for each such pair. We partition the $Helper$ view and $View2$ by minute. When a new partition of $View2[i]$ needs to be created, we first create a new partition of the helper view $Helper[i]$ by visiting its previous partition $Helper[i-1]$ and updating the high-loss measurement counts based on newly arrived measurements in $S[i]$. The incremental maintenance strategy can be expressed as: $Helper[i] = Helper[i-1] + S[i]$. We then generate $View2[i]$ by selecting from the helper view $Helper[i]$ all those source-destination pairs that have at least four consecutive high-loss measurements at the current time.

It is easy for ViewDF to express this optimization by considering partitions and queries that define their contents, which is similar to the way of using ViewDF to express the optimization of sliding window aggregation discussed in Chapter 3. The problems are how to generate ViewDF to encode the incremental maintenance strategy, and what should be maintained in the $Helper$ view. We provide algorithms to make full use of finite-state machine (FSM), which is a mathematical model of computation used to design both computer games and sequential logic circuits, to solve these problems. The specific procedure is:

1. First generate an FSM based on users' view queries;

2. Generate ViewDF based on FSM;

3. Runtime translation.

In the next two subsections, we elaborate on the algorithms implementing this procedure using the view query provided in Section 3.2.1 as an example.

4.1.1 FSM Generation

This section discribes an algorithm for FSM generation (Algorithm 1) and explains the algorithm using the example view query. Algorithm 1 accepts user's view query Q as input and outputs the generated FSM F and an auxiliary list *auxlist* which contains the needed maintained values in the helper view. Formally an FSM $F = (S[], E[][])$ consists of a set of States, S and a set of directed edges with a set of formulas E. The output FSM for the

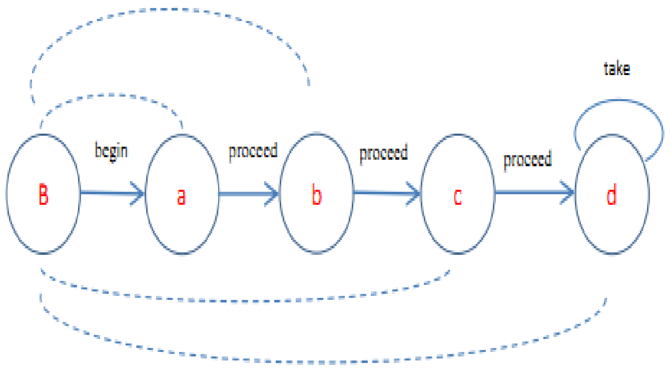


Figure 4.1: FSM generated for the example query

Table 4.1: Basic Formulas on Edges

E[0][1]	E[1][2]	E[2][3]	E[3][4]	E[4][4]
$loss > 10$	$loss > 10$	$loss > 10$	$loss > 10$	$loss > 10$

above query using Algorithm 1 is illustrated in Figure.4.1. The states $S[]$ and edges with predicates $E[][]$ are generated based on the following rules.

States. $S[]$ is constructed from the PATTERN clause in the query as shown in Line 3-10 in Algorithm 1. In addition, for every query, we add state $S[0]$ as the beginning state, that is, the state before any input has been processed. In the example view query, $S[]$ is constructed from “PATTERN(a,b,c,d+)”. Combined with $S[0]$, $S[]$ consists of 5 states: $S[0]$, a, b, c, $d+$. In Figure 4.1, we can see that these states are arranged as a linear sequence with each state represented as a circle. There are two types of states: singleton state (which does not contains a ‘+’ in the PATTERN clause) and Kleene Closure state. For the example query, a, b, c are singleton states while $d+$ is a Kleene Closure state. The difference between these two types of states is whether the state contains self edge or not, which will be discussed in the next paragraph.

Edges and Predicates. Each state is associated with a number of edges, representing the actions that can be taken and the checking predicates at the state. In Algorithm 1, $E[i][j]$ represents the edge between state $S[i]$ and $S[j]$. If one tuple in source partition k stays in state $S[i]$, when the corresponding tuple in the new source partition $k+1$ arrives with the value satisfying the formulas in $E[i][j]$, then we say the tuple's state moves from state $S[i]$ to $S[j]$ (for simplicity, we say $S[i]$ moves to $S[j]$ in the following paragraphs) and the formulas in $E[i][j]$ are called *state transition predicates*. For a singleton state $S[i]$, it can only move to the next state if the tuple satisfies $E[i][i+1]$, or move to the beginning state $S[0]$ if $E[i][i+1]$ is not satisfied. For a KleeneClosure state $S[j]$, apart from moving to the next state $S[j+1]$ and $S[0]$,it can also move to $S[j]$ itself when $E[j][j]$ is satisfied. $E[j][j]$ is called a self edge. In summary, there are three types of edges: a proceed edge denoted as $E[i][i+1]$, a self edge denoted as $E[i][i]$, and a backtrack edge denoted as $E[i][0]$ indicating state $S[i]$ goes back to $S[0]$, which usually occurs when the formulas on the corresponding edges cannot be satisfied or the whole match is finished.

The state transition predicates are constructed from the predicates in WHERE clause (from line 11 in Algorithm 1). One thing to note is that, as every state has a backtrack edge, we ignore the backtrack edge generation in Algorithm 1. For each *pred*, we can get its left operand and right operand (denoted as lOperand and rOperand in Algorithm 1), e.g., for $a.loss > 10$, $a.loss$ is the left operand and 10 is the right operand. In addition, we maintain an auxiliary list (denoted as *auxlist* in Algorithm 1 which contains the intermediate variables that need to be maintained in the *Helper* view. The corresponding predicate is constructed based on the following set of rules:

1. If *pred* only contains one state $S[p]$, for line 12-31 in Algorithm 1, we illustrate different sub-conditions as follows:

    (a) The lOperand in *pred* does not contain $[i]$: for example, $a.loss > 10$. This condition is shown in line 30, in which a proceed edge $E[p-1][p]$ is generated. As there can be more than one predicate that satisfy this condition, in line 30, we append the predicate to $E[p-1][p]$.

    (b) The lOperand in *pred* contains $[i]$: this condition means *pred* is a predicate on the self edge for the Kleene Closure state (shown in line 13-28) because there is no subscription $[i]$ for singleton state. Therefore, we generate the self edge $E[p][p]$ shown in line 14. This condition contains the following sub-conditions.

        i. The rOperand contains $[..i-1]$: for example, $d[i].loss > avg(d[..i-1].loss)$. This condition means that there is a comparison between the ith occurrence of Kleene Closure state and the aggregation of the previous occurrences of the Kleene Closure state. In this condition, we need to maintain the corresponding aggregation value in the *auxlist* (shown in lines 15-19) so that we can incrementally maintain the materialized view. In the proposed example, we need to maintain sum(loss) and count of d's occurrences for all the previous loss value for state 'd' in order to compute the average. For predicate like $d[i].loss > sum(d[..i-1].loss)$ and $d[i].loss > count(d[..i-1].loss)$, we only need to maintain sum(loss) or count of d's occurrences for all the previous occurrences of state 'd'. In the following algorithms, for simplicity, we only list the algorithms for sum and count.

   ii. The rOperand contains $[i-1]$: for example, $d[i].loss > d[i-1].loss$. This condition means that there is a comparison between the current value and the previous value. We need to maintain the previous value in the *auxlist* (shown in line 20-23) so that we can avoid revisiting the previous source partition. In the proposed example, we need to maintain the loss value in the previous source partition in the *auxlist*.
   iii. The rOperand contains [1]: for example, $d[i].loss > d[1].loss$. This condition means that there is a comparison between the current value and the value of the first occurrence of the Kleene Closure state. We need to maintain the value of the first occurrence in the *auxlist* (shown in lines 24-27) so that we can avoid revisiting the source partition which contains the first occurrence. In the proposed example, we need to maintain the loss value in the source partition which contains the first occurrence in the *auxlist*.
   iv. The rOperand does not contain any of above three: for example, $d[i].loss > 10$. In this case, we do not need to maintain any value in the *auxlist*.

2. If *pred* contains two states $S[p]$ and $S[p-1]$ shown in lines 32-36, then it is a proceed edge from $S[p-1]$ to $S[p]$. In this condition, we need to maintain the $S[p-1]$'s corresponding value in *auxlist*. For example, predicate $b.loss > a.loss$ contains two states a and b. We need to maintain the loss value of a in the *auxlist*.

Based on the above rules, the predicates on edges for Figure 4.1 are in Table 4.1 and the *auxlist* does not need to contain anything.

4.1.2 FSM-based ViewDF

We provide the algorithm for ViewDF generation in Algorithm 2 and explain it using the provided view query in Section 3.2.1. Algorithm 2 accepts user's query Q, the FSM $F(S[], E[][])$, *auxlist* and the name of the materialized view *mv* as input, and outputs the final ViewDF query $ViewDFQ$ which encodes the incremental view maintenance strategy. The FSM $F(S[], E[][])$ and *auxlist* are generated by Algorithm 1.

