Master’s Thesis

Modeling and Efficiently Processing Hybrid Pattern Matching Queries over Live and Archived Streams

by
Merve Soner
msoner@student.ethz.ch

Advisers
Prof. Nesime Tatbul
Nihal Dindar

June 2010

ETH Zurich
Department of Computer Science
Systems Group
Abstract

Integrating pattern matching functionality over live and archived streams of events with hybrid queries has become very crucial for various complex event processing (CEP) applications including financial market data analysis and RFID-based asset tracking. Hybrid queries allow us to verify current live events, analyze archived events or even make predictions about future event occurrences. Although the semantics and requirements of hybrid queries for pattern matching over live and archived events differ for various applications, they have some common features like archiving live events, and accessing and processing relevant chunks of archived events.

The first goal of this thesis is to define a clean semantics of hybrid queries for pattern matching over live and archived streams which can be generalized to many CEP applications. The second goal of this thesis is to propose algorithms to process hybrid queries with low latency in spite of the I/O cost of archived stream access and overhead of processing archived streams. We apply our solutions on DejaVu\cite{7} CEP engine by extending its architecture to efficiently store and process both archived streams and previous match results. Additionally, two different workload-sensitive archived stream processing strategies, lazy and eager, are proposed to be used with different application requirements and data rates. With the experiments, a clear improvement in the performance provided by the extended architecture is demonstrated. Moreover, processing archived streams in lazy or eager manner are compared under different scenarios and the winning strategy for each scenario is examined.
Acknowledgements

I would like to thank Prof. Nesime Tatbul for her valuable support and guidance through this thesis. I am grateful to Nihal Dindar for her advices and assistance with big patience and encouragement.

Finally, I am very thankful to my family for their unconditional love and support during my studies. This work would not have been possible without them.
Contents

1 Introduction \hspace{1.5cm} 6
  1.1 Motivation \hspace{1.5cm} 6
  1.2 Problem Statement and Scope \hspace{1.5cm} 6
  1.3 Related Work \hspace{1.5cm} 7
  1.4 Contribution \hspace{1.5cm} 8

2 Background \hspace{1.5cm} 9
  2.1 DejaVu Complex Event Processing System \hspace{1.5cm} 9
    2.1.1 Overview \hspace{1.5cm} 9
    2.1.2 Query Language \hspace{1.5cm} 9
    2.1.3 Architecture \hspace{1.5cm} 10
    2.1.4 Environment \hspace{1.5cm} 11
  2.2 Input & Output Representation \hspace{1.5cm} 12
    2.2.1 Primitive Events \hspace{1.5cm} 12
    2.2.2 Complex Events \hspace{1.5cm} 13

3 Modeling Hybrid Queries \hspace{1.5cm} 14
  3.1 Use Case \hspace{1.5cm} 14
  3.2 Semantics \hspace{1.5cm} 15
    3.2.1 History of an Event \hspace{1.5cm} 16
    3.2.2 Time Distance Between Two Events \hspace{1.5cm} 16
  3.3 Representation \hspace{1.5cm} 17
  3.4 Definitions \hspace{1.5cm} 18

4 Architectural Extensions \hspace{1.5cm} 21
  4.1 Hierarchical Storage \hspace{1.5cm} 22
    4.1.1 Motivation \hspace{1.5cm} 22
    4.1.2 Components \hspace{1.5cm} 24
    4.1.3 Implementation \hspace{1.5cm} 25
    4.1.4 Size of The Recent Buffer \hspace{1.5cm} 27
  4.2 Historical Result Cache \hspace{1.5cm} 27
    4.2.1 Motivation \hspace{1.5cm} 27
    4.2.2 Data Access \hspace{1.5cm} 29
List of Figures

2.1 MATCH-RECOGNIZE Syntax .................................. 10
2.2 DejaVu System Architecture ................................. 11
2.3 Events .......................................................... 12

3.1 Use Case ......................................................... 15
3.2 Time Distance Between Two Events ......................... 16
3.3 Creating a DStream Table with Its Archive ............... 17
3.4 Hybrid Query .................................................. 18

4.1 Extended DejaVu Architecture ............................... 22
4.2 Hierarchical Storage .......................................... 24
4.3 Creating a DStream Table ...................................... 24
4.4 Creating a DArchive Table .................................... 25
4.5 Historical Result Cache ....................................... 28
4.6 General Flow .................................................. 33

5.1 Activity Diagram in Lazy Mode ............................... 35
5.2 Activity Diagram in Eager Mode .............................. 36
5.3 Tuple Processing Time Analysis .............................. 38

6.1 Performance Evaluation of Archiving Live Events ........ 41
6.2 Query 1 (Simple Archive Pattern) .......................... 43
6.3 Query 2 (Complex Archive Pattern) ......................... 44
6.4 Experiment Details ............................................. 45
6.5 Simple Query Performance Evaluation ...................... 46
6.6 Complex Query Performance Evaluation .................... 47
6.7 Simple and Complex Query Throughput Ratio Comparison . 48
6.8 Memory Consumption .......................................... 49
6.9 Average Response Time (Freq = 0.1) ....................... 52
6.10 Throughput (Freq = 0.1) .................................... 52
6.11 Average Response Time (Freq = 1) ......................... 53
6.12 Throughput (Freq = 1) .................................... 53
6.13 Average Response Time (Freq = 10) ....................... 54
6.14 Throughput (Freq = 10) .................................... 54
Chapter 1

Introduction

1.1 Motivation

With the availability of large volumes of data generated by sensing devices, monitoring applications, financial services etc., complex event processing (CEP) over live streams of events has become very crucial. Complex event processing applications like RFID-based asset tracking, health monitoring, financial market data analysis, etc. require complex pattern analysis on the sequences of data. For these applications, it is also important to run hybrid pattern matching queries over live and archived streams for correlation, verification, analysis or forecasting. For example, an interesting scenario for a health monitoring system is to correlate current blood test results in the case of an abnormal pattern with the results of previous months to check whether a similar behaviour has been observed before or not.

1.2 Problem Statement and Scope

Integrated processing of live and archived events is a key requirement for many complex event processing applications. However, integrating live events with all archived events might be very costly and time consuming if the size of archived events is very large. In most cases, some portion of archived events is considered to be more relevant and desirable compared to whole archived events. For instance in the health monitoring system, archived events belonging to the last few months or years is more relevant than the older archived events. In order to query live and archived events with low latency, relevant portion of the archived events should be accessed and processed according to the application requirements.

Even if the amount of archived events to be accessed is reduced, it is still challenging to run hybrid queries with low latency due to the I/O cost of data access and processing cost of archived events.
The main goal of this thesis is to define a clean semantics of hybrid queries that require integrated processing of live and recent archived events and propose algorithms to process hybrid queries over live and archived streams in an efficient manner. We apply our solutions on DejaVu CEP engine which integrates declarative pattern matching over live and archived streams of events on top of a novel system architecture[7]. We extend the current DejaVu architecture to customize storing, accessing, and processing of archived streams and to optimize hybrid query processing.

1.3 Related Work

Cayuga [6], SASE+ [15, 11] and ZStream [12] are complex event processing engines that focus on high-performance pattern matching over live streams of events. Each system proposes its own custom pattern matching language to define patterns over live events but none of them supports archiving of live streams or hybrid queries over live events and historical data. On the other hand, DataDepot [8, 9, 4] is a stream warehousing system that is designed to automate the ingestion of streaming data from a wide variety of sources and to maintain complex materialized views over these sources. It provides the warehousing of streams but does not support combining live events with historical data or pattern matching over historical data.

The need for combining live events with historical data is identified by Balazinska et al.[3], Chandrasekaran et al.[5], Maier et al.[14] and Reiss et al.[13]. Due to the large volumes of historical data, these systems decrease the amount of historical data to be integrated with live events by using different approaches. Moirae is a general purpose event monitoring system that supports integration of historical data into a near-real-time monitoring system[3]. It prioritizes the recent historical data and produces an approximate set of results by using multi-level storage, recent event materialization, and context similarity metric. The context differs for different event types. For example, it might be the state of the system at a system failure or a set of running processes at a system overload. Moirae gives approximate result set at first which includes the $k$ most similar past events and gives more accurate and complete results incrementally. Chandrasekaran et al. points out the need for accessing incoming streams and archived data at the same time and proposes a special disk access method called OSCAR[5]. This method addresses overload in hybrid query processing and applies a reduction technique on archived data by random sampling or window aggregations and gives a summarized result set. Different approaches(lazy, eager and hybrid) for the reduction of archived data are also discussed to speed up hybrid query processing. Latte is another system that combines live streams
with large data archives and introduces new approaches to support archive window scans[14]. It uses periodicity for live and historical event correlation. A corresponding past period that has the same characteristics with the current time period like the same day of the week or the same time of the year is defined to partition historical data into periods. Finally, Reiss et al. examines the usage of Bitmap Indices for storing and querying historical data with the help of update characteristics and query requirements of streaming applications[13]. TelegraphCQ streaming query processor is used to analyse streams and FastBit bitmap index is used to correlate current behaviours with historical trends. A set of representative queries for real-time network analysis is created consisting of stream processing and historical component. Historical part of the query limits the amount of the historical data to be accessed with a given time window or period value.

