Efficiently Correlating Complex Events over Live and Archived Data Streams

Nihal Dindar, Peter M. Fischer, Merve Soner, Nesime Tatbul
Systems Group, ETH Zurich, Switzerland
{dindarn, peter.fischer, msoner, tatbul}@inf.ethz.ch

ABSTRACT

Correlating complex events over live and archived data streams, which we call Pattern Correlation Queries (PCQs), provides many benefits for domains which need real-time forecasting of events or identification of causal dependencies, while handling data at high rates and in massive amounts, like in financial or medical settings. Existing work has focused either on complex event processing over a single type of stream source (i.e., either live or archived), or on simple stream correlation queries (e.g., live events trigerring a database lookup). In this paper, we specifically focus on recency-based PCQs and provide clear, useful, and optimizable semantics for them. PCQs raise a number of challenges in optimizing data management and query processing, which we address in the setting of the DejaVu complex event processing system. More specifically, we propose three complementary optimizations including recent input buffering, query result caching, and join source ordering. Furthermore, we capture the relevant query processing tradeoffs in a cost model. An extensive performance study on synthetic and real-life data sets not only validates this cost model, but also shows that our optimizations are very effective, achieving more than two orders magnitude throughput improvement and much better scalability compared to a conventional approach.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - Query Processing

General Terms

Performance, Languages, Theory

Keywords

Complex Event Processing, Pattern Matching, Data Streams, Stream Correlation, Stream Archiving

1. INTRODUCTION

Complex Event Processing (CEP) has proven to be a key technology for analyzing complex relationships over high volumes of data in a wide range of application domains reaching from financial trading to health care. Complex events are typically expressed by means of patterns that declaratively specify the event sequences to be matched over a given data set.

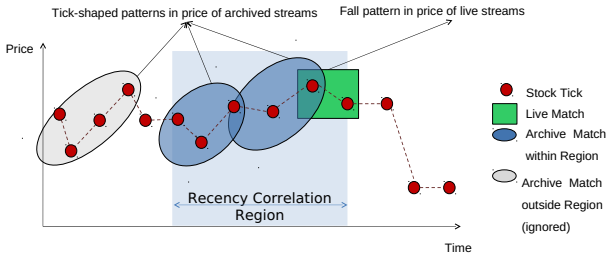


Figure 1: Financial use case: Whenever a price fall is detected on live, find all "tick-shaped" patterns on recent archive.

Early CEP research primarily focused on pattern matching techniques over real-time event streams (e.g., Cayuga [13], SASE+ [10], ZStream [21]). More recently, there has been an increasing interest in archiving streams, not only for regulatory or reliability reasons, but also for supporting longer-term data analysis [16]. The presence of a stream archive enables a new class of CEP applications which can correlate patterns matched over live and archived event streams. Such correlations form the basis for forecasting or predicting future event occurrences (e.g., algorithmic trading in finance, or travel time estimation/route planning in intelligent transportation systems), as well as for identifying causal relationships among complex events across multiple time scales (e.g., medical diagnosis in health care [3, 4]).

We will present a motivating use case from the financial domain, which we will also use as a running example throughout the paper.

1.1 Running Example

In algorithmic trading, market data reports such as price quotes or bids [20] are evaluated using various heuristics to automatically predict the most profitable stocks to trade based on. We can model each individual report as an event and the trading algorithms as pattern matching queries over these events. Patterns are typically defined as regular expressions. Analysis in finance industry includes among others increase/decrease/stability of stock prices. As a concrete example, consider a simple query (used by e.g., a day trader), which does the following: Upon detecting a fall in the current price of stock X on the live report stream, look for a "tick"-shaped pattern for X within a recent archive, where a fall in price was followed by a rise in price that went beyond than the starting price of the fall (see Figure 1). Such an observation might indicate stocks that could bring profits in the near future. The high-rate live

DEBS'11, July 11–15, 2011, New York, New York, USA.
https://doi.org/10.1145/2002259.2002293.

```
SELECT symbolL, initPriceL, minPriceL,
       initPriceA, maxPriceA
FROM StockLive MATCH_RECOGNIZE(
     PARTITION BY symbol
     MEASURES  A.symbol AS symbolL,
               A.price AS initPriceL,
               LAST(B.price)AS minPriceL
     ONE ROW PER MATCH
     AFTER MATCH SKIP TO NEXT ROW
     INCREMENTAL MATCH
     PATTERN (A B+)
     DEFINE /* A matches any row */
            B AS (B.price < PREV (B.price))
),
StockArchive MATCH_RECOGNIZE(
     PARTITION BY symbol
     MEASURES  A.symbol AS symbolA,
               A.price AS initPriceA,
               LAST(D.price) AS maxPriceA
     ONE ROW PER MATCH
     AFTER MATCH SKIP TO NEXT ROW
     MAXIMAL MATCH
     PATTERN (A B+ C* D+)
     DEFINE /* A matches any row */
            B AS (B.price < PREV(B.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_a = symbol_l
RECENCY = 7 seconds;
```

Fall in stock price

Tick-shaped in stock price

Correlation of live and archive patterns

Figure 2: Financial use case: PCQ

stream and the high-volume archived stream, as well as the need for low-latency results for catching momentary trading opportunities to make profits, render this use case a highly challenging one.