Algorithm 1: Generate_FSM Algorithm

```
Input: A user's query $Q$
Output: an FSM $F(S[], E[][])$, $auxlist$
 1: $S[0]$ = begin state;
 2: $j = 1$;
 3: for all variable $v$ in PATTERN clause do
 4:    $S[j] = v$;
 5:    $S[j]$.isKleeneClosure = false;
 6:    if $v$.contains('+') then
 7:       $S[j]$.isKleeneClosure = true;
 8:    end if
 9:    j++;
10: end for
11: for all predicate $pred$ in WHERE clause do
12:    if $pred$ referencing one state $S[p]$ alone then
13:       if $pred$.lOperand.contains('[i]') then
14:          $E[p][p]$.append($pred$.subString(pred.indexOf('.')));
15:          if $pred$.rOperand.contains('[..i-1]') then
16:             $aggrName$ = rOperand.subString(0,rOperand.indexOf('('));
17:             $stateAttribute$ = rOperand.subString(rOperand.indexOf('.'),
    rOperand.indexOf(')'));
18:             $auxlist$.add($aggrName(S[p].stateAttribute)$);
19:          end if
20:          if $pred$.rOperand.contains('[i-1]') then
21:             $stateAttribute$ = rOperand.subString(rOperand.indexOf('.'));
22:             $auxlist$.add($S[p].stateAttribute$);
23:          end if
24:          if $pred$.rOperand.contains('[1]') then
25:             $stateAttribute$ = rOperand.subString(rOperand.indexOf('.'));
26:             $auxlist$.add($S[p].stateAttribute$);
27:          end if
28:       end if
29:    else
30:       $E[p-1][p]$.append($pred$.subString(pred.indexOf('.')));
31:    end if
32:    if $pred$ referencing two states $S[p]$ and $S[p-1]$ then
33:       $E[p-1][p]$.append($pred$.subString(pred.indexOf('.')));
34:       $stateAttribute$ = rOperand.subString(rOperand.indexOf('.'));
35:       $auxlist$.add($S[p-1].stateAttribute$);
36:    end if
37: end for
```

Algorithm 2: ViewDF Generation

```
Input: A user's query $Q$, an FSM $F(S[], E[][])$, $auxlist$, name of the materialized view $mv$
Output: A ViewDF query $ViewDFQ$
 1: $select-list = SELECT clause in $Q$;
 2: $last-state = S[S.length-1];
 3: $num-state = S.length;
 4: for all term $t$ in $select-list do
 5:    if $t$.isAggregation() then
 6:       $auxlist$.add($t$);
 7:    else
 8:       $non-aggr-list.add($t$);
 9:    end if
10: end for
11: $helper-select-list = $non-aggr-list + currentState + $auxlist$;
12: Write("CREATE VIEW Helper AS");
13: Write("INITIALIZE $Helper[k]$ AS");
14: Write("SELECT $non-aggr-list, 1 as currentState, ");
15: for all term $v$ in $auxlist$ do
16:    if $v$.contains("sum") then
17:       Write("v.subString(v.indexOf('('), v.indexOf(')')) as sum");
18:    else if $v$.contains("count") then
19:       Write("1 as count, ");
20:    else
21:       Write("v");
22:    end if
23: end for
24: Write("FROM $S[k]$ WHERE $E[0][1]$");
25: Write("UPDATE $Helper[i]$ AS");
26: Write("SELECT $non-aggr-list, case");
27: for all i=1 to num-state do
28:    if $S[i]$.isKleeneClosure then
29:       Write("when prevState=S[i] and E[i][i] then prevState ");
30:    end if
31:    Write("when prevState=S[i] and E[i][i+1] then prevState+1 ");
32: end for
33: Write("end,");
```

```
34: for all term $v$ in auxlist do
35:   if $v$.contains("sum") then
36:     Write("sum+v.subString(v.indexOf('('), v.indexOf(')')) as sum");
37:   else if $v$.contains("count") then
38:     Write("1+count as count");
39:   else
40:     Write("v");
41:   end if
42: end for
43: Write("FROM(");
44: Write("SELECT ");
45: for all term $n$ in $non-aggr-list do
46:   Write("New.n, ");
47: end for
48: Write("Prev.currentState as prevState, ");
49: Write("auxlist");
50: Write("FROM $S[i]$ as New Natural Left Outer Join $Helper[i-1]$ as Prev WHERE
    $E[0][1]$) AS Temp");
51: Write("WHERE currentState!=0");
52: Write("CREATE VIEW mv AS");
53: Write("INITIALIZE $mv[k]$ AS");
54: Write("SELECT $select-list");
55: Write("FROM $Helper[k]$");
56: Write("WHERE currentState = $last-state");
57: Write("UPDATE $mv[i]$ AS");
58: Write("SELECT $select-list");
59: Write("FROM $Helper[i]$");
60: Write("WHERE currentState = $last-state");
```

In the algorithm, we first get the $select-list from the SELECT clause in the user's view query. In the example query, "src, dest, count(*) and sum(loss)" are the elements in the $select-list. For each element in the $select-list, we check whether it is an aggregation (lines 4-9). If it is, we will add the element to the *auxlist* (e.g., count(*) and sum(loss) in the example) which will later be used as the schema of the Helper table. If it is not, we will add the element to the $non-aggr-list (e.g., src, dest in the example). Finally, we determine Helper table's schema $helper-select-list by combining the $non-aggr-list, *auxlist* and another attribute *currentState* shown in line 11. The reason that we add *currentState* is that it is a necessary intermediate variable for the incremental view maintenance; it records the state that the event stays in the matching in order for us to quickly find the corresponding state and predicates in the FSM.

After getting the necessary (line 12), we begin the ViewDF code generation. The generated code consists of two parts: INITIALIZE and UPDATE. In the INITIALIZE part, the ViewDF initializes the maintained Helper table $Helper[k]$. In $Helper[k]$, we maintain the attributes in the $non-aggr-list, currentState and *auxlist* when the $E[0][1]$ is satisfied, shown in line 14-24. In the UPDATE part shown from line 25, every time a new source partition arrives, we need to generate a new Helper partition $Helper[i]$ based on $Helper[i-1]$ and the new source partition $S[i]$ shown in Line 50. In $Helper[i]$'s SELECT clause, we keep the $non-aggregate-list unchanged and generate new *currentState* based on new values in $S[i]$ and prevState which is *currentState* in $Helper[i-1]$, as shown in line 26-33. Note that we treat differently for Kleene Closure state and non-Kleene Closure state because Kleene Closure state contains both self edge and proceed edge. For both Kleene Closure and non-Kleene Closure state, when the predicates on the proceed edge are satisfied, *currentState* will move to the next state $prevState + 1$. For Kleene Closure state, when the predicates on the Self edge are satisfied, *currentState* remain unchanged. We process the terms in the *auxlist* in lines 34-42. We can see for sum and count, since we maintain the sum and count in the $Helper[i-1]$, we can easily get the sum and count for $Helper[i]$, meanwhile avoid revisiting the previous source partitions.

Based on the *Helper* table's maintained attributes, we can easily generate $mv[i]$ by selecting the needed attributes from $Helper[i]$ with *currentState* equals to the last state in the FSM, which means that the specific pattern has been found and can be maintained in the materialized view. The code is shown in line 52-60.

After replacing $E[0][1]$, $helper-select-list, $non-aggregate-list, $S[i]$, $E[i][i]$ with the corresponding lists and predicates, the automatically generated ViewDF for the proposed view query is:

```
CREATE VIEW Helper AS
INITIALIZE Helper[k] AS
    SELECT timestamp,src,dest,1 AS currentState, 1 AS count, loss AS sum_loss
    FROM S[k]
    WHERE loss>10
UPDATE Helper[i] AS
    SELECT timestamp,src,dest,
    case when prevState=0 and loss>10 then prevState+1
         when prevState=1 and loss>10 then prevState+1
         when prevState=2 and loss>10 then prevState+1
         when prevState=3 and loss>10 then prevState+1
         when prevState=4 and loss>10 then prevState end,
    sum_loss+loss as sum_loss, 1+count as count
    FROM
    (
        SELECT
            New.timestamp, New.src as src, New.dest as dest,
            Prev.currentState as prevState, count, sum_loss
        FROM
            S[i] as New Natural
            Left Outer Join Helper[i-1] as Prev
        WHERE New.loss>10
    )AS Temp
    WHERE currentState!=0
```

```
CREATE VIEW mv AS
INITIALIZE mv[k] AS
    SELECT timestamp, src, dest, count, sum_loss
    FROM Helper[k]
    WHERE currentState=4
UPDATE mv[i] AS
    SELECT timestamp, src, dest, count, sum_loss
    FROM Helper[i]
    WHERE currentState=4
```

One thing to note for the generated ViewDF is that we use CASE to provide conditional execution. Another way to express ViewDF for *Helper* table for the given view query can be with many unions expressed as follows. The following query is not automatically generated from Algorithm 2. It is presented only for comparison.

```
CREATE VIEW Helper AS
INITIALIZE BK[k] AS
    SELECT timestamp,src,dest,1 AS prevState, 1 as count, loss as sum_loss
    FROM S[k]
    WHERE loss>10
UPDATE BK[i] AS
    SELECT New.timestamp, New.src as src, New.dest as dest,
           Prev.currentState as prevState, count, sum_loss
    FROM S[i] as New Natural
        Left Outer Join Helper[i-1] as Prev
    WHERE loss>10

CREATE VIEW Helper AS
INITIALIZE Helper[k] AS
    SELECT timestamp,src,dest,1 AS currentState, 1 AS count, loss AS sum_loss
    FROM S[k]
    WHERE loss>10
UPDATE Helper[i] AS
    SELECT timestamp, src, dest, prevState+1, count+1, sum_loss+loss
    FROM BK[i]
    WHERE currentState!=4 and loss>10
    UNION
    SELECT timestamp, src, dest, prevState, count+1, sum_loss+loss
    FROM BK[i]
    WHERE currentState=4 and loss>10
```

However, for the ViewDF with many unions, the query execution plan needs to scan the table many times (number of unions) which is very time consuming, while using CASE to provide conditional execution requires only one table scan. Therefore, we use CASE to provide conditional execution in Algorithm 2.

4.1.3 Runtime Translation

In this section, we discuss how to resolve the partition subscripts in the INITIALIZE or UPDATE statement to their physical tables. In our implementation, we partition both the source table and materialized view by timestamp which is UNIX time. Based on the concept of inheritance, we first create a parent table with the name *p* and the corresponding schema. Then when a new partition needs to be initialized or updated, we create a separate table (per partition) that inherits the parent table *p*. We name each partition by attaching parent table's name *p* with the partition number. As mentioned in Chapter 3,

Algorithm 3: Runtime Translation

```
Input: A ViewDF query $ViewDFQ$
Output: An executable ViewDF query $ViewDFQ\_exe$
 1: Subscript oldsub = 0;
 2: Subscript prev_sub = 0;
 3: Subscript newsub = 0;
 4: Partition_name lname;
 5: Physical Partition pname;
 6: for all sentence in $ViewDFQ$ do
 7:   if sentence.contains('[') and sentence.contains(']') then
 8:     oldsub = sentence.subString(sentence.indexOf('['), sentence.indexOf(']'));
 9:     lname = get the name before '[oldsub]';
10:     prev_sub = select s from CATALOG where n = lname;
11:     newsub = replace $i$ or $k$ in oldsub with prev_sub+1;
12:     pname = lname + "_" + newsub;
13:     sentence = replace lname[oldsub] with pname;
14:   end if
15:   Write sentence to $ViewDFQ\_exe$;
16: end for
```

we maintain a CATALOG table to record the name of the physical tables n and the previous partition's subscript s. Every time a new partition of a table T needs to be created, the partition number of T is generated by increasing s_T with 1 where s_T represents the previous partition's subscript for T, initialized as 0 when there is no partition of T.