1.4 Contribution

Different from existing CEP engines, we focus on integrating pattern matching functionality over live and archived streams with hybrid queries. Hybrid query processing over live and historical data has been examined by various systems. Different from these systems, we define hybrid queries in a way to enable complex event correlation between live and archived events by finding similar patterns, etc., rather than providing historical look-ups. We focus on the applications that require access to recent archived events and give complete set of results in the specified recent history. Among the existing systems, Moirae[3] is the most similar one to our work by prioritizing recent historical data and we borrow their idea of storing most recent data in the memory. However, their motivation behind storing recent data in memory is to produce approximate results with the available data whereas we use it to build a hierarchical storage to archive and process recent streams efficiently. Apart from hierarchical storage, we provide caching of previously found matches over archived streams that might be used by future live events. Additionally, two different archived stream processing strategies, lazy and eager, are proposed to be used with different workloads and application requirements.

The main contribution of the thesis is to provide efficient hybrid pattern matching over live and archived streams with the introduction of new architectural components and different archived stream processing strategies.
Chapter 2

Background

2.1 DejaVu Complex Event Processing System

2.1.1 Overview

DejaVu is a complex event processing system that integrates declarative pattern matching over live and archived streams of events on top of a novel system architecture[7]. DejaVu is built on MySQL open-source database system[2]. Query processing engine of MySQL has been extended with FSM-based plans to run pattern matching queries. Besides, two new storage engines (Live and Archived Stream Stores) have been introduced to run one-time, continuous, and hybrid queries on live and archived streams.

2.1.2 Query Language

DejaVu’s query language is based on the SQL-based declarative pattern matching language standard proposed in [16]. This language expresses a pattern query with MATCH-RECOGNIZE clause using the syntax in Figure 2.1.

MATCH-RECOGNIZE clause has input specification (section 1 in Figure 2.1), pattern definition (section 3 in Figure 2.1), future pattern start specification (section 4 in Figure 2.1), and output specification parts (section 2 in Figure 2.1). PARTITION BY and ORDER BY clauses partition and/or order the input data with a given set of attributes before performing the pattern matching query. In pattern definition section, patterns are defined as regular expressions with PATTERN and DEFINE keywords and match mode is specified with MAXIMAL or INCREMENTAL keywords. Future pattern start specification section of MATCH-RECOGNIZE clause specifies the starting point of the next pattern search after a match is found. Finally, output specification section specifies the output columns and the number of rows to be reported per match.
The original query language proposal[16] is designed for sequence tables but it is extended to streams for DejaVu CEP engine.

2.1.3 Architecture

The main components of DejaVu Architecture (Figure 2.2) are as follows:

- **Input Adapter** is responsible for inserting input streams to Live and Archived Stream Stores. It reads data from different sources like a file, RFID readers etc. and inserts it to DStream. If there is an archive attached to this stream, the tuples are also inserted to DArchive.

- **Live Stream Store (DStream)** is an in-memory storage engine which accepts push-based inputs. It acts like a tuple queue and provides pull and push modes for the query processor. It is implemented to answer continuous pattern matching queries.

- **Archived Stream Store (DArchive)** is an optional stream store to archive live events. It is an append-only persistent storage engine and designed for one-time and hybrid queries. It can also be used as a backup when a failure occurs in live stream store.

- **Query Processing Engine** is the core component of DejaVu which extends regular MySQL query processing engine with pattern matching functionality. The pattern in the query is represented by Finite State

### Figure 2.1: MATCH-RECOGNIZE Syntax

```sql
SELECT <select-list>
FROM <table-name> MATCH_RECOGNIZE (
  PARTITION BY <field-name>
  ORDER BY <field-name>
  MEASURES <measure-list>
  MATCH_NUMBER
  CLASSIFIER
  ONE/ALL ROW PER MATCH
  MAXIMAL / INCREMENTAL MATCH
  AFTER MATCH SKIP TO NEXT ROW/ PAST LAST ROW /...
  PATTERN (.....)
  DEFINE <define-alphabets>
)
```

The syntax allows for pattern recognition within the SELECT clause, providing a flexible way to process and analyze stream data.
Machine (FSM) with a set of states and edges. Each FSM may have multiple active states due to the non-determinism and overlapping semantic windows. Active states of the FSM share input tuples via input holders where different partitions of the input tuples reside. Input tuples are placed into the correct input holder in an efficient manner via router component of QP. The query processing engine accesses the input tuples in either pull or push mode by making use of pluggable storage engine API. Besides, query processor has the ability to switch between pull and push input handling modes with respect to the data input and processing rate.

- *DejaVu Client* displays the results reported by DejaVu. It can be a regular MySQL client or an application for a specific use case.

![DejaVu System Architecture](image)

**Figure 2.2: DejaVu System Architecture**

### 2.1.4 Environment

DejaVu is built on mysql-6.0.3-alpha version of MySQL. The implementation is developed in C/C++ as an addition to MySQL source code. GCC version 4.2.4 is used as a compiler with Ubuntu 8.04 distribution of Linux operating system.
2.2 Input & Output Representation

Live events originating from different sources like RFID readers, TCP/IP connections or even DejaVu CEP System can be given to DejaVu CEP System as input as long as they satisfy the corresponding data schema. The data schemas of live events differ with respect to the source of the events and consist of a unique time-stamp pair \(< start, end >\) and a set of attributes. Live events are assumed to arrive totally ordered by time. Since each event has a time-stamp pair, they are first ordered by their start times and then the events with the same start time are ordered by their end times. For instance, the events in Figure 2.3 are ordered as \(e_0, e_1, e_2, e_3, e_4, e_5, e_6,\) and \(e_7\).

The output of DejaVu CEP system are the events having the same or different schema with the input events. They are also totally ordered by time. Input and output events of DejaVu can be either primitive or complex events.

2.2.1 Primitive Events

Primitive events represent the events which happen at a point in time like \(e_0, e_5,\) and \(e_7\) in Figure 2.3. Primitive events have identical start and end time-stamps.
2.2.2 Complex Events

Complex events represent the events that happen within a time period like $e_1$, $e_2$, $e_3$, $e_4$, and $e_6$ in Figure 2.3. Complex events consists of some or all fields of consecutive primitive events as well as extra aggregation fields on them. A complex event has different start and end time-stamps which are respectively the time-stamps of the first and last primitive events which contributed to that complex event.
Chapter 3
Modeling Hybrid Queries

Hybrid queries are used by various CEP applications to integrate live and archived streams for taking some statistics, performing analysis or even making predictions about future event occurrences. Although the semantics and requirements of hybrid queries vary with respect to the purpose of the application and use cases scenarios, they have some features in common like archiving live events, accessing and processing relevant chunks of archived events, etc. In this chapter, we will first describe a use case where hybrid queries can be used and later define the semantics of hybrid queries which can be generalized to different use cases with similar requirements.

3.1 Use Case

One of the complex event processing applications that require pattern matching over a sequence of data is financial data market analysis. A particular stock might be tracked for different types of events like increase, decrease or stability in the price. Even more, a user may want to query archived streams when a specific event occurred on live streams in order to see the recent behaviours of this stock and take some actions. In this case, a hybrid query should be defined over live stream and its archive in order to see query results with low latency. Suppose that there is a live stream of financial dataset with the schema \((t_s, t_e, \text{Symbol}, \text{Price})\). Let the user want to detect a tick-shaped pattern in price (a fall in the price followed by an increase that has a larger value than the beginning price of the fall) of archived streams whenever a decrease in price is detected on live streams of Stock A as seen in Figure 3.1 in order to predict whether Stock A could bring benefit in the near future or not.
Using the hybrid query, the user can specify the patterns over live and archived streams as well as the columns to be shown in the result set. The user can also apply some aggregation functions over match results to get minimum and/or maximum value of the result set or the value of first and/or last tuples.

In the remaining parts of the thesis, we will stick to this use case for some optimizations and testing. However, the solutions we offer are flexible and can be used for other use cases with tiny modifications.