This use case requires evaluating complex events over live (fall) and archived streams ("tick"-shaped), and correlating them based on a recency criteria. Price fall for a specific stock (live pattern) can be expressed as an event of that stock (A) followed by another event of the same stock (B), where $B.price < A.price$. Contiguous fall in price can be expressed with a regular expression as AB^+. Fall in price followed by a rise in price ("tick"-shaped archive pattern) can be expressed in a similar way. This query is expressed more concretely in Figure 2, based on the SQL syntax extended with a `MATCH_RECOGNIZE` clause for pattern matching over row sequences [25]. The first `MATCH_RECOGNIZE` clause utilizes the live stream (StockLive), capturing the price fall. The second one works on the archive stream (StockArchive), expressing a "tick" by a decrease and then an increase to a higher price than at the beginning. Finally, the two are correlated with an equality on the stock symbol and a recency criterion of 7 seconds.

1.2 Challenges

Correlating complex events over live and archived streams poses a number of technical challenges:

- First, the semantics for such queries should be cleanly defined. In general, correlations are based on some notion of similarity [11, 24], related to factors such as the structure of the data (e.g., temporal recency, spatial proximity) or application semantics (e.g., contextual similarity). A clean semantics is important not only in terms of usefulness and clarity for the user, but also for the optimizations that it enables in evaluation of such queries.

- Second, the size of the stream archive will quickly grow, while depending on the correlation criteria, not all of it will be relevant to answer a given query. On the other hand, it may not always be possible to pre-fetch this relevant portion of the archive due to the live component in the query. As a result, a dynamic yet low-cost method is needed for selective archive access.

- Third, the system should scale well with potentially high live stream rates. In particular, CEP on high-volume stream archive should be able to keep up with CEP on high-rate live stream arrival.

- Last but not least, the whole problem becomes more difficult to handle due to the complexity of pattern matching queries involving variable processing window sizes and slides on both live and archive sources, some of which will result in non-match, not contributing to the result despite costing processing power. This makes both the cost and selectivity of live-archive pattern correlation queries more difficult to track.

1.3 Contributions and Outline

In this paper, we provide the following contributions to address the above listed challenges:

- a **formal definition** of pattern correlation queries (PCQs) with correlation criteria based on the notion of temporal recency (e.g., happened at most n time units before), providing a composable, useful, and optimizable semantics;

- a **set of algorithms and optimizations** for processing PCQs in storage- and computation-efficient ways in the context of the DejaVu CEP engine [15], focusing on recent input buffering, query result caching, and join source ordering optimizations;

- a **cost model** capturing the relevant trade-offs and cost factors in query processing for PCQs; and

- extensive **experimental evaluation** on synthetic and real-life data sets (NYSE TAQ data set [6]), validating our cost model and showing orders of magnitude benefits in throughput over a conventional approach.

The outline of rest of this paper is as follows: In the next section, we present how we model the semantics of recency-based pattern correlation queries (PCQs) in detail. In Section 3, we discuss the state of the art for processing PCQs in existing CEP systems (algorithm and its complexity analysis), and provide two possible baseline algorithms that suggest obvious improvements over the state of the art. Section 4 provides a detailed description of our techniques for further optimizing PCQs over the baselines. We present the results of our experimental performance evaluation in Section 5. Related work is summarized in Section 6, and we conclude the paper in Section 7 with a brief discussion of potential avenues for future work.

2. MODELING PCQ

In this section, we formally define the semantics of pattern correlation queries. Our formal semantics is based on three essential design goals:

1. *Total order of input streams:* Pattern matching queries over data sequences require establishing a total order among their elements.

2. *Total order of output streams:* For full query composability, it is also desirable to maintain the total order on output streams that result from pattern matching queries, including pattern correlation queries. This is especially challenging in the case of PCQs, as these queries involve two input sequences, one being the archive relative to the other. Thus, the correlations should be allowed only between pairs of events where one has happened before the other (e.g., an event should not be in the archive of itself, or two events should not be in the archive of each other at the same time). In this case, simply following the total order on the two input sources is not sufficient. Furthermore, a total order of the output resulting from PCQs also needs to be defined explicitly.

3. *Usefulness and clarity:* Pattern correlation queries should provide the necessary semantics for expressing a class of queries that are useful in real-life applications. Furthermore, users should be able to be grasp and apply these semantics.

We will gradually establish our semantics to meet these goals, beginning with a number of fundamental concepts (such as event streams, happened-before relation, recency correlation, etc.).