When a new partition of a source table or a materialized view needs to be created, the partition subscripts referenced in the INITIALIZE or UPDATE statement are resolved to their physical tables. For example, assuming the source table's name is S, when the new partition of the source table arrives, the partition's name is $S\_(s_S + 1)$, where s_S is the subscript of the previous partition of S. Then we update s_S to be $s_S + 1$. For materialized view with name mv, when a new partition needs to be generated, the partition's name is $mv\_(s_{mv} + 1)$, where s_{mv} is the subscript for the previous partition of mv. Note that since the partition size of the source table and materialized view can be different, their partition numbers (even two partitions with the same timestamp) can be different. The algorithm for resolving the partition subscripts is provided in Algorithm 3 based on the procedures that we described.

4.2 Sliding Window Aggregation

In this section, we show how ViewDF can be used to express the sliding window aggregation queries. Sliding window aggregation is a large class of continuous queries in SDWs. Complete re-evaluation is the naive and straightforward approach to process sliding window aggregation queries. The idea is simple: every time a window is complete, we compute the result over all tuples in the window. Although this could be sufficient for tumbling and hopping windows, i.e., windows that slide per one, or more than one, full window size at a time, it is far from optimal when it comes to the more common and challenging case of *overlapping sliding windows*, that is, windows slide less than one window size at a time. In this case, complete re-evaluation will process the same data over and over again. For example, for a window with size n minutes and the window slides 1 minute per time, the same tuple will be processed n times until it finally expires from the window. In this thesis, we only consider *overlapping sliding windows*. In this section, we will show how ViewDF can be used to express the efficient way of processing sliding window aggregation queries proposed in [9, 45]. In addition, we also propose our own optimization expressed by ViewDF.

The basic idea of Liarou et al [45] approach was discussed in Chapter 2. To repeat, it consists of four steps: 1) Split the input stream into n basic windows, 2) Process each basic window separately and store the partial results, 3) Merge partial results from each basic window, 4) Slide to prepare for the next basic window. As the partial results are stored, we don't need to recompute every tuple from scratch for the next sliding window. One thing to note is that the size of the basic window is equal to the sliding length of the window. For example if the window slides 1 minute each time, the size of the basic window will be 1 minute.

Arasu et al [9] propose several types of data structures called *synopses* to store enough states to efficiently compute various sliding window aggregates. Figure 4.2 illustrates a *prefix synopses* that stores pre-computed values over prefixes of the stream. Different lengths of prefix synopses are maintained. To compute f over a sliding window of size ns at time t, i.e., $f(t - ns, t])$, we only need to compute $f([1, t]) - f([1, t - ns])$. This synopsis is suitable for subtractable aggregate functions such as SUM and COUNT. An *interval synopses* is used with distributive aggregates that are not subtractable, such as MIN and MAX [9]. Figure 4.3 shows an interval synopses to compute MAX over a window size of 7s. In Figure 4.3, we can see the four rectangles in the two upper lines represent four interval synopses. The rectangle in the first upper line represents a synopses with a 4-second interval and the three rectangles in the second upper line represent synopses with a 2-second interval. In order to compute the max value at time t, i.e., $max(t - 7s, t]$, we

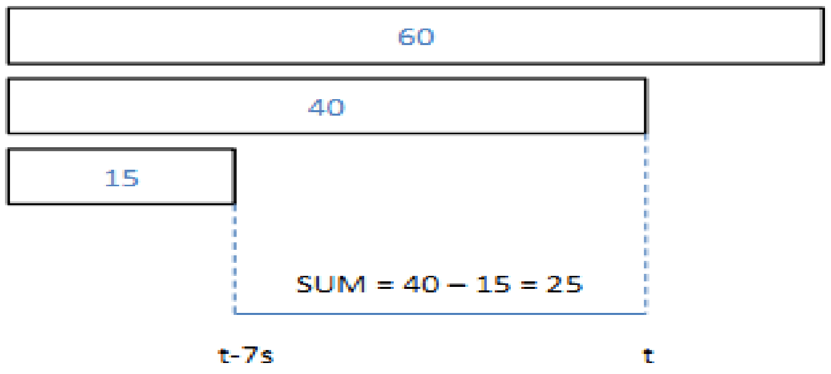


Figure 4.2: A prefix synopsis

only need to compute the max values from three rectangles with red numbers shown in Figure 4.3. In general, the length of the interval synopses can be 2^i with i a non-negative integer. In this way, we avoid recomputing the max value from scratch. The *synopses* will be maintained as auxiliary materialized views in our framework. We will explain the *prefix synopses* using a query with sum and *interval synopses* using a query with max.

We highlighted our proposed optimization for sliding window operation in Chapter 1. The basic idea is: we make full use of the previous materialized view partition, add the contribution of the new data and subtract the contribution of the expired data.

In the following paragraphs, we will use the following two queries to explain how we use ViewDF to express the three optimizations discussed above for sliding window aggregation queries.

```
select src, dest, sum(loss) from S group by src, dest;

select src, dest, max(loss) from S group by src, dest;
```

The source table S has the same schema as in the motivating example. For the first and second queries, we want to materialize a view containing the total packet loss (for first query) or maximum loss (for second query) for each source-destination pair over a sliding window of 60 minutes, updated every minute to reflect the windowed total as of that minute. We assume there is one source partition arriving every minute and the window contains 60 source partitions. Every minute, the window slides one partition. One thing to note is that, in each partition, there can be more than 1 tuple for each source and destination pair.

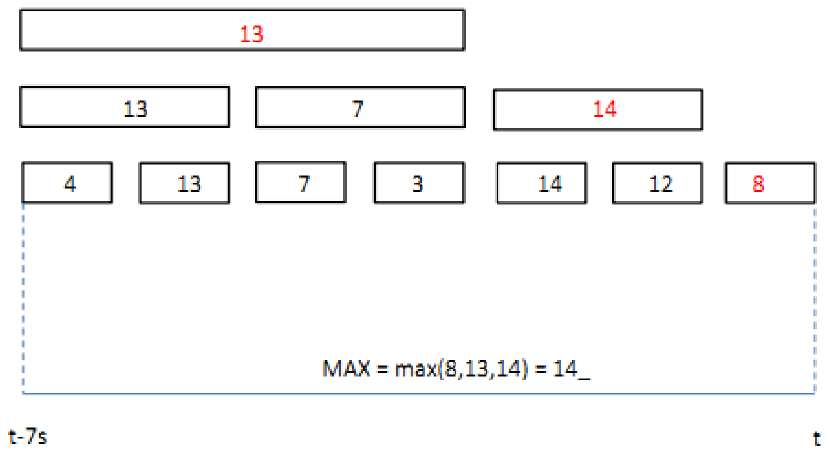


Figure 4.3: An interval synopsis

For the first query, the ViewDF for the idea in Liarou et al's proposal [45] is as follows. As the window slides one partition each minute, the size of the basic window should be one partition. Therefore, we split the window into 60 partitions, get the sum of the loss for each partition, and store the partial results in the auxiliary partitions (denoted as $aux[i]$ in the following ViewDF query). Then we generate one partition of $View3$ by merging the results in $aux[i]$. For the execution of the following ViewDF query, we don't begin the INITIALIZE part until the first window is full. Only for the first window, we need to process every partition. We process the next window in the UPDATE part, in which we only need to compute the newest basic window (newest partition) and combine with the pre-computed results from the non-expired basic windows. For all the following ViewDFs, we use "..." to omit the repeated operations.

```
CREATE VIEW aux3 AS
INITIALIZE aux3[k] AS
    select src,dest,sum(loss) as loss from S[k-59] group by src,dest;
    ...
    select src,dest,sum(loss) as loss from S[k] group by src,dest;
UPDATE aux3[i] AS
    select src,dest,sum(loss) as loss from S[i] group by src,dest;

CREATE VIEW View3 AS
INITIALIZE View3[k] AS
```

```
    select src,dest,sum(loss) as sum_loss from(
       select * from aux3[k-59]
       union all
       ...
       union all
       select * from aux3[k]) as X
    group by src,dest;
UPDATE View3[i] AS
    select src,dest,sum(loss) as sum_loss from(
          select * from aux3[i-59]
          union all
          ...
          union all
          select * from aux3[i]) as X
    group by src,dest;
```

For the first query, the ViewDF corresponding to Arasu et al [9] proposal is as follows. As sum is a subtractable aggregation function, we store the prefix synopses in the auxiliary partitions (denoted as $aux[i]$ in the following ViewDF query). From the ViewDF query, we can see the content in $aux[k]$ is very different from the content in [45]'s ViewDF query. The latter only stores the sum value for the $S[k]$ partition while the former stores the sum value from the very beginning of the source partition to $S[k]$ partition. For efficiency, we don't recompute every partition to get $aux[k]$. Instead, we compute $aux[k]$ based on $aux[k-1]$ and $S[k]$. For the execution of the following ViewDF query, we don't begin the INITIALIZE part until the first window is full. Only for the first window, we need to process every partition. We process the next window in the UPDATE part, in which we only need to compute the newest prefix synopses (from the very beginning to the newest partition) and subtract corresponding prefix synopses. The reason that we add "where $sum\_loss > 0$" in the UPDATE statement of $View4$ is because we want to exclude the tuples that exist in the expired window, but not exist in the new window.

```
CREATE VIEW aux4 AS
INITIALIZE aux4[k] AS
    select src,dest,sum(loss) as loss from S[k-59] group by src,dest;
    ...
    select src,dest,sum(loss) as loss from(
       select * from aux4[k-1]
       union all
       select src,dest,sum(loss) as loss from S[k] group by src,dest) as X
    group by src, dest;
UPDATE aux4[i] AS
    select src,dest,sum(loss) as loss from(
       select * from aux4[i-1]
       union all
       select src,dest,sum(loss) as loss from S[i] group by src,dest) as X
    group by src, dest;

CREATE VIEW View4 AS
INITIALIZE View4[k] AS
    select * from aux4[k];
UPDATE View4[i] AS
    select * from
    (
       select src,dest,sum(loss) as sum_loss from(
           select * from aux4[i]
           union all
           select src,dest,loss*(-1) as loss from aux4[i-60]) as X
       group by src,dest
    ) as Y where sum_loss>0;
```