### 3.2 Semantics

As discussed in Section 1.2, recent archived streams are considered to be more relevant compared to older archived streams in many CEP applications. For example, in our use case, the user would be interested in the behaviour of a particular stock within the last week or month.

Instead of giving summarized and/or approximated results over whole history, we define hybrid queries in a way that they provide complete set of historical events in the specified portion of recent history. Based on this definition, current and archived events are integrated by using time metrics. The user decides how much to go back in the history by defining a time distance between the live and archived streams. Since processing all of the archived streams for each newly detected event is costly and unnecessary in many situations, this feature brings a crucial advantage in terms of query response time especially when the user does not want to go back too much in the history. Besides, the user still has the choice to go back to the beginning of the history by setting the time distance infinite. In the following
sections, the history of an event and the time distance between two events are explained under the given semantics.

3.2.1 History of an Event

The history of an event over a live stream of events starts when a hybrid query over that stream is registered to the system. Since hybrid queries define a time distance $t_d$ between live and archived events, the history of a live event is limited to the archived events that has started and finished before the live event and are not farther away than $t_d$ to the live event.

3.2.2 Time Distance Between Two Events

DejaVu is a complex event processing system and the output events are generally complex events which happen within a time interval rather than at a point in time. This feature brings an ambiguity during the definition of time distance between two events.

Let $event_1$ and $event_2$ be two events as shown in Figure 3.2 with time-stamps $<t_{s1}, t_{e1}>$ and $<t_{s2}, t_{e2}>$ respectively. As defined in Section 2.2, $event_1$ occurs before $event_2$ since $t_{s1} < t_{s2}$ but whether $event_1$ is in the history of $event_2$ or not depends on the time distance definition.

![Figure 3.2: Time Distance Between Two Events](image)

Since DejaVu orders events with their start timestamps, time distance could be defined on start timestamps of the events. Let the history of $event_2$ be the time interval $[history_s, history_e]$ and $|s|$ is the size of this interval. Then,

$$\begin{align*}
  history_s &= \begin{cases} 
  t_{s2} - t_d & \text{if } t_d < t_{s2} \\
  0 & \text{otherwise}
  \end{cases} \quad (3.1) \\
  history_e &= t_{e2} - 1 \quad (3.2) \\
  max(|s|) &= history_e - history_s + 1 \\
  &= (t_{e2} - 1) - (t_{s2} - t_d) + 1 \\
  &= t_d + (t_{e2} - t_{s2})
\end{align*}$$
As in equation 3.3, the maximum amount of archived streams to be accessed depends on the length of the live event which is $t_{e2} - t_{s2}$ apart from $t_d$. The main purpose of defining time distance is to have control over the size of the history of a live event. If the length of the live pattern is included in time distance, we cannot estimate the size of history for varying size of live events. Thus, we define time distance as the difference between end time stamp of the live event and start time stamp of the archived event and modify the equation 3.1 as follows

$$history_s = \begin{cases} 
  t_{e2} - t_d & \text{if } t_d < t_{e2} \\
  0 & \text{otherwise}
\end{cases}$$  \hspace{1cm} (3.4)

Then the maximum size of the history will be

$$max(|s|) = history_e - history_s + 1 \hspace{1cm} (3.5)$$

$$= (t_{e2} - 1) - (t_{e2} - t_d) + 1$$

$$= t_d$$

### 3.3 Representation

In DejaVu, a hybrid query is represented as a join operation on at least two MATCH-RECOGNIZE clause over live and archived stream tables with optional join conditions. Time distance of the hybrid queries should not be considered as a join condition but a part of the MATCH-RECOGNIZE clauses since it is used during pattern matching.

Since the pattern matching query over live stream is continuous, the hybrid query is also continuous, i.e, it runs until the streaming is finished or the query is dropped.

Assume a DStream table LiveStock and its archive ArchiveStock which is a DArchive table are created for the use case explained in Section 3.1 as in Figure 3.3.

```sql
CREATE STREAM LiveStock (Symbol varchar(20), Timestamp time, Price double);
CREATE TABLE ArchiveStock (Symbol varchar(20), Timestamp time, Price double) live-LiveStock;
```

Figure 3.3: Creating a DStream Table with Its Archive

Then, the user should submit the hybrid query shown in 3.4 for the scenario in our use case (finding a decrease in price of live events and finding a tick
shape in price of archived events). Using the syntax explained in Section 2.1.2, the pattern specifications over live and archived streams are defined using MATCH-RECOGNIZE clause. For each table in the query, desired columns of important rows or aggregations over some/all rows can be reported using MEASURES clause. Finally, the join conditions over live and archived streams are specified in WHERE clause. After WHERE clause, desired time distance for the hybrid query should be given in a specific time unit using TDISTANCE clause.

```sql
SELECT symbol_1, start_1, end_1, start_a, end_a, min_price_1, min_price_a, max_price_1, max_price_a
FROM Livestock MATCH RECOGNIZE{
  PARTITION BY Symbol
  MEASURES A.Symbol AS symbol_1,
  A.Timestamp AS start_1,
  LAST(B.Timestamp) AS end_1,
  MIN(B.Price) AS min_price_1,
 tdist = 3600;
}
ON ROW PER MATCH
AFTER MATCH SKIP FIRST LAST ROW
INCREMENTAL MATCH
PATTERN (A B+)
DEFINE /* A matches any row */
  B AS (B.Price < A.Price AND B.Price <= PREV(B.Price))
}
ArchivedLivestock MATCH RECOGNIZE{
  PARTITION BY Symbol
  MEASURES A.Symbol AS symbol_a,
  A.Timestamp AS start_a,
  LAST(B.Timestamp) AS end_a,
  MIN(B.Price) AS min_price_a,
  MAX(B.Price) AS max_price_a
  tdist = 3600;
}
ON ROW PER MATCH
AFTER MATCH SKIP FIRST LAST ROW
INCREMENTAL MATCH
PATTERN (A B+ C+ D+)
DEFINE /* A matches any row */
  B AS (B.Price < A.Price AND B.Price <= PREV(B.Price))
  C AS (C.Price >= PREV(C.Price) AND C.Price <= A.Price)
WHERE symbol_1 = symbol_a
};
```

Figure 3.4: Hybrid Query

Please note that, no other field columns that are not listed in MEASURES clause can be used in SELECT or WHERE clauses. The reason for that is hybrid queries are used to correlate whole patterns in live and archived streams instead of correlating each row of these patterns. For the same reason, the output specification of each table in hybrid query must be ONE ROW PER MATCH instead of ALL ROWS PER MATCH.

### 3.4 Definitions

In this section, the definitions of the concepts that are used for hybrid query modeling and experiments are given.
**Definition 1** (Live Match). Live match refers to each newly detected pattern match over live streams with respect to the hybrid query specifications.

**Definition 2** (Archive Match). Archive match refers to each of the detected pattern matches over relevant archived streams with respect to the hybrid query specifications.

**Definition 3** (Response Time of a Live Match). The response time of a live match is the time elapsed from the availability of the last tuple contributing to the live match until the end of hybrid query processing of that match.

**Definition 4** (Frequency of a Match). In order to measure how often a match occurs on a data set where the matches are uniformly distributed, we define the frequency of a match $f_{\text{match}}$ as the number of matches that occur in unit time. The frequency of a match depends on the pattern specifications and the data set.

**Definition 5** (Average Access Frequency of Archived Streams). Different from live events, archived events may not be accessed or accessed more than once during hybrid query processing. In order to measure the amount of archived data that is not accessed or accessed more than once, we define average access frequency of archived streams which is the average count of accessing to an archived stream. For example, assume we have 5 archived tuples $t_1, t_2, t_3, t_4,$ and $t_5$ which are accessed for 0, 1, 2, 2, and 2 times respectively during hybrid query execution. Then, the average access frequency of these tuples is

$$\text{Average Access Frequency} = \frac{\text{total number of tuple accesses}}{\text{total number of tuples}} (3.6)$$

$$= \frac{0 + 1 + 2 + 2 + 2}{5} = \frac{1}{1.4}$$

If each tuple in the data set has been accessed exactly once, then the average access frequency will be 1. Larger values of average access frequency indicate more re-accesses over archived streams. Average access frequency depends on the frequency of matches over live events and the time distance of the hybrid query. Frequency of live matches determines how often archived stream access is required and time distance determines how many tuples from archived streams are required for each access.

**Definition 6** (Throughput). Throughput of the system is defined as

$$\text{Throughput} = \frac{|\text{live}|}{\text{time}_{\text{elapsed}}} (3.7)$$
where $|\text{live}|$ is the size of live events and $\text{time}_{\text{elapsed}}$ is the time elapsed to process hybrid query over all live events. Time elapsed to process the hybrid query starts when the first tuple is entered to DejaVu System and ends when the last tuple is processed both for live and archive pattern matching. The time spent for processing and archiving the events as well as preparing the query result is included in the measured time. The time spent for delivering output to the client is excluded as it depends on the client program and will be the same in our experiments.
Although pattern matching over live and archived streams have a lot in common like the functionality and language requirements, the performance issues and optimization techniques of two processes differ from each other. Hybrid queries require access to live and archived streams during the query execution. Moreover, a large portion of archived streams is accessed more than once. Thus, an efficient storage management is necessary in order not to slow down query processing over live stream and to have low latency. Besides, with the introduction of time distance feature, a history notification mechanism is required to reduce the amount of archived streams to be processed during hybrid query execution. In order to meet the new requirements, the current DejaVu architecture discussed in Section 2.1.3 is extended with a new hierarchical storage, historical result cache for archived streams and a history notification mechanism as shown in Figure 4.1. The general goal is to optimize hybrid queries which may include different pattern specifications and time distances and process various input streams with different match frequencies and input rates.
4.1 Hierarchical Storage

4.1.1 Motivation

With the introduction of hybrid queries, the importance of archiving live events and accessing these events have increased. DejaVu should return a complete set of results and have consistency among different instances of the same hybrid query on the same data set which requires a synchronization between Archived Stream Store and Live Stream Store. In other words, when a new event is detected over live stream with end time-stamp $t$, Archived Stream Store should provide all the tuples starting from the history start time of this event until time $t$. Thus, as the data arrives to DStream it should be archived automatically in order to run hybrid queries on live and archived streams as well as ad-hoc queries on archived streams. However, data insertion to DArchive is costly because of write operations on disk and may slow down the query processing and should be managed in an efficient manner.

A trivial solution for automatic data insertion to DArchive would be using triggers or manually inserting data to DArchive via ODBC(Open Database Connectivity).

- **Triggers**: Triggers are high level functions that can be written on INSERT and DELETE queries over a table. For our purpose we could either write a trigger to insert data to DArchive when the data is
inserted to DStream or when the data is deleted from DStream. However, data insertion is performed by input adapters and data deletion is performed during continuous or hybrid query execution without INSERT/DELETE queries. Triggers operate at query level rather than data level, i.e. triggers can not recognize data insertion/deletion unless explicit INSERT/DELETE queries are written. Since there is only SELECT query on DStream during continuous/hybrid query processing, it can not be used for triggers. Besides, triggers are performed as the triggering query is finished. Since we have continuous queries, they can not trigger another query during their operation. Thus, triggers do not meet the requirements of automatic data insertion to DArchive from DStream. Even if their limitations were eliminated with some modifications, they would perform poorly since they should be called for each live stream.

• **ODBC**: Another option is to insert data to DStream and DArchive at the same time by connecting to MySQL via ODBC from the input adapter where the data insertion to DStream is performed. Then, the same data that is inserted to DStream could be inserted to DArchive. However, Live Stream Store is an in-memory store whereas Archive Stream Store resides on disc. Archived Stream Store is not as fast as Live Stream Store and falls behind it during tuple insertion. As a result of this situation, either query processing should be blocked until the required tuples are inserted to DArchive or an in-complete set of results should be returned which are both undesirable. Besides, duplication of input streams and reinsertion of duplicates may slow down the input rate since the query via ODBC goes through query parsing and optimization steps of MySQL.

As the discussed options do not meet live event archiving requirements of DejaVu, we introduce a new option which is inserting data to DArchive within MySQL engine. The idea is to insert raw data which is extracted from DStream and residing in MySQL engine to DArchive. With the usage of low level functions that Pluggable Storage Engine API provides we can perform efficient insertions to DArchive without query parsing and optimization steps. In order to apply this idea, we need an in-between structure that buffers tuples to be inserted to DArchive as they may be deleted by query processor after they are processed but not yet archived. This brings us to the introduction of a new hierarchical storage (Figure 4.2) which consists of Live and Archived Stream Stores and the recent buffer.
The recent buffer stores the tuples extracted from Live Stream Store and inserts them to Archive Stream Store in bulks. With new hierarchical storage, insertions to the Archive Stream Store is performed automatically and in bulks and a full history for the hybrid queries is provided.

4.1.2 Components

This section explains the components of hierarchical storage briefly. For the detailed implementation of each component please see section 4.1.3.

Live Stream Store (DStream)

This is the entrance point of live events to DejaVu. For this level of hierarchical storage, current version of DStream is used without any modifications. A DStream table is created using the syntax in Figure 4.3.

```
CREATE STREAM stream_name(column_name data_type[(length)], ...);
```

Figure 4.3: Creating a DStream Table

Recent Buffer

The recent buffer is an in-memory structure that resides in the middle of DStream and DArchive. It stores the most recent tuples that has been extracted from DStream. It is responsible for inserting tuples to DArchive in
bulks and providing input tuples to QP for hybrid queries as well as organizing the coordination between two stream stores. Its size may be fixed to a specific number of tuples or memory during system configuration. Otherwise, its size is adjusted automatically with respect to query requirements.

**Archived Stream Store (DArchive)**

This is an append-only on-disk storage engine to store archived streams. Live streams are transferred to DArchive via the recent buffer. No DELETE, UPDATE, and REPLACE operations are allowed on DArchive tables. It ensures the time order among tuples with an index on time-stamp and may have additional indexes on different attributes. A DArchive table is created and bound to the live stream using the syntax in Figure 4.4. After the creation of DArchive table, the recent buffer between DStream table and DArchive table is created automatically.

```
CREATE TABLE archive_name(column_name data_type[(length)], ...) LIVE = stream_name;
```

Figure 4.4: Creating a DArchive Table

DStream and the recent buffer are in-memory components of hierarchical storage. In the current version of hierarchical storage, no persistence guarantee is given for the streams residing in DStream and the recent buffer. In the case of a failure at memory, the streams that are not forwarded to DArchive might be lost. However, this issue is not the main focus of this thesis and left as a future work.

### 4.1.3 Implementation

*Live Stream Store* is implemented as a shared memory area which is accessible by input adapter and query processing engine. All meta data regarding the name, schema, size etc. of the table and the input streams are stored in this shared memory area. Input adapter is responsible to insert input streams to DStream. Input streams are stored in an in-memory FIFO queue and accessible by query processing engine with read functions of pluggable storage engine API. Depending on the input handling mode of DejaVu, input streams are pushed to or pulled by query processing engine during query execution. With the usage of FIFO queue structure, DStream ensures the time order among input streams. On the other hand, input streams are protected with read and write semaphores since they are accessed by both input adapter and query processing engine.
**Recent buffer** is designed to store recent live events that are going to be inserted to DArchive. It is implemented as an in-memory FIFO queue within MySQL engine. Live events residing in input holders of FSM that is processing the pattern on live streams are forwarded to the recent buffer before they are totally deleted from the system. The recent buffer is not only a buffering mechanism for bulk insertion to the DArchive but also a source buffer for the input holders of FSM that is processing the pattern on archived streams. Thus, the decision on when and which part of the data will be inserted to the recent buffer should be taken carefully. If the pattern matching queries over live and archived streams have the same field on PARTITION BY clause, it is more reasonable to keep the partitioned structure within the recent buffer. Otherwise the recent buffer keeps the recent streams in their original order. Please note that the different structures of the recent buffer does not contradict with the time order guarantee of DArchive since DArchive has an index on time.

In the scope of this thesis, we focus on the hybrid pattern matching queries that have the same field on PARTITION BY clause and leave the queries with different PARTITION BY clauses as a future work. In this case, live streams are inserted to the recent buffer after they are forwarded to input holders by the router. The recent buffer is implemented as a combination of sub-buffers each of which stores the recent streams of a specific partition. The recent buffer uses a hash index to reach each partition efficiently. Recent live streams are not inserted to sub-buffers immediately when they arrive to input holders but inserted when a newly detected event on live stream needs them or they are going to be extracted from input holders. Immediate insertion would make the recent buffer store the most recent events. However, if a run instance is processing older live events and finds a match on these events than some of the events in the recent buffer will be ahead of live pattern and will not be used. Choosing the other option, we ensure that only the necessary recent events within the relevant time distance will be stored in the recent buffer and it will not be overloaded by the newest live events.

Being an in-memory structure, the recent buffer brings crucial advantages to the DejaVu CEP system for efficient hybrid query processing. However, it does not provide a full fault tolerance as discussed in Section 4.1.2 and may postpone archiving of live events.