For the first query, the ViewDF for our optimization is as follows. Instead of accessing the most recent 60 minutes of data in S and recomputing the total loss for each source-destination pair, we take the total packet loss computed over the previous window, add the loss from the current minute, and subtract the oldest loss measurement from the previous window which is outside the current window. The formula can be expressed as: $View3[i] = View3[i-1] + S[i] - S[i-60]$. The subscripts for $View3$ and S denote the partition's number. For the execution of the following ViewDF query, again, we don't begin the initialize part until the first window is full. Only for the first window, we need to process every partition similar to the naive way of complete re-evaluation. We omit the INITIALIZE part in the following ViewDF because it is the same as the one in the ViewDF for [45]. The reason that we add "where sum_loss¿0" in the UPDATE statement of $View5$ is because we want to exclude the tuples that exist in the expired window, but not exist in the new window.

```
CREATE VIEW View5 AS
```

```
INITIALIZE View5[k] AS
    select src,dest,sum(loss) as sum_loss from(
       select * from aux5[k-59]
       union all
       ...
       union all
       select * from aux5[k]) as X
    group by src,dest;
UPDATE View5[i] AS
    select * from
    (
       select src,dest,sum(loss) as sum_loss from(
           select src,dest,sum(loss) as loss from aux5[i] group by src,dest
           union all
           select src,dest,sum(loss)*(-1) as loss from aux5[i-60]
           group by src,dest
           union all
           select * from View5[i-1]) as X
       group by src,dest
    ) as Y where sum_loss>0;
```

For the second query, as our optimization is based on adding the contribution of new data and subtracting the contribution of old data, it is only useful for subtractable aggregation. Therefore, it cannot be used for the aggregation functions that are not subtractable, such as MAX and MIN. Therefore, we only use ViewDF to express the other two proposals.

For the second query, the ViewDF for Liarou et al [45] proposal is as follows. We split the window into 60 partitions, get the max value for each source-destination pair in each partition and store the partial results in the auxiliary partitions (denoted as $aux[i]$ in the following ViewDF query). Then we generate one partition of $View3$ by merging the results in $aux[i]$. For the execution of the following ViewDF query, we don't begin the initialize part until the first window is full. Only for the first window, we need to process every partition similar to the naive way of complete re-evaluation. We process the next window in the UPDATE part, in which we only need to compute the max value in the newest basic window (newest partition) and combine with the pre-computed results from the non-expired basic windows.

```
CREATE VIEW aux6 AS
```

```
INITIALIZE aux6[k] AS
    select src,dest,max(loss) as loss from S[k-59] group by src,dest;
    ...
    select src,dest,max(loss) as loss from S[k] group by src,dest;
UPDATE aux6[i] AS
    select src,dest,max(loss) as loss from S[i] group by src,dest;

CREATE VIEW View6 AS
INITIALIZE View6[k] AS
    select src,dest,max(loss) as max_loss from(
       select * from aux6[k-59]
       union all
       ...
       union all
       select * from aux6[k]) as X
    group by src,dest
UPDATE View6[i] AS
    select src,dest,max(loss) as max_loss from(
       select * from aux6[i-59]
       union all
       ...
       union all
       select * from aux6[i]) as X
    group by src,dest
```

4.3 DBToaster

DBToaster [7] provides a higher-order IVM as described next. They make use of discrete forward differences (delta queries) recursively, on multiple levels of derivation. That is, they use delta queries ("first-order deltas") to incrementally maintain the view of the input query, then materialize the delta queries as views, maintain these views using delta queries over the delta queries ("second-order deltas"), and continue alternating between materializing views and deriving higher-order delta queries for maintenance. We use the following example to explain this and show our ViewDF can also express this higher-order IVM.

Consider a query Q that counts the number of tuples in the product of relations R and S. We want to maintain Q's result in $View7$ under insertion. We use ΔR (resp. ΔS)

to represent the change to a view as one tuple is inserted into R (resp. S). We can use $\Delta_R Q$ and $\Delta_S Q$ (both are"first-order deltas") to maintain the view of Q. In addition, we use $\Delta_R \Delta_S Q$ and $\Delta_S \Delta_R Q$ (both are "second-order deltas") to materialize $\Delta_S Q$ and $\Delta_R Q$. As $\Delta_R \Delta_S Q$ and $\Delta_S \Delta_R Q$'s values are constant (1 in this example), the recursion stops. In total, in order to maintain $View7$, we maintain the four views: $\Delta_R Q$, $\Delta_S Q$, $\Delta_R \Delta_S Q$ and $\Delta_S \Delta_R Q$. We can simultaneously maintain all these views using each other, without the recomputation of the product of R and S. Specifically, the dependencies are as follows.

- Initialize
$$\Delta_S Q = count(R) \tag{4.1}$$
$$\Delta_R Q = count(S) \tag{4.2}$$
$$\Delta_R \Delta_S Q = \Delta_S \Delta_R Q = 1 \tag{4.3}$$

- On insert into R
$$Q_{new} = Q_{old} + \Delta_R Q \tag{4.4}$$
$$\Delta_S Q_{new} = \Delta_S Q_{old} + \Delta_R \Delta_S Q = \Delta_S Q_{old} + 1 \tag{4.5}$$
$$\Delta_R Q_{new} = \Delta_R Q_{old} + \Delta_R \Delta_R Q = \Delta_R Q_{old} + 0 \tag{4.6}$$

- On insert into S
$$Q_{new} = Q_{old} + \Delta_S Q \tag{4.7}$$
$$\Delta_R Q_{new} = \Delta_R Q_{old} + \Delta_S \Delta_R Q = \Delta_R Q_{old} + 1 \tag{4.8}$$
$$\Delta_S Q_{new} = \Delta_S Q_{old} + \Delta_S \Delta_S Q = \Delta_S Q_{old} + 0 \tag{4.9}$$

We can see Q's maintenance is based on $\Delta_R Q$ or $\Delta_S Q$ on insertion into R or S, and $\Delta_S Q$ and $\Delta_R Q$ are materialized based on $\Delta_R \Delta_S Q$ and $\Delta_S \Delta_R Q$. We use the following table to illustrate the maintained view for Q. Time 0 represents the initial state without any insertion. At time 0, relation R has 3 tuples and S has 4 tuples. From equations 3.1 and 3.2, we know $\Delta_S Q = count(R) = 3$ and $\Delta_R Q = count(S) = 4$. Initially, we compute the number of tuples for Q by multiplying $count(R)$ and $count(S)$ and getting 12. When there is an insertion into relation S at time 1, based on equation 3.7, we can get Q's result by computing $Q_{new} = Q_{old} + \Delta_S Q = 12 + 3 = 15$ instead of recomputing the produce of R and S. Note that the add operator is much cheaper than the multiply operator. Meanwhile, we update the $\Delta_R Q$ and $\Delta_S Q$ based on equations 3.8 and 3.9. When there is an insertion into relation R at time 2, based on equations 3.4, 3.5 and 3.6, we can easily get the Q's result.

Based on equations 3.4-3.6, we write a ViewDF for insertion to R as follows. Equation 3.5's ViewDF is:

Table 4.2: DBToaster materialize the product of two relations R and S

time	insert into	——R——	——S——	Q	$\Delta_R Q$	$\Delta_S Q$	$\Delta_S \Delta_R Q$ and $\Delta_R \Delta_S Q$
0	-	3	4	12	4	3	1
1	S	3	5	15	5	3	1
2	R	4	5	20	5	4	1

```
CREATE VIEW Helper_S AS
INITIALIZE Helper_S[k] AS
    SELECT count(*) as count
    FROM R
UPDATE Helper_S[i] AS
    SELECT count + 1 as count
    FROM Helper_S[i-1]
```

Equation 3.6's ViewDF is:

```
CREATE VIEW Helper_R AS
INITIALIZE Helper_R[k] AS
    SELECT count(*) as count
    FROM S
UPDATE Helper_R[i] AS
    SELECT count + 0 as count
    FROM Helper_R[i-1]
```

Equation 3.4's ViewDF for $View7$ is:

```
CREATE VIEW View7 AS
INITIALIZE View7[k] AS
    SELECT count(*) as count
    FROM R*S
UPDATE View7[i] AS
    SELECT Q.count + R.count as count
    FROM View7[i-1] as Q, Helper_R[i-1] as R
```

Based on equations 3.7-3.9, we write a ViewDF for insertion to S as follows. Equation 3.8's ViewDF is:

```
CREATE VIEW Helper_R AS
INITIALIZE Helper_R[k] AS
    SELECT count(*) as count
    FROM S
UPDATE Helper_R[i] AS
    SELECT count + 1 as count
    FROM Helper_R[i-1]
```

Equation 3.9's ViewDF is:

```
CREATE VIEW Helper_S AS
INITIALIZE Helper_S[k] AS
    SELECT count(*) as count
    FROM R
UPDATE Helper_S[i] AS
    SELECT count + 0 as count
    FROM Helper_S[i-1]
```

Equation 3.7's ViewDF for $View7$ is:

```
CREATE VIEW View7 AS
INITIALIZE View7[k] AS
    SELECT count(*) as count
    FROM R*S
UPDATE View7[i] AS
    SELECT Q.count + S.count as count
    FROM View7[i-1] as Q, Helper_S[i-1] as S
```

This example presents single-tuple update. Viewlet transforms are not limited to this but support bulk updates. For bulk updates for this example, we need to change equations 3.4 and 3.7 to the following in which n_R and n_S represent the number of tuples inserted into relation R and S:

$$Q_{new} = Q_{old} + \Delta_R Q * n_R \tag{4.10}$$

$$Q_{new} = Q_{old} + \Delta_S Q * n_S \tag{4.11}$$

Chapter 5

Experiments

We implemented all the techniques discussed in the previous chapters using Java and PostgreSQL's procedural language PL/PGSQL. In this chapter, we report on experiments that compare our proposed approach with the existing algorithms for pattern matching and sliding window aggregation. All the experiments are performed on a workstation with a AMD Phenom(tm) II X4 955 3200 Mhz processor and 8 GB memory running Linux 4.1. We use PostgreSQL as the underlying DBMS and set the size of the shared memory used by the database server to 600MB.

5.1 Data Generation

We generate the data for the source partitions by Java with the schema (src:String, dest:String, loss:Integer). We first write all the needed data to a file and then use PL/PGSQL to load the data from the file to the corresponding table. We create a database called "viewdf" in which we create three empty parent tables (source table, auxiliary table and materialized view table). For pattern matching, the schema of the materialized view is src:String, dest:String, count:Integer, sum_loss:Integer. The schema of the auxiliary table depends on different algorithms because different algorithms for pattern matching maintain different intermediate variables in the auxiliary tables. We will provide the schema of the auxiliary table in the next section. For sliding window aggregation, the schema of the materialized view is src:String, dest:String, sum_loss:Integer, and the schema of the auxiliary table is src:String, dest:String, loss:Integer. Every time we need to create a new partition, we trigger a PL/PGSQL function to create a child table from the corresponding parent table (inherit the corresponding schemas), and then either copy the data from the file to the child table or inserting data to the child table by executing a ViewDF query.

Since we have experiments that compare different algorithms under different selectivities[1], we need to generate loss values that distributed in different intervals as the user requests. We achieve this by generating random numbers for the loss values in specific ranges.

In addition, we have different requirements for the data in the source partitions. For the experiment on pattern matching, we generate only one tuple for each src/dest pair in every source partition. For the experiment on sliding window aggregation, we generate 10 tuples for each src/dest pair in every source partition.

5.2 Pattern Matching

5.2.1 Algorithms

In order to compare different algorithms for pattern matching, we first illustrate these algorithms using the example pattern matching query provided in Section 3.1.1.

Direct: We implement the naive method (Section 4.1) to update $View2$ when a new batch of packet loss measurements arrives. We call this "Direct" in the following experiments. The specific implementation is to re-run the pattern query. Since it is not known how many previous partitions should be revisited, we provide a variable called *scope* to represent the number of previous partitions to revisit and give it different values in the following experiments. Direct 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 intervals that happened in the past. To solve this problem, we check the source-destination pairs that have reported at least four consecutive high-loss measurements to see whether they happen as of the current time or not.

Incremental Strategy: We implement the incremental maintenance strategy proposed in Section 4.1 (first generate new $Helper$ view partition based on previous $Helper$ view partition, then generate new $View2$ partition based on the latest $Helper$ view partition) by the following three different algorithms.

1. **Hardcode**: In this algorithm, we use JDBC to connect to PostgreSQL and maintain $first\_red$, red_ct and sum_loss in the $Helper$ view partition. $first\_red$ keeps track

[1]Selectivity will be explained in the next section

of the timestamp of the first red measurement that satisfies four consecutive reds up to now (We call the measurement with loss value larger than 10 “red” measurement), *red_ct* counts the number of red measurements in the current interval and *sum_loss* sums up the loss values over the current interval. Therefore, the schema of the auxiliary table (*Helper* view) is src:String, dest:String, $first\_red$:timestamp, *red_ct*:Integer and *sum_loss*:Integer. Every time a new source partition $S[i]$ arrives, we generate the new partition $Helper[i]$ by visiting $Helper[i-1]$ and $S[i]$. The reason that we call this algorithm Hardcode is that, different from ViewDF proposed in Section 4.1 to automatically figure out what to maintain in the *Helper* view, this algorithm expects the user to figure out what to maintain in the *Helper* view. As it is not automatic for the pattern matching queries, we call it “Hardcode”.

The algorithm for Hardcode algorithm is in Algorithm 4. Specifically, for a src-dest pair tuple t in $S[i]$,

(a) t is a red tuple:

   i. If t exists in $Helper[i-1]$ and $first\_red$ for t in $Helper[i-1]$ is a non-zero value V, which means there are consecutive red measurements since time V until now, then the matching can continue.

   ii. Else, we need to begin a new run for pattern matching.

(b) If t is not a red tuple, there is no need to store this tuple in $Helper[i]$. This is because the example pattern matching query maintains materialized views that contain more than four consecutive reds as of the current time while tuple t destroys the consecutiveness.

2. **ViewDF_union**: We call the ViewDF with many unions in Section 4.1.2 “ViewDF_union”. The schema of the auxiliary table (*Helper* view) is src:String, dest:String, prevState:Integer, count:Integer and sum_loss Integer.

3. **ViewDF**: We call the last ViewDF in Section 4.1.2 “ViewDF”. The schema of the auxiliary table (*Helper* view) is the same as ViewDF_union’s.

5.2.2 Experiment Results

We conduct the following experiments to compare different algorithms for pattern matching from different aspects. All the experiments are based on the example pattern matching query provided in Section 3.1.1. All the results are average values after we run each experiment 50 times.

Algorithm 4: Algorithm for Hardcode

```
Input: partition $S[i]$ and partition $Helper[i-1]$
Output: partition $Helper[i]$
 1: statement stmt;
 2: ResultSet rs = stmt.executeQuery("select * from S[i] order by src,dest");
 3: ResultSet rs2 = stmt.executeQuery("select * from Helper[i-1] order by src,dest");
 4: while rs.next() do
 5:   while rs2.next() do
 6:     tuple t = rs.getTuple();
 7:     if t is a red tuple then
 8:       if t exists in rs2 AND $t.first\_red != 0$ then
 9:         red_ct = rs2.get(red_ct);
10:         sum_loss = rs2.get(sum_loss);
11:         red_ct+=1;
12:         sum_loss+=rs.get(loss);
13:       else
14:         red_ct=1;
15:         sum_loss = rs.get(loss);
16:         first_red = currentTimestamp;
17:       end if
18:       Write t to a file helper
19:     end if
20:   end while
21: end while
22: load data from file helper to partition $Helper[i]$
```

Scalability

In this experiment, we compare the execution time of the four algorithms (Direct, ViewDF_union, ViewDF and Hardcode) to show ViewDF's scalability. Figure 5.1 shows the time comparison of these methods for varying number of tuples in the source tables when the selectivity (ratio between the number of red tuples and the total number of tuples in the source table) is 0.1 and the scope for Direct is 20. From Figure 5.1, we can see the execution time of every method increases linearly when the number of tuples in the source partitions increases. Direct's execution time is worse than all the other methods. This is because Direct needs to recompute over all the needed source partitions while other three methods use Helper view to avoid the recomputation. ViewDF_union's execution time is much larger than Hardcode and ViewDF. This is because ViewDF_union with several unions needs to scan

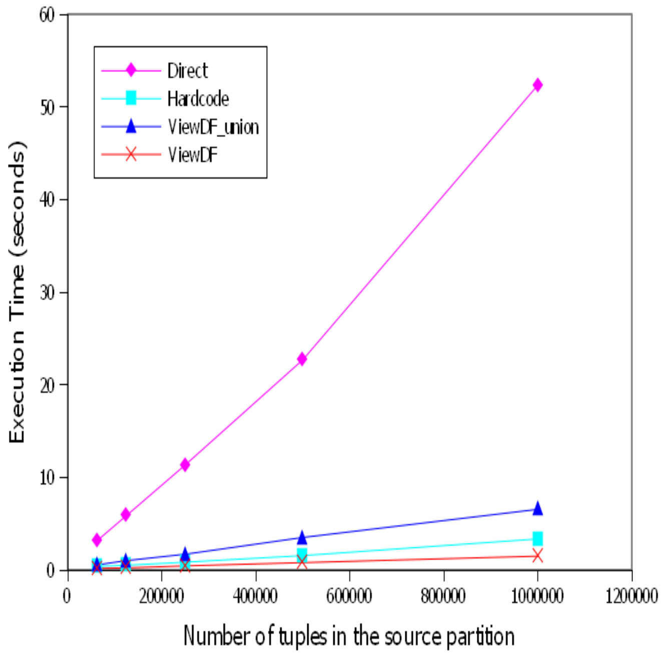


Figure 5.1: Pattern Matching: comparison between Direct, Hardcode, ViewDF_union and ViewDF under different number of tuples in the source partition

the source table many times while Hardcode and ViewDF only need to scan once. ViewDF is a little better than Hardcode, because Hardcode needs to move two cursors (rs and rs2 in the pseudocode) and process the cursor-pointed data using JDBC while ViewDF depends on the underlying DBMS to generate an optimized query execution plan. The result can show ViewDF's scalability. Figure 5.1 show that when the number of tuples in source table is 10^6, the execution time for ViewDF is only 5 seconds, which is very efficient.

Scope

In this experiment, we compare the execution time of the four algorithms by varying the scope from 5 to 100 to show the efficiency of the three algorithms with incremental strategy. Figure 5.2 shows the time comparison among these methods when the number of tuples in the source partitions is 10^6 and the selectivity is 0.1. We can see that, when the scope increases, Direct's execution time increases linearly while the execution time of other methods keeps nearly constant. This is because Direct needs to recompute over all the needed source partitions while the other three methods use Helper view to avoid recomputation.