**DArchive** was using Archive Storage Engine of MySQL in the previous version of DejaVu. With the introduction of hierarchical storage, MyISAM Storage Engine is modified slightly to implement DArchive. DArchive implementation is based on MyISAM since it supports the features that DArchive requires like B-Tree indexing, uncompressed storage of raw tuples, etc. that
Archive Storage Engine does not provide. MyISAM is modified to ensure append-only data by disabling DELETE, UPDATE, and REPLACE operations. With the append-only data guarantee, row-based locking is ensured for DArchive. Locking mechanism of MyISAM is also modified to enable automatic insertions from the recent buffer.

4.1.4 Size of The Recent Buffer

As discussed in Section 4.1.2, the size of the recent buffer might be limited to a certain number of events or a specific value in Bytes, Kilobytes, etc. during system configuration. By default, the recent buffer is designed to store all live events that can be seen within the given time distance. Then, the maximum memory consumption of the recent buffer is

\[ \max(|\text{recent}|) = k \times \text{size}_{\text{event}} \]  (4.1)

where \( k \) is the maximum number of events that can be stored in the recent buffer and \( \text{size}_{\text{event}} \) is the size of each event.

If the recent buffer is limited to a certain size, it stores possible number of events in memory and sends extra ones to DArchive. In this case, it cannot store all the recent events that are included in the history of a live event and disk access might be required to provide complete history. Hence, the size of the recent buffer should be configured properly as it may increase the possibility of disk accesses especially when it contains a small amount of data compared to relevant history of a live event.

4.2 Historical Result Cache

4.2.1 Motivation

One important bottleneck of hybrid queries in DejaVu is the re-computation of pattern matching queries over archived streams. During live event detection, each event entering DejaVu system is accessed once and is not processed again after all run instances using that even are finished. However, this situation is not the same for archived stream processing. In general, the history of an event intersects with the history of another event which results in accessing and processing same portion of archived streams more than once. In traditional databases, materialized views are used in order to avoid the overhead of accessing and processing same portion of data. The query result is stored as a regular table which should be updated as the base data changes and this table is used to respond future queries. The usage of materialized views is advantageous when database access to the materialized view is faster than recomputing the view[10]. Similarly, caching previously found matches over archived streams may decrease the access and process
costs in hybrid queries especially when the histories of live events highly overlap.

For this reason, we introduce the historical result cache for efficient query processing. As shown in Figure 4.5, archive matches that are computed for Live Match 1 are stored in the historical result cache to be used by future live matches like Live Match 2.

![Figure 4.5: Historical Result Cache](image)

As the requirements of hybrid queries differ from regular queries, we do not use traditional materialized views but offer an in-memory structure for result caching. As a result of time distance feature of hybrid queries, the historical result cache should include matches within a time window which always slides forward. This situation limits the size of the historical result cache to the maximum number of matches within this time window and makes it a dynamic structure rather than a lookup table. This dynamism should not be confused with view maintenance problem in traditional databases which is defined in related work[10] as the process of updating a materialized view in response to changes to the underlying data. In our case, we ensure that the underlying data is append-only and no updates are possible. Thus, all the data in the historical result cache is consistent with raw data and does not need to be updated. The issue is that former match results in the historical result cache become irrelevant as the time passes and should be deleted since they will not take place in the history of future live events. As the old matches are deleted from the historical result cache, fresh ones will be inserted. As a consequence of dynamic structure and size limitation of the historical result cache, it is designed as an in-memory structure rather than a regular database table.
4.2.2 Data Access

The historical result cache is accessed when a new event is detected over live stream. Let the newly detected event have start and end timestamps \( t_s, t_e \) respectively and the time distance be \( t_d \). Then, the query processor computes the relevant history time interval \([t_1, t_2]\) where

\[
t_1 = \begin{cases} 
  t_s - p_d & \text{if } p_d < t_s \\
  0 & \text{otherwise}
\end{cases}
\quad \text{and} \quad t_2 = t_e - 1
\]

The historical result cache is notified via history notification mechanism which will be discussed in Section 4.3 of this time interval. The previously found matches with start timestamp smaller than \( t_1 \) goes stale since the live events are ordered by their start timestamps and no live event started before \( t_s \) will occur in the future. The matches that fall behind the current history time interval will not be required again and they are deleted from the cache permanently. On the other hand, some of the previously found matches may have end timestamp larger than \( t_2 \) since the events are ordered by their start timestamps (see Section 2.2). These matches do not take part in the current history either but they should not be deleted as they may be in the history of future live events. Thus, only the previously found matches that are in the current history time interval are joined with the newly detected event and reported to the client if they satisfy the join conditions. Having reported the previously found matches, the query processor starts to process the data in the recent buffer to detect events over archived streams that might occur after the previous live event. Newly detected matches over archived streams are also joined with the live event and inserted to the historical result cache in the order that they occur.

4.2.3 Size Estimation

The historical result cache stores matches which are in a certain time interval as a result of time distance feature of hybrid queries. The start and end points of this time interval changes for every live event but its size is limited to the time distance. If the frequency of archived matches \( f_a \) is given, the maximum number of matches that can occur within time distance \( t_d \) could be estimated as \( t_d \times f_a \) assuming that the matches are uniformly distributed. Then, the maximum memory consumption of the historical result cache is

\[
\max(|cache|) = t_d \times f_a \times size_{match} \tag{4.2}
\]

where \( size_{match} \) is the size of each match over archived streams.
4.2.4 Implementation

Requirements of accessing to previously found matches over archived streams determine the structure of the historical result cache.

No update operation is performed over archived matches as the base data and consequently the matches over this data will never change. On the other hand, matches are written to the cache in the order they occur and needed to be read and deleted in the same order. Having these data access requirements, we implement the historical result cache as a FIFO queue. Each node of the queue represents a match over archived streams which consists of a set of field and/or aggregate attributes and some meta data about the match like the start-end timestamps and total number of rows contributed to the match. Each newly detected event over archived streams is inserted to the queue with enqueue function and old matches are deleted from the queue with dequeue function. With the FIFO structure, the historical result cache disables random data insertion and deletion operations and maintains the time order among matches.

4.2.5 Indexing

According to hybrid query specifications, we could build some indexes on the historical result cache in order to have a fast access to the previously found matches. Access to the previously found matches may require some filtering if there is a join condition over live and archived events. At least, a filtering on start timestamp of the match is required because of the time distance feature of the hybrid queries. In other words, a read operation on the historical result cache requests matches which has started at a given history start time $t$ or later. After this read request, the matches with the start time smaller than $t$ should be deleted from the cache since no live event will require matches started before $t$. In this case, using a B-Tree index on start time of the match seems to speed up the process but actually it does not help since the read operation requires almost a full scan of the historical result cache for deletion of the matches.

Apart from the time index, the historical results might be stored in partitions or some other indexes on different attributes may be implemented. For example, if there is a join condition over live and archived matches on attribute $a_1$, an index on $a_1$ might be beneficial especially if $a_1$ is in PARTITION BY clause. In our use case scenario, we have PARTITION BY clause and equality condition on $Symbol$ attribute. Hence, we partition the historical results by this attribute and build a hash index to access each partition.

Please note that partitioning the historical results would bring the highest
benefit when there is an equality condition on the same attribute. Otherwise, an extra structure is required to maintain the time order among different partitions. In cases where partitioning is not efficient, trade off between benefits and costs of different indexes should be examined in details since the maintain cost of the index might be high due to the frequent insert and delete operations.

4.2.6 Advantages and Limitations

The historical result cache provides benefits by reducing the amount of archived streams to be accessed and processed. The advantage of using historical result cache increases as

- the amount of archived streams to be accessed,
- the cost of archived stream access, and
- the processing cost of pattern matching over archived streams

increase.

The amount of archived streams to be accessed depends on the average access frequency defined in Section 3.4 Definition 5. The cost of archived stream access depends on where the data resides. If the recent buffer is not able to store necessary archived streams, disc access is required to obtain the complete history and the cost will increase. Finally, the cost of pattern matching over archived streams depends on the complexity of the pattern. Pattern specifications and the predicates over the pattern affect the complexity. For example, the cost of processing a fixed size pattern like $AB$ will not be the same as a varying size pattern $AB^+$ or the cost of a pattern that has a dynamic predicate over $A$ and $B$ will not be the same as a pattern with a static predicate over $A$.

Apart from these advantages, the historical result cache has some limitations since it is an in-memory structure. First of all, its size is limited to the available memory. On the other hand, stored matches might be lost in case of a memory failure. However, these matches could be recomputed using raw archived streams in DArchive.