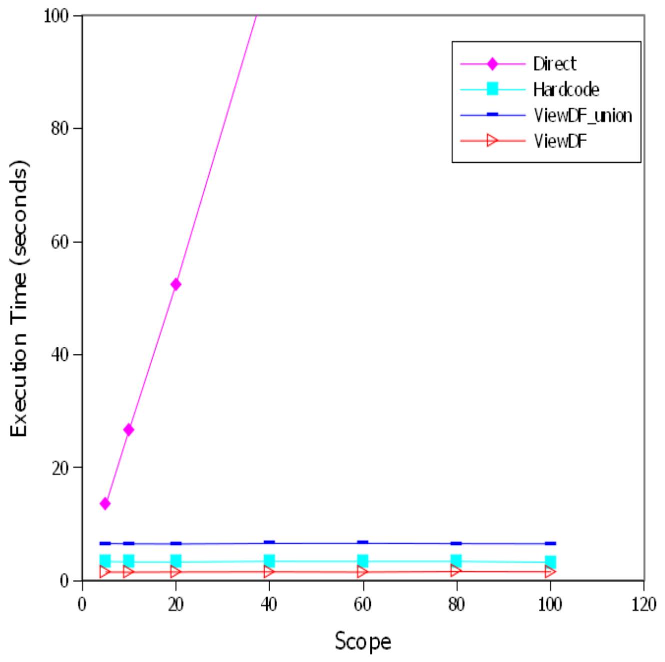


Figure 5.2: Pattern Matching: comparison between Direct, Hardcode, ViewDF_union and ViewDF under different scopes

Therefore, when the scope increases, Direct needs to recompute more source partitions. The experiment result can show the efficiency of the other three algorithms. We conclude that algorithms with incremental strategy can improve the performance.

Selectivity

We also conduct an experiment in which we compare these methods' execution time under different selectivity when the number of tuples in the source partition is 10^6 and the scope for Direct is 4. This experiment aims to test the overhead of the three algorithms with incremental strategy. The results are shown in Figure 5.3 which shows that execution time of each method increases as the selectivity increases. This is because we need to maintain more results in the Helper and materialized view. However, Direct increases much slower than the others. This is because when selectivity increases, Direct only maintains more results in the materialized view while other three algorithms need to maintain more results in both Helper and materialized view. The Helper view maintenance is the overhead of the other three algorithms. Therefore, when the selectivity equals to 0.7, ViewDF_union's

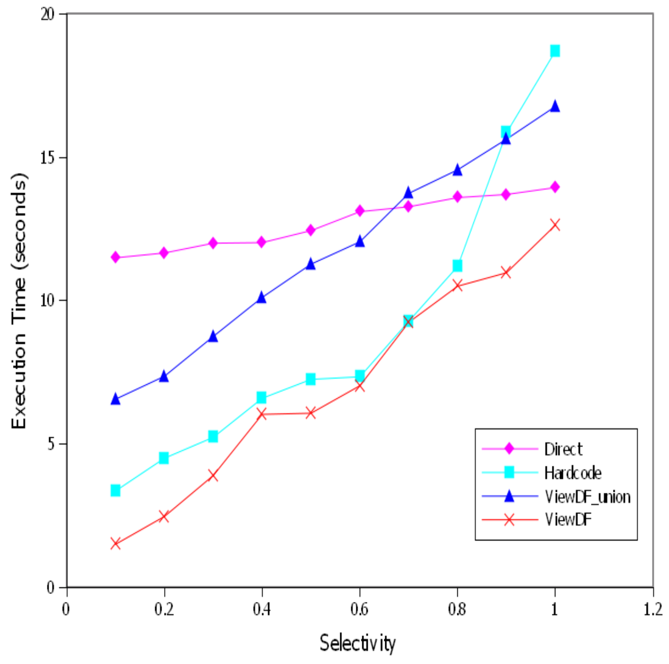


Figure 5.3: Pattern Matching: comparison between Direct, Hardcode, ViewDF_union and ViewDF under different selectivities

execution time is larger than Direct's, and when the selectivity equals to 0.9, Hardcode's execution time is larger than Direct's.

5.3 Sliding Window Aggregation

5.3.1 Algorithms

In order to compare different algorithms for sliding window aggregation, we first illustrate these algorithms using the two example queries provided in Section 4.2 (one is maintaining sum_loss and the other is maintaining max_loss in the materialized view).

Algorithms for sum_loss

1. **DataCell**: We implement the method of Liarou et al [45] by executing the ViewDFs for $aux3$ and $View3$ for the sum_loss query and call it "DataCell_Sum", since this

method is built on the DataCell architecture. The specific procedures for DataCell_Sum for the sum_loss query are: split the window into 60 partitions, get the sum of the loss for each partition and store the pre-aggregation results in the auxiliary partitions, and then generate one partition of $View3$ by merging the results in the auxiliary partitions. The reason that we must do pre-aggregation for each source partition is that there are more than one tuple for each src/dest pair in each source partition.

2. **Direct_Sum**: The implementation of Direct_Sum for the sum_loss is almost the same as DataCell_Sum except that Direct_Sum does not store the pre-aggregation results in the auxiliary partitions. Therefore, every time a new materialized view partition needs to be generated, Direct_Sum needs to recompute the sum of the loss for the previous source partitions while DataCell_Sum can use the existing pre-aggregation results in the auxiliary partitions.

3. **Prefix**: We implement the method of Arasu et al [9] by executing the ViewDFs for $aux4$ and $View4$ for the sum_loss query and call it "Prefix", because sum is a subtractable aggregation function and we store the prefix synopses in the auxiliary partitions denoted as $aux[i]$. Therefore, every time a new materialized view partition needs to be generated, Prefix can compute view[i] by subtracting $aux[i - win\_size]$ from $aux[i]$ instread of recomputing from scratch.

4. **ViewDF_Sum**: We call our optimization for the sum_loss query "ViewDF_Sum" and execute the ViewDFs for $aux5$ (the same as $aux3$) and $View5$. The specific procedures are: instead of accessing the most recent 60 minutes of data in S and recomputing the total loss for each source-destination pair, we take the total packet loss computed over the previous window, add the loss from the current minute, and subtract the oldest loss measurement from the previous window which is outside the current window.

5. **Hardcode_Sum**: We use JDBC to connect to PostgreSQL and implement our optimization "ViewDF_Sum". We maintain the *sum_loss* in the auxiliary partition $aux[i]$. Every time a new materialized view needs to be generated, we generate three cursors to point to the current auxiliary partition $aux[i]$, the expired auxiliary partition $aux[i-win\_size]$ and the previous materialized view partition $view[i-1]$. Then we use the cursor-pointed data to generate $view[i]$ by adding $view[i-1]$ with the loss from $aux[i]$ and subtracting the loss from $aux[i-win\_size]$. The specific rules for Hardcode_Sum algorithm are as follows. For a src-dest pair tuple t in $aux[i]$,

(a) If t exists in $aux[i - win\_size]$ and $view[i-1]$, then we add $t.loss$ in $view[i-1]$ with $t.loss$ in $aux[i]$, subtract $t.loss$ in $aux[i - win\_size]$, and write the result to a file. After all the tuples in $aux[i]$ is processed, we copy the content in the file to $view[i]$.

(b) If t does not exist in $aux[i - win\_size]$ and $view[i-1]$, we directly write the tuple's value in to the file.

Algorithms for max_loss

1. **DataCell_Max**: We implement the method of Liarou et al in [45] by executing the ViewDFs for $aux6$ and $View6$ for the max_loss query and call it "DataCell_Max". The specific procedures for DataCell_Max for the max_loss query are: split the window into 60 partitions, get the max of the loss for each partition and store the pre-aggregation results in the auxiliary partitions, and then generate one partition of $View6$ by merging the results in the auxiliary partitions.

2. **Direct_Max**: The implementation of Direct_Max for the max_loss is almost the same as DataCell_Max except that Direct_Max does not store the pre-aggregation results in the auxiliary partitions. Therefore, every time a new materialized view partition needs to be generated, Direct_Sum needs to recompute the pre-aggregation results for the previous base partitions while DataCell_Max can use the existing partial results in the auxiliary partitions.

3. **Hardcode_Max**: We use JDBC to connect to PostgreSQL, implement the same logic as $View6$ but outside the database and call it "Hardcode_Max". Hardcode_Max maintains the pre-aggregation results for every basic window in the auxiliary partitions and an array max_loss[] consisting of the intermediate values for every src/dest pair's max_loss. Then Hardcode_Max scans every auxiliary partition and update the intermediate values in the max_loss[].

5.3.2 Experiment Results

We conduct the following experiments to compare different algorithms for sliding window aggregation to maintain sum_loss and max_loss in the materialized view. All the experiments are based on the two example queries in Section 4.2. All the results are average values after we run each experiment 50 times.

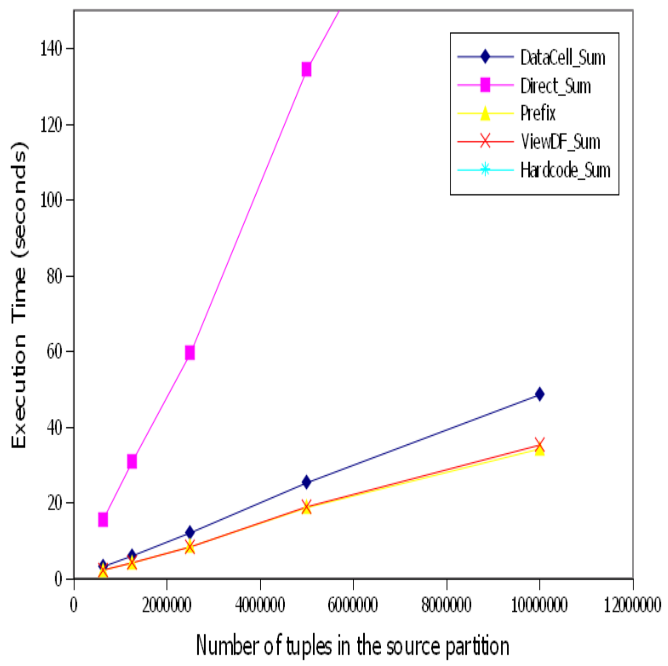


Figure 5.4: Sliding Window Operation for sum_loss for non-first window: comparison between DataCell_Sum, Direct_Sum, Prefix, ViewDF_Sum and Hardcode_Sum under different number of tuples in the source partition

Scalability for Sum

In this experiment, we implemented sliding window aggregation for sum_loss by DataCell_Sum, Direct_Sum, Prefix, ViewDF_Sum and Hardcode_Sum. We compare these algorithms' execution time to show ViewDF_Sum's scalability. Figure 5.4 shows the execution time comparison among these methods under different number of tuples in the source partition when the window size is 10. Figure 5.4 shows that the execution time of each method increases linearly when the number of tuples in the source partitions increases. Direct_Sum's execution time is much larger than all the other methods, because it needs to recompute the sum of the loss for the previous source partitions while others can use the existing pre-aggregation results in the auxiliary partitions. The execution time for Prefix, ViewDF_Sum and Hardcode_Sum is almost the same while DataCell_Sum's execution is larger than these three. This is because DataCell_Sum needs to merge the results in all of the 10 auxiliary partitions while other three only needs to access two or three auxiliary or materialized partitions. In addition, from Figure 5.4, we can see that, when the number of

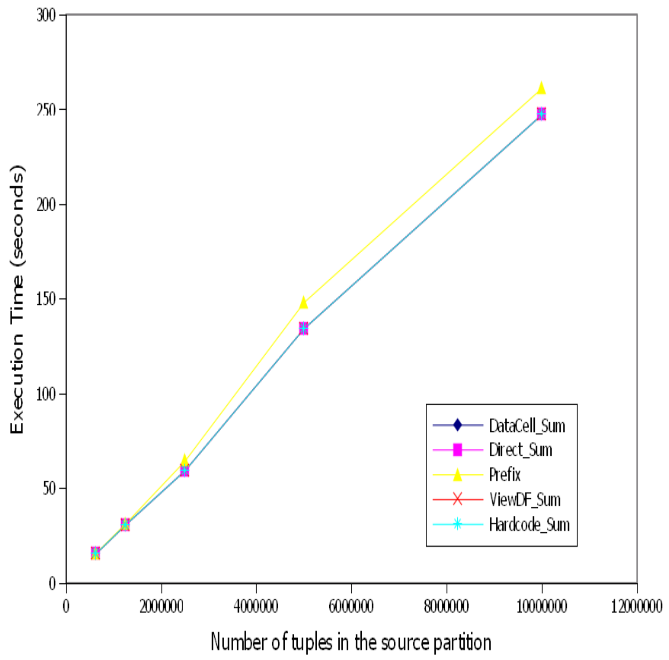


Figure 5.5: Sliding Window Operation for sum_loss for first window: comparison between DataCell_Sum, Direct_Sum, Prefix, ViewDF_Sum and Hardcode_Sum under different tuples

tuples in source table is 10^7 (10^6 different source-destination pairs with each pair consists of 10 tuples), the execution time for ViewDF_Sum is only 34.39 seconds, which is quite efficient.

The execution time for different methods in Figure 5.4 is for the non-first window aggregation. Sum_loss in the first window is computed from the INITIALIZE statement in the ViewDFs, while sum_loss in the non-first window is computed from the UPDATE statement in the ViewDFs. Therefore, the execution time of the first and non-first window is different. The execution time for the first window aggregation are shown in Figure 5.5. The execution time of all the methods for the first window aggregation is far larger than for the non-first window aggregation. This is because the first window aggregation needs to access all the 10 source partitions while the non-first window aggregation needs to access much smaller number of partitions (Prefix, ViewDF_Sum and Hardcode_Sum need to access two or three auxiliary or materialized view partitions. DataCell_Sum needs to access 10 auxiliary partitions instead of source partitions). One exception is Direct_Sum, whose execution time for the first window aggregation is the same as the non-first window aggregation, because both needs to access 60 source partitions. The execution time of

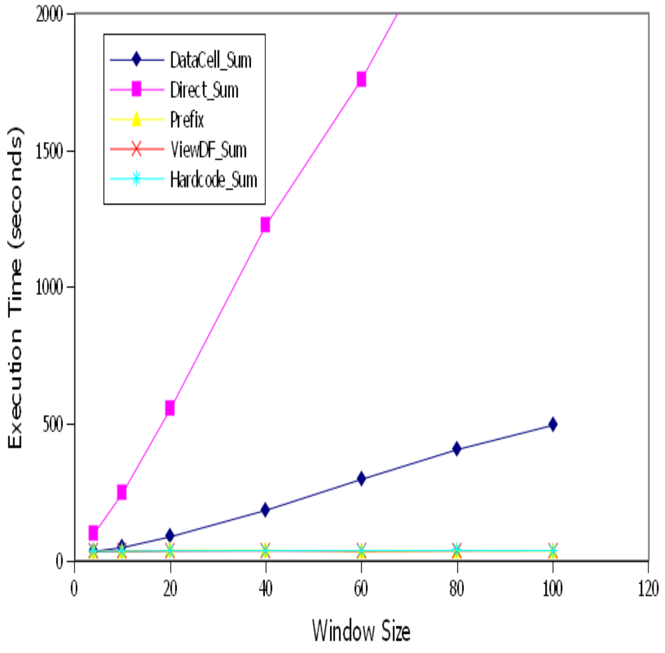


Figure 5.6: Sliding Window Operation for sum_loss: comparison between DataCell_Sum, Direct_Sum, Prefix, ViewDF_Sum and Hardcode_Sum under different window sizes

DataCell_Sum, Direct_Sum, ViewDF_Sum and Hardcode_Sum is the same, because they use the same execution statements. The execution time of Prefix is a little larger than others because Prefix maintains the prefix auxiliary partitions for the source partitions (each prefix auxiliary partition's computation needs to base on a source partition and the previous auxiliary partition) while other methods maintain computes an auxiliary partition only based on one source partition.

Window Size for Sum

We then compare the execution time of the above methods (DataCell_Sum, Direct_Sum, Prefix, ViewDF_Sum and Hardcode_Sum) under different window sizes when the number of tuples in the base partition is 10^7. The results are shown in Figure 5.6. We can see that as the window size increases, the execution time of Direct_Sum and DataCell_Sum increases almost linearly while other three methods' execution times are nearly constant. This is because both Direct_Sum and DataCell_Sum need to access the most recent win_size partitions and recompute the result while other methods only need to access two or three

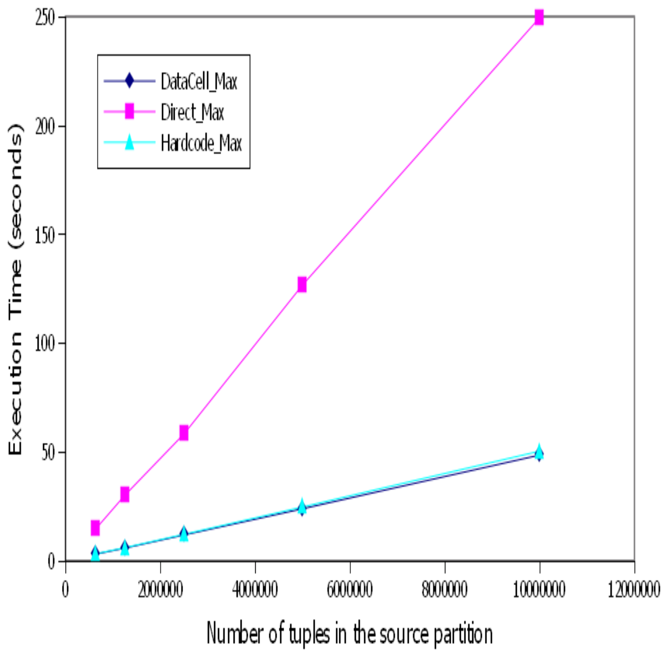


Figure 5.7: Sliding Window Operation for max_loss: comparison between DataCell_Max, Direct_Max and Hardcode_Max under different tuples

auxiliary or materialized view partitions. The execution time of Direct_Sum is larger than DataCell_Sum's because Direct_Sum needs to recompute the sum of the loss for the previous source partitions while DataCell_Sum can use the existing pre-aggregation results in the auxiliary partitions. Note that all the execution time in Figure 5.6 is for non-first window aggregation.

Scalability for Max

In this experiment, we implemented the sliding window aggregation for max_loss by DataCell_Max, Direct_Max and Hardcode_Max. Figure 5.7 shows the execution time comparison among these methods under different number of tuples in the source partition when the window size is 10. The execution time of each method increases linearly when the number of tuples in the source partitions increases. Direct_Max's execution time is much larger than the other two methods, because Direct_Sum needs to recompute the max of the loss for the previous source partitions while others can use the existing pre-aggregation results in the auxiliary partitions. The execution time for Direct_Max and Hardcode_Max is al-

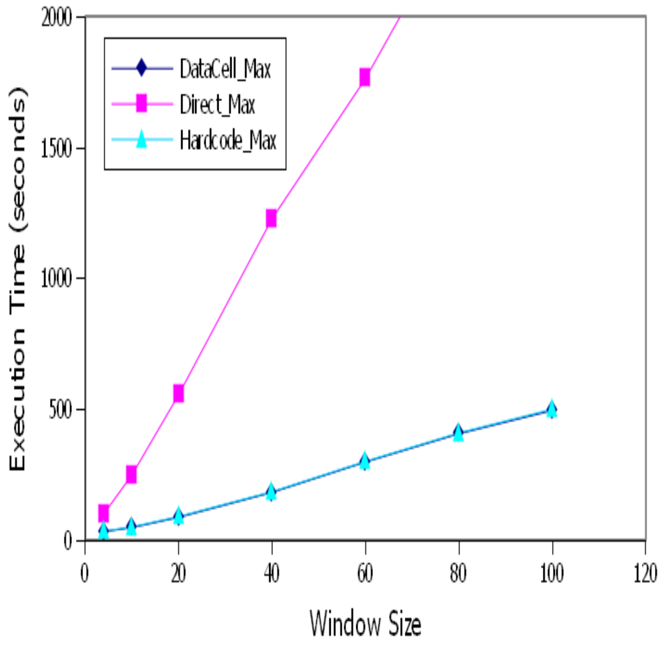


Figure 5.8: Sliding Window Operation for max_loss: comparison between DataCell_Max, Direct_Max and Hardcode_Max under different window sizes

most the same, because they execute the same statements. In addition, when the number of tuples in source table is 10^7, the execution time for ViewDF_Sum is only 48.68 seconds, which is quite efficient.