4.3 History Notification

With the introduction of time distance in hybrid queries, only a limited portion of archived streams are required for each newly detected live event. The aim of defining a time distance is to deduce relevant portion of archived streams to be accessed and processed without accessing and processing all
archived streams. Since the start and end time of an event can not be known until the event is detected, time distance between live and archive matches should not be regarded as a regular join condition but should be pushed down to hybrid pattern matching process. Therefore, an internal mechanism is required to access and process archived streams according to time distance value. For this reason, a notification system which notifies the recent buffer and the historical result cache of relevant history start and end times is introduced. This notification system is build in query processor. After a live match is found, query processor notifies the recent buffer and the historical result cache of the start and end time of this event and they update themselves accordingly.

4.4 The Detailed Flow of Data

Figure 4.6 includes the detailed version of extended DejaVu architecture in Figure 4.1 and demonstrates the detailed flow of live events, archived events, history notifications and query results during hybrid query processing as discussed in previous sections.
4.5 Language Extensions

In order to fully express hybrid queries following keywords are added to the current DejaVu query language.

1. **LIVE**: This keyword is used when a DArchive table to store archived streams is created. It specifies the DStream table which stores the live streams.

2. **TDISTANCE**: This keyword is used to specify the time distance in unit time to be used to determine history of a live event. It is set to infinity by default where history of a live event will include all the events from the beginning of history.
Chapter 5

Archived Stream Processing Strategies

Hybrid queries consist of at least two pattern matching queries over live and archived streams as discussed in Section 3.3. The pattern matching over archived streams depends on the detected events over live streams due to the time distance and other possible join conditions between two matches. In this case, each event is processed for live pattern matching query but not necessarily processed by the archive pattern matching query. By default, DejaVu processes archived streams in lazy manner, i.e., when a match is detected over live stream. On the contrary, it may also process each live event eagerly for the pattern matching query over archived streams. As we will show in this chapter, each strategy can provide benefits under different conditions.
5.1 Lazy

In lazy mode, each live event is processed to detect matches over live events and buffered in the recent buffer or archived on the disc. The historical result cache stores the previously found matches but the most recent data since the last live event detection is unprocessed by the pattern over archived matches. When a match is detected over live events, necessary portion of archived streams is accessed and processed by the query processor. Then, the previously found matches and the newly detected matches over archived streams are joined with the live match and delivered to the user. Figure 5.1 shows the activity diagram of DejaVu when it is in lazy mode.

![Activity Diagram in Lazy Mode](image)

If the average access frequency of the archived streams is less than 1 or there exists a dynamic condition between live and archived streams, some portion of the archived streams may not be required for the hybrid query. In lazy mode, only the necessary portion of archived streams is processed and query processor is not overloaded with the unnecessary computations.
5.2 Eager

In eager mode, each live event is considered to be the history of future live matches and processed to detect matches over both live and archived streams. The possible history of a live event is processed on the fly and cached in the historical result cache. When a match is detected over live events, prepared match results residing in the historical result cache are joined with the live match and delivered to the user immediately. Figure 5.2 shows the activity diagram of DejaVu when it is in eager mode.

Figure 5.2: Activity Diagram in Eager Mode

If the average access frequency of the archived streams is greater than 1 and the join conditions are static, evaluating pattern matching queries over archived streams eagerly, i.e., as the data arrives to the system, does not cause any unnecessary computations and may decrease the response time of live matches under specific conditions, which will be discussed in the following sections.
5.3 Lazy vs. Eager

5.3.1 Memory Consumption Analysis

Lazy and eager modes differ from each other in terms of memory consumption. In lazy mode, the size of the recent buffer may grow to its maximum size until a new match is detected over live events. In eager mode, the recent buffer size is constant since it stores only the last inserted tuple and no disc access is required for hybrid queries. On the other hand, the maximum memory consumption of the historical result cache is the same in lazy and eager modes as the frequency and size of the archived matches and the time distance are the same. The only difference in terms of historical result cache usage is the update frequency of the historical result cache. In lazy mode, it is updated when a new match is detected over live events, whereas it is updated for each live event in eager mode.

5.3.2 Response Time and Throughput Analysis

The performance of DejaVu in different modes could be analyzed with respect to the average response time of live matches and the throughput of the system. The performance of DejaVu in lazy and eager mode with the same data set and hybrid query depends on the relation between input rate and the processing cost of the live and archive pattern matching. Assume that the live events have a data rate of N tuples per second and a live event is inserted to the system at every \( t_{input} = \frac{1}{N} \) seconds. Let \( t_{live} \) be the average time to process one event for live pattern matching and \( t_{archive} \) be the time to process one event for archive pattern matching. Then, the cost of processing one live event in case of a match or non-match in lazy and eager modes are

\[
\begin{align*}
  t_{l_{nonmatch}} &= t_{live} \\
  t_{e_{nonmatch}} &= t_{live} + t_{archive} \\
  t_{l_{match}} &= t_{live} + n \times t_{archive} + t_{join} \\
  t_{e_{match}} &= t_{live} + t_{archive} + t_{join}
\end{align*}
\]

where \( n \) is the number of live events which are buffered in the recent buffer to be processed by archive pattern match in lazy mode and \( t_{join} \) is the cost to join detected events over live and archived streams.

Figure 5.3 shows tuple processing time analysis in lazy and eager mode. In both modes, processing time of a tuple remains constant until a live match is found. When a live match is found, processing time of a tuple increases dramatically in lazy mode compared to eager mode due to the processing cost of buffered recent events.
The performance of DejaVu CEP system in lazy or eager manner varies as $t_{input}$ decreases. Tuple processing time analysis of eager and lazy modes shown in Figure 5.3 can be divided into two region with a threshold $t_{threshold}$ on $t_{input}$. The performance of DejaVu differs with respect to the chosen archived stream processing strategy for different values of $t_{input}$.

**Region 1: $t_{input} \geq t_{threshold}$**

This region represents the cases where input rate is too slow and the query processor has enough time to process live events both in lazy and eager manner. In this case, each live event is processed immediately after it is available to the system and the processing in both lazy and eager mode finishes before the next event is available. In this region, throughput of the system is dominated by the input rate but a difference in response time of live events is expected between lazy and eager modes.

**Region 2: $t_{input} < t_{threshold}$**

In this region, input rate gets faster and query processing in different modes can not cope with the data rate. Each live event is processed with a delay since the processing of the previous event finishes after arrival of the new event. In this region, performance difference between lazy and eager modes
depends on the query and input specifications. However, the performance of lazy mode in terms of throughput and response time is expected to be better compared to eager mode due to the possible unnecessary computations, more update costs on the historical result cache and more communication cost between live and archive pattern search processes in eager mode.
Chapter 6

Experiments

6.1 Experimental Setup

All experiments are measured on a notebook with Intel Core 2 Duo 2.53 GHz CPU and 4 GB memory on Ubuntu 8.04 operating system.

6.2 Recent Buffer

In this set of experiments, we aim to demonstrate the benefits of the recent buffer.

The recent buffer is designed to manage archiving of live events and provide complete history for the hybrid queries in an efficient manner. The advantage of using recent buffer for providing history is straightforward since it decreases the number of disc accesses by storing most recent streams in the memory. Assume that the query processor needs $N$ tuples from history in order to respond a hybrid query. If the size of the recent buffer is not limited by user and there is sufficient space in the memory, the recent buffer could store all $N$ tuples in memory and provide complete history without any disc accesses. On the other hand, if the recent buffer size is limited to $M$ tuples where $M < N$, it still decreases the number of disc accesses by $M$.

Besides, the recent buffer stores the tuples in the format that query processor can directly use. However, tuples on disc are stored in raw format and needed to be parsed before processing. Hence, the recent buffer provides an obvious advantage during archived stream access. Apart from that it also provides efficient archival of live events. In order to see the advantage of recent buffer, the time elapsed to archive randomly generated 10000 live events with and without the recent buffer is captured. First, the recent buffer is disabled and live events are archived as they arrive to input adapter via ODBC. Live events are duplicated before they are inserted to DStream.
After each tuple is parsed and placed in the INSERT query, input adapter connects to DArchive table via ODBC. For a better performance, live events are inserted in bulks of 100, 1000, and 10000 tuples. In order to have an even comparison between insertion times, the time taken for preparing the query and connecting to the server are not counted. Then the recent buffer is enabled to insert live events within query processor. The time elapsed to insert 10000 tuples in bulks of 100, 1000 and 10000 is again measured. Figure 6.1 shows the performance evaluation of archiving live events by connecting via ODBC and using the recent buffer with different bulk insertion sizes.

![Performance Evaluation of Archiving Live Events](image)

Figure 6.1: Performance Evaluation of Archiving Live Events

As seen in the Figure 6.1, ODBC connection performs at least 1.5 times worse than using recent buffer as it goes through query parsing and optimization steps of MySQL. After a threshold (1000 tuples under this setup), the performance of ODBC decreases in bulk inserts since the query parsing phase consumes more time for longer queries. On the other hand, the recent buffer has a stable performance for different number of bulk inserts since it only pays the fixed cost of data insertion for each tuple.
6.3 Historical Result Cache

In this set of experiments, the efficiency of using the historical result cache is demonstrated by manipulating the amount of archived streams to be accessed, the cost of archived stream access and the processing cost of pattern matching over archived streams which are the factors affecting the advantage of the historical result cache as discussed in Section 4.2.6.

The performance of DejaVu CEP System is measured under three scenarios by using two different queries and 8 different average access frequencies over a synthetic data set of 5000 tuples. In the first two scenarios, the historical result cache is enabled and disabled respectively to observe the overhead of reprocessing of archived streams when the historical result cache is not used. In order to measure the effect of reprocessing more precisely, the recent buffer size is not limited in these scenarios assuming that the required memory for maximum recent buffer size is always available. However, in the real life applications, the recent buffer size may grow substantially due to the larger time distance values and the memory may not be available to store all the recent tuples. In order to simulate real-life use case, the recent buffer size is limited in the third scenario so that it could store only the half of the required tuples for each average access frequency. The motivation behind the third scenario is to observe the disc access cost of hybrid queries when the historical result cache is not used.

The performance of the system is measured in terms of throughput which is defined in Section 3.4 Definition 6. The pre-recorded input file is dumped to the system before the queries are run to eliminate the effect of input rate on the throughput. Throughput values in the experiments are the average of 5 runs.

6.3.1 Input Specifications

A synthetic data set with the schema given in 3.1 is generated to keep the value of average access frequency under control. It includes 5000 events within a time interval of 5000 seconds (one tuple per second) and uniformly distributed matches over live events with a frequency of 1/2. By using this data set and changing the time distance, the value of average access frequency is manipulated. It is also possible to manipulate the average access frequency with a fixed time distance and varying live match frequency but the frequency of live match could be changed by using different data sets which would affect the unit cost of processing pattern matching over live and archived events. Thus, the first method is used in our experiments to fix the unit cost of processing the pattern matching over live and archived streams for each run.
Synthetic data set is also adjusted to have matches for two different patterns over archived streams used in the test queries.

6.3.2 Query Specifications

Test queries \textit{Query}_1 and \textit{Query}_2 are shown in Figure 6.2 and 6.3 respectively. They have the same pattern specifications over live events and the same output schema but different archive pattern complexities.

```sql
SELECT symbol_l, start_l, end_l, start_a, end_a
FROM LiveStock 
MATCH_RECOGNIZE(
    PARTITION BY symbol
    MEASURES A.symbol AS symbol_l, A.time AS start_l, B.time AS end_l
    ONE ROW PER MATCH
    AFTER MATCH SKIP to next ROW
    ALL MATCHES
    PATTERN (A B)
    DEFINE /* A matches any row */
           B AS (B.price < A.price)),
ArchiveStock 
MATCH_RECOGNIZE(
    PARTITION BY symbol
    MEASURES A.symbol AS symbol_a, A.time AS start_a, B.time AS end_a
    ONE ROW PER MATCH
    AFTER MATCH SKIP to next ROW
    ALL MATCHES
    PATTERN (A B C D)
    DEFINE /* A matches any row */
           B AS (B.price < A.price)
           C AS (C.price = A.price)
           D AS (D.price > A.price)
WHERE symbol_l = symbol_a
TDISTANCE = <t>;
```

Figure 6.2: Query 1 (Simple Archive Pattern)
Figure 6.3: Query 2 (Complex Archive Pattern)

```
SELECT symbol_l, start_l, end_l, start_s, end_s
FROM LiveStock MATCH RECOGNIZE(
  PARTITION BY symbol
  MEASURES A.symbol AS symbol_l, A.time AS start_l, B.time AS end_l
  ONE ROW PER MATCH
  AFTER MATCH SKIP to next ROW
  ALL MATCHES
  PATTERN (A B)
  DEFINE /* A matches any row */
  B AS (B.price < A.price),
ArchiveStock MATCH RECOGNIZE(
  PARTITION BY symbol
  MEASURES A.symbol AS symbol_a, A.time AS start_a, D.time AS end_a
  ONE ROW PER MATCH
  AFTER MATCH SKIP to next ROW
  ALL MATCHES
  PATTERN (A B+ C+ D+)
  DEFINE /* A matches any row */
  B AS (B.price <= PREV(B.price) AND B.price < A.price)
  C AS (C.price => PREV(C.price) AND C.price <= A.price)
  D AS (D.price > PREV(D.price) AND D.price > A.price)
WHERE symbol_l = symbol_a
TDISTANCE = <t>;
```

Query2 is a more complex pattern compared to Query1 since it is a varying size pattern and has more complex correlation on pattern variables. The processing cost of Query2 is expected to be larger than the processing cost of Query1 on the current data set and with the same time distance.
6.3.3 Results

The performance evaluation of DejaVu is given in the following sections. The throughput values are specific to this experimental setup and should not be considered as the general throughput of the system. The goal is to demonstrate the trend in the throughput which can be generalized to other experimental setups with different data sets and query specifications.

Table 6.4 shows the time distance values for each different average access frequency as well as the total number of required archived stream access for the following experiments. For the same average frequency, total archived stream access is the same as total required archived stream access without the historical result cache. With the usage of historical result cache, total number of archived stream access decreases to the total number of tuples since accessing each tuple is accessed once.

<table>
<thead>
<tr>
<th>Time Distance ($t_i$)</th>
<th>Total Number of Tuples</th>
<th>Frequency of Live Matches ($f_l$)</th>
<th>Average Access Frequency</th>
<th>Total Number of Required Historical Data Access</th>
<th>Total Number of Historical Data Access without Query Result Cache</th>
<th>Total Number of Historical Data Access with Query Result Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5000</td>
<td>0.5</td>
<td>2.499</td>
<td>12495</td>
<td>12495</td>
<td>5000</td>
</tr>
<tr>
<td>8</td>
<td>5000</td>
<td>0.5</td>
<td>4.4964</td>
<td>22482</td>
<td>22482</td>
<td>5000</td>
</tr>
<tr>
<td>16</td>
<td>5000</td>
<td>0.5</td>
<td>8.4864</td>
<td>42432</td>
<td>42432</td>
<td>5000</td>
</tr>
<tr>
<td>32</td>
<td>5000</td>
<td>0.5</td>
<td>16.4472</td>
<td>82236</td>
<td>82236</td>
<td>5000</td>
</tr>
<tr>
<td>64</td>
<td>5000</td>
<td>0.5</td>
<td>32.292</td>
<td>161460</td>
<td>161460</td>
<td>5000</td>
</tr>
<tr>
<td>128</td>
<td>5000</td>
<td>0.5</td>
<td>63.6744</td>
<td>318372</td>
<td>318372</td>
<td>5000</td>
</tr>
<tr>
<td>256</td>
<td>5000</td>
<td>0.5</td>
<td>125.2104</td>
<td>626052</td>
<td>626052</td>
<td>5000</td>
</tr>
<tr>
<td>512</td>
<td>5000</td>
<td>0.5</td>
<td>243.3672</td>
<td>1216836</td>
<td>1216836</td>
<td>5000</td>
</tr>
</tbody>
</table>

Figure 6.4: Experiment Details

Query$_1$ Results

Figure 6.5 shows the performance of DejaVu CEP System for query Query$_1$ under three scenarios. Comparing the first two scenarios, throughput of the system decreases dramatically without the historical result cache as the average access frequency increases. Using the historical result cache results in slight decrease in the throughput as the average access frequency increases.
The reason of the decrease in throughput in both cases is the increase in the amount of data to be processed and reported. Since the historical result cache reduces the amount of data to be processed substantially, throughput decreases slightly due to the amount of data to be reported. However, if the historical result cache is not used, throughput decreases in a great extent due to the multiple processing of the same archived streams. On the other hand, throughput is worse than the first two scenarios in the third scenario, due to the disc access. Different than the first two scenarios, throughput increases slightly with the average access frequency until some point and then starts to decrease. The reason for that is the different disc access costs for different frequencies. As the average access frequency increases, the number of disc accesses increase but the average cost of each access decrease.