The execution time for different methods in Figure 5.7 is for the non-first window aggregation. Since all three methods use the same execution statements for the first window, their execution time for the first window aggregation is the same. Since there is no difference between the execute time of Direct_Max in the first window and non-first window aggregation, all the three methods' execution time can be seen from the execution time of Direct_Max in Fig. 5.7.

Window Size for Max

We then compare the execution time of the above methods (DataCell_Max, Direct_Max and Hardcode_Max) under different window sizes when the number of tuples in the base partition is 10^7. The results are shown in Figure 5.8. As the window size increases, the execution time of all the methods increases linearly, because all the methods need to access the most recent *win_size* partitions and recompute the result. The execution time of Direct_Max is larger than the other two because Direct_Max needs to recompute the max of the loss for the previous source partitions while others can use the existing pre-aggregation results in the auxiliary partitions. Note that all the execution time in Figure 5.8 is for non-first window aggregation.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we presented ViewDF: a flexible framework for incremental maintenance of materialized views in SDW systems that generalizes existing techniques and enables new optimizations for views defined with operators that are common in stream analytics. We proposed a special view definition (ViewDF) to enhance the traditional way of creating views in SQL by being able to specifying the contents of new view partitions over time and reference any partition of any table. We show that the declarative approach can naturally express incremental maintenance algorithms for relational queries as well as other types of queries that frequently appear in stream analytics such as sliding window aggregates and pattern matching. We implemented a prototype system based on this idea, which allows users to write ViewDFs directly and can automatically translate a broad class of users' normal queries into ViewDFs. Experiments show that our proposed system can greatly improve view maintenance time.

6.2 Future Work

There are some interesting directions for future work:

1. In the thesis, we assume that data arrives in order of timestamp. However, in a stream environment, it is very often that data arrives out-of-order. In the future, we want to develop techniques to handle out-of-order data.

2. One interesting direction for future work is to develop multi-query optimization strategies for ViewDFs so that related views may be updated together more efficiently than one-by-one.

3. Another direction is to extend the ViewDF framework to views partitioned on multiple attributes, not just time.

4. Currently, we have different methods for pattern matching and sliding window aggregation. In the future, we want to develop a cost-based optimizer to select the best method for a specific configuration.

5. In a distributed environment, naive materialized view maintenance approach has to store needed source partitions together on one machine so that it is not necessary to fetch source partitions from other machines to compute the new materialized view partition. ViewDF only needs to ensure that the necessary partitions are stored together on one machine, which is much easier to achieve. For example, in order to compute the sum of loss for each source-destination pair in a sliding window of 60 minutes, naive approach has to store all the 60 source partitions together. ViewDF only needs to store the expired source partition $S[i-60]$, previous materialized view $View[i-1]$ and the latest source partition $S[i]$ together on one machine. From this, we can see ViewDF framework is better suited for distributed environment. In the future, we want to deploy our setting into distributed environments and explore other strategies to improve the efficiency of distributed query processing.

References

[1] http://blogs.msdn.com/b/streaminsight/.

[2] http://dev.mysql.com/doc/refman/5.5/en/adding-functions.html.

[3] http://docs.oracle.com/cd/e11882_01/server.112/e25523/*part_warehouse*.htm.

[4] http://www.postgresql.org/.

[5] http://www.streambase.com/.

[6] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 147–160, 2008.

[7] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *Proceedings of the VLDB Endowment*, 5(10):968–979, 2012.

[8] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

[9] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 336–347, 2004.

[10] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proceedings of the 20th International Conference on Data Engineering*, pages 350–361, 2004.

[11] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems*, 29(3):545–580, 2004.

[12] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, et al. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[13] M. Balazinska, Y.C. Kwon, N. Kuchta, and D. Lee. Moirae: History-enhanced monitoring. In *Proceedings of the Third Conference on Innovative Data Systems Research*, pages 375–386, 2007.

[14] J.A. Blakeley, P.A. Larson, and F.W. Tompa. Efficiently updating materialized views. *ACM SIGMOD Record*, 15(2):61–71, 1986.

[15] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 479–490, 2006.

[16] Ahmet Bulut and Ambuj K Singh. Swat: Hierarchical stream summarization in large networks. In *Proceedings of the 19th International Conference on Data Engineering*, pages 303–314, 2003.

[17] Ahmet Bulut and Ambuj K Singh. A unified framework for monitoring data streams in real time. In *Proceedings of the 21st International Conference on Data Engineering*, pages 44–55, 2005.

[18] B. Cadonna, J. Gamper, and M.H. Böhlen. Sequenced event set pattern matching. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 33–44, 2011.

[19] B. Chandramouli, J. Goldstein, and S. Duan. Temporal analytics on big data for web advertising. In *Proceedings of the 28th International Conference on Data Engineering*, pages 90–101, 2012.

[20] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S.R. Madden, F. Reiss, and M.A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 668–668, 2003.

[21] Sirish Chandrasekaran. *Query processing over live and archived data streams.* 2005.

[22] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proceedings of International Conference on Extending Database Technology*, pages 627–644. Springer, 2006.

[23] Alan Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, Walker M White, et al. Cayuga: A general purpose event monitoring system. In *Proceedings of the Third Innovative Data Systems Research*, pages 412–422, 2007.

[24] N. Dindar, P.M. Fischer, M. Soner, and N. Tatbul. Efficiently correlating complex events over live and archived data streams. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System*, pages 243–254, 2011.

[25] N. Dindar, B. Güç, P. Lau, A. Ozal, M. Soner, and N. Tatbul. Dejavu: declarative pattern matching over live and archived streams of events. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 1023–1026, 2009.

[26] Luping Ding, Nishant Mehta, Elke Rundensteiner, and George Heineman. Joining punctuated streams. In *Proceedings of International Conference on Extending Database Technology*, pages 519–520. Springer, 2004.

[27] M.J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre. Continuous analytics: Rethinking query processing in a network-effect world. In *Proceedings of the 4th Conference on Innovative Data Systems Research*, 2009.

[28] Thanaa M Ghanem, Moustafa A Hammad, Mohamed F Mokbel, Walid G Aref, and Ahmed K Elmagarmid. Incremental evaluation of sliding-window queries over data streams. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):57–72, 2007.

[29] T.M. Ghanem, A.K. Elmagarmid, P.Å. Larson, and W.G. Aref. Supporting views in data stream management systems. *ACM Transactions on Database Systems*, 35(1):1, 2010.

[30] L. Golab, T. Johnson, J.S. Seidel, and V. Shkapenyuk. Stream warehousing with datadepot. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 847–854, 2009.

[31] L. Golab, T. Johnson, S. Sen, and J. Yates. A sequence-oriented stream warehouse paradigm for network monitoring applications. In *Proceedings for Passive and Active Measurement*, pages 53–63, 2012.

[32] L. Golab, T. Johnson, and V. Shkapenyuk. Scalable scheduling of updates in streaming data warehouses. *IEEE Transactions on Knowledge and Data Engineering*, 24(6):1092–1105, 2012.

[33] Lukasz Golab and Theodore Johnson. Consistency in a stream warehouse. In *Proceedings of the 5th Conference on Innovative Data Systems Research*, pages 114–122, 2011.

[34] Lukasz Golab and M Tamer Özsu. Data stream management. *Synthesis Lectures on Data Management*, 2(1):1–73, 2010.

[35] Rick Greer. Daytona and the fourth-generation language cymbal. *ACM SIGMOD Record*, 28(2):525–526, 1999.

[36] A. Gupta and I.S. Mumick. *Materialized views: techniques, implementations, and applications*. MIT Press, 1999.

[37] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immerman. On supporting kleene closure over event streams. In *Proceedings of the 24th International Conference on Data Engineering*, pages 1391–1393, 2008.

[38] Dirk Habich, Wolfgang Lehner, and Michael Just. Materialized views in the presence of reporting functions. In *Proceedings of the 18th International Conference on Scientific and Statistical Database Management*, pages 159–168, 2006.

[39] J-H Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Proceedings of the 21st International Conference on Data Engineering*, pages 779–790, 2005.

[40] Sailesh Krishnamurthy, Michael J Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. Continuous analytics over discontinuous streams. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 1081–1092, 2010.

[41] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *ACM SIGMOD Record*, 34(1):39–44, 2005.

[42] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of*

ACM SIGMOD International Conference on Management of Data, pages 311–322, 2005.

[43] Ming Li, Mo Liu, Luping Ding, Elke A Rundensteiner, and Murali Mani. Event stream processing with out-of-order data arrival. In *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, pages 67–67, 2007.

[44] E. Liarou and M.L. Kersten. Datacell: Building a data stream engine on top of a relational database kernel. In *Proceedings of the 35th International Conference on Very Large Data Bases PhD Workshop*, 2009.

[45] Erietta Liarou, Stratos Idreos, Stefan Manegold, and Martin Kersten. Enhanced stream processing in a dbms kernel. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 501–512, 2013.

[46] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 802–813, 2002.

[47] K.A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. *ACM SIGMOD Record*, 25(2):447–458, 1996.

[48] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. Expressing and optimizing sequence queries in database systems. *ACM Transactions on Database Systems*, 29(2):282–318, 2004.

[49] H. Wang and C. Zaniolo. Atlas: A native extension of sql for data mining. In *Proceedings of the 3rd SIAM International Conference on Data Mining*, pages 130–141, 2003.

[50] Andrew Witkowski, Srikanth Bellamkonda, Hua-Gang Li, Vince Liang, Lei Sheng, Wayne Smith, Sankar Subramanian, James Terry, and Tsae-Feng Yu. Continuous queries in oracle. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 1173–1184, 2007.

[51] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 407–418, 2006.

[52] Y. Xing, S. Zdonik, and J.H. Hwang. Dynamic load distribution in the borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering*, pages 791–802, 2005.

[53] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *Proceedings of the 17th International Conference on Data Engineering*, pages 51–60, 2001.