Figure 6.5: Simple Query Performance Evaluation
Query \textsubscript{2} Results

Figure 6.6 shows the performance of DejaVu CEP System for query \textit{Query}\textsubscript{2} under three scenarios. We observe the similar decrease in the throughput as in the performance evaluation of \textit{Query}\textsubscript{1} due to the same reasons. Throughput of each average access frequency is lower than the throughput of the same access frequency in the first query since \textit{Query}\textsubscript{2} is more complex than \textit{Query}\textsubscript{1}.

![Complex Query Performance Evaluation](image)
**Query**$_1$ vs. **Query**$_2$

We observe the similar trends in throughputs of **Query**$_1$ and **Query**$_2$. Performance of the system running complex pattern (**Query**$_2$) is worse than the performance of the system running simple pattern (**Query**$_1$) but the gain provided by the historical result cache increases as the complexity of the query increases. In order to observe the increasing advantage of the historical result cache with more complex queries, first two scenarios are taken. Figure 6.7 shows the comparison of throughput ratio of the first two scenarios for simple (**Query**$_1$) and complex (**Query**$_2$) pattern queries.

![Figure 6.7: Simple and Complex Query Throughput Ratio Comparison](image-url)
Memory Consumption

Table 6.8 shows the maximum memory consumption of the recent buffer and the historical result cache under three scenarios for different time distance values. These values are calculated based on the formulas 4.1 and 4.2 in Chapter 4 where the size of a live event is 60 Bytes and a match on archived data is 100 Bytes.

<table>
<thead>
<tr>
<th>Time Distance t_d</th>
<th>Average Access Frequency</th>
<th>Recent Buffer Size</th>
<th>Maximum Memory Consumption in Bytes (Simple and Complex Query)</th>
<th>Maximum Memory Consumption in Bytes (Query Result Cache)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.499</td>
<td>unlimited</td>
<td>120</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>6.4964</td>
<td>unlimited</td>
<td>120</td>
<td>200</td>
</tr>
<tr>
<td>16</td>
<td>8.4964</td>
<td>unlimited</td>
<td>400</td>
<td>800</td>
</tr>
<tr>
<td>32</td>
<td>16.4962</td>
<td>unlimited</td>
<td>120</td>
<td>160</td>
</tr>
<tr>
<td>64</td>
<td>32.292</td>
<td>unlimited</td>
<td>120</td>
<td>1400</td>
</tr>
<tr>
<td>128</td>
<td>63.6744</td>
<td>unlimited</td>
<td>120</td>
<td>1300</td>
</tr>
<tr>
<td>256</td>
<td>125.2104</td>
<td>unlimited</td>
<td>120</td>
<td>1200</td>
</tr>
<tr>
<td>512</td>
<td>243.3872</td>
<td>unlimited</td>
<td>120</td>
<td>12500</td>
</tr>
</tbody>
</table>

Figure 6.8: Memory Consumption

In this experiment set, the frequency of a live match $f_l$ is 1/2 and the frequency of an archive match $f_a$ is 1/2 for simple query (Query 1) and 1/4 for complex query (Query 2). Then, the maximum size of the recent buffer is $t_d f_l$ tuples in Scenario 1, $t_d$ tuples in Scenario 2, and $t_d/2$ tuples in Scenario 3. The maximum size of the historical result cache is $t_d f_a$ matches for each query.

Comparing the total memory consumption of 3 scenarios, we can say that the historical result cache may result in more or less memory usage with respect to the size of the live events, stored matches, recent buffer and time distance.
In this set of experiments, the performance of lazy and eager strategies is measured in terms of throughput (Section 3.4 Definition 6) and average response time of the live matches (Section 3.4 Definition 3) with three different average access frequencies and varying data rate.

Input rate is manipulated from 1000 events/sec to 30000 events/sec and the performance of the system is plotted with the Bézier curves [1] which is a widely-used curve fitting algorithm.

6.4.1 Input Specifications
A synthetic data set of 30000 events within a time interval of 30000 seconds (one tuple per second) and having the schema given in Section 3.1 is generated for this set of experiments. Since the input rate is manipulated during the experiments, the size of input file is increased to 30000 which is the maximum number of events with the given schema that can be inserted to the DejaVu via DStream per second. The input file contains uniformly distributed live matches with a frequency of $1/100$ as well as matches over archived streams used in the test query.

Three different average access frequencies are used during the experiments with the same input file but different time distance values which are 10, 100, and 1000 respectively. The reason to manipulate the average access frequency is to observe the effects of processing archived streams eagerly when some of the pre-computed matches are unnecessary, used once or used more than once.

6.4.2 Query Specifications
The complexity of the pattern matching query over archive streams affects the response time of a live match. In order to observe the difference between lazy and eager cases more precisely, the complex pattern query ($Query_2$) shown in Figure 6.3 is used during the experiments.

6.4.3 Results
The behaviour of the average response time of live matches and the throughput of DejaVu against the increasing data rate in lazy and eager modes for access frequencies 0.1, 1, and 10 are given in the following sections.

Threshold values in Figures 6.9, 6.10, 6.11, 6.12, 6.13, and 6.14 are drawn where the average response time of live matches in eager mode start to increase dramatically since this point corresponds to the threshold value
shown in Figure 5.3.

Please note that the graphs are in logarithmic scale to precisely observe the difference in average response time with low data rates.
Average Access Frequency = 0.1

Figure 6.9: Average Response Time (Freq = 0.1)

Figure 6.10: Throughput (Freq = 0.1)
Average Access Frequency = 1

Figure 6.11: Average Response Time (Freq = 1)

Figure 6.12: Throughput (Freq = 1)
Average Access Frequency = 10

Figure 6.13: Average Response Time (Freq = 10)

Figure 6.14: Throughput (Freq = 10)
6.4.4 Discussion

Results show that the average response time of live matches and the throughput of the system behave differently in two regions discussed in Section 5.3. This experiment set shows a general trend in average response time and throughput values. The values of the thresholds, throughput, and average response time may differ according to the query and input specifications but the trend will be similar for different hybrid pattern matching queries.

Region 1: Left Side of Threshold

Average Response Time

For all average access frequencies, the average response time of live matches in eager mode is better than the average response time of live matches in lazy mode. The difference between the average response time of each mode increases as the average access frequency increases since the workload of lazy processing increases with the increasing average access frequency and processing in eager mode does not cause unnecessary computations.

Throughput

In this region, the data rate is too slow compared to query processing and throughput of the system is dominated by the data rate. Thus, it is approximately the same for lazy and eager modes in all average access frequencies.

Winning Strategy

For each average access frequency, the winning strategy in this region is eager mode since it has a better average response time with the same throughput value.

Region 2: Right Side of Threshold

Average Response Time

For all average access frequencies, the average response time of live matches increases substantially and then reaches the maximum value in both modes since the data rate is too fast compared to processing rate. During this increase, lazy starts to beat eager mode after a certain data rate since the system suffers from frequent switching between live and archive pattern matching and/or unnecessary computations in eager mode.
Throughput

In this region, throughput of the system is better in eager mode while the average response time is better for eager mode. After a certain data rate discussed in previous paragraph, throughput of the lazy mode beats throughput of the eager mode in all average access frequencies.

Winning Strategy

For each average access frequency, the long-term winning strategy in this region is lazy mode since it has a better average response time and throughput after a certain data rate. Until this data rate, eager mode may perform better depending on the query and input specifications. However, the region between the threshold and this data rate can be interpreted as a transition step and the behaviour of the system could be ignored since the system may behave differently in this region for different hybrid pattern matching queries.
Chapter 7

Conclusions and Future Work

In this thesis, the semantics of hybrid pattern matching queries over live and archived streams is defined focusing on recent archived streams. Besides, new components and archived stream processing strategies are proposed to efficiently process these queries with low latency. DejaVu\cite{7} CEP engine is extended to apply new algorithms. Approximately 900 lines of code are added for efficient hybrid query processing and 500 lines of code are added for architectural extensions to 350,000 lines of MySQL code base. With the experiments, a clear improvement in the performance provided by the extended architecture is demonstrated. Moreover, processing archived streams in lazy or eager manner are compared under different scenarios and the winning strategy for each scenario is examined.

As a future work, proposed algorithms could be extended to cover other types of hybrid pattern matching queries. Moreover, workload-sensitive adaptive switch between two archived stream processing strategies could be designed or some other strategies could be introduced by generalizing archived stream processing to a join-order problem. On the other hand, finer-grained analysis and optimization of hybrid pattern matching queries could be performed to detect and exploit pattern similarity and build suitable indices on data.
Bibliography