PCQs operate over live and archived streams of tuples, each representing an event. We distinguish between primitive and complex events. Primitive events happen at a specific point in time, whereas complex events are composed of a sequence of other events (primitive or complex) that happen within a time period. To model this, each tuple is assigned a start timestamp (t_s) and an end timestamp (t_e). Primitive events have identical t_s and t_e values, whereas complex events take on the minimum t_s and the maximum t_e of the events that contributed to their occurrence. Furthermore, to achieve total order on input streams (i.e., design goal #1), we assume that tuples have unique (t_s, t_e) values and they appear ordered in a stream (we show this order with $\prec$), primarily by their t_s values, then with their t_e values if a tie needs to be broken (e.g., $(1,4) \prec (2,3)$ and $(1,2) \prec (1,3)$). More formally:

Definition 1 (Time Domain) *The time domain $\mathbb{T}$ is a discrete, linearly ordered, countably infinite set of time instants $t \in \mathbb{T}$. We assume that $\mathbb{T}$ is bounded in the past, but not necessarily in the future.*

Definition 2 (Event) *An event e $\langle t_s, t_e, v\rangle$ consists of a relational tuple v conforming to a schema $\mathbb{S}$, with a start time value $t_s \in \mathbb{T}$, and an end time value $t_e \in \mathbb{T}$, where $t_e \geq t_s$. We use the notation $e.t_s$, $e.t_e$ to denote the start and end time value of an event e, respectively.*

Definition 3 (Primitive Event) *An event e is a primitive event if $e.t_s = e.t_e$.*

Definition 4 (Complex Event) *An event e is a complex event if $e.t_s \neq e.t_e$.*

Definition 5 (Stream) *A stream S is a totally ordered, countably infinite sequence of events such that:*

$\forall e_i, e_j \in S, e_i \prec e_j$, *iff* $e_i.t_s < e_j.t_s$
$\vee (e_i.t_s = e_j.t_s \wedge e_i.t_e < e_j.t_e)$

Establishing a total order on outputs of PCQs (i.e., design goal #2) requires the definition of a happened-before relation between two complex events. In the following, we will first define this relation, and then build the recency correlation definition on top.

Definition 6 (Happened-before Relation ($\rightarrow$)) *Given a pair of events $e_i, e_j \in S$, $e_i \rightarrow e_j$, if $e_i.t_e < e_j.t_e$ and $e_i.t_s < e_j.t_s$, Happened-before is:*

- *transitive:* $\forall e_1, e_2, e_3$, *if* $e_1 \rightarrow e_2$ *and* $e_2 \rightarrow e_3$, *then* $e_1 \rightarrow e_3$.
- *irreflexive:* $\forall e_1, e_1 \not\rightarrow e_1$.
- *antisymmetric:* $\forall e_1, e_2$, *if* $e_1 \rightarrow e_2$, *then* $e_2 \not\rightarrow e_1$.

Definition 7 (Archive of a Stream) *Archive of a stream S at time t (denoted as S_a^t) consists of all events $e \in S$ where $e.t_e < t$. Archive of a stream S as of an event $e \in S$ (denoted as S_a^e) contains all events $e_i \in S$ for which $e_i \rightarrow e$.*

Based on the above, S_a^e cannot contain e itself or any other event $e_j \in S$ which has not happened before e (i.e., $e_j \not\rightarrow e$).

For PCQs, we take temporal recency as the main correlation criteria. In this case, two events are correlated if they are within a specified temporal distance from each other, which we call "recency correlation distance". More formally:

Definition 8 (Recency Correlation) *For a given stream S and a recency correlation distance of P, any event $e \in S$ has recency correlation with the following set of events*

$$\textit{recent(e, P)} = \{\forall e_i \in S \text{ where } e_i \rightarrow e \text{ (i.e., } e_i \in S_a^e\text{) and } e.t_e - e_i.t_s \leq P\}$$

The above definition ensures that for a given live event $e_l \in S$, if an archive event $e_a \in S_a^{e_l}$ has recency correlation with e_l, then all other archive events $e_i \in S_a^{e_l}$ for which $e_a \rightarrow e_i$ must also have recency correlation with e_l. This follows from the transitive property of the happened-before relation.

Recency-based PCQs join live events with archive events based on the recency correlation distance specified in the query, as we define next.

Definition 9 (Recency-based PCQ) *A recency-based PCQ $Q(S_a, p_a, S_l, p_l, P, q)$ takes six parameters: Archived and live data stream sources (S_a and S_l, respectively), patterns to be matched over these sources (p_a and p_l, respectively), a recency correlation distance (P), and a join predicate (q). For a given set of p_l matches over S_l as M_l and p_a matches over S_a as M_a, the recency-based PCQ's result will be as follows:*

$$M_a \bowtie_{P,q} M_l = \{\forall e_l \in M_l, \forall e_a \in M_a (e_l \bowtie_q e_a), \text{ where } e_a \in recent(e_l, P)\}$$

If a live event e_l joins with an archive event e_a to produce an output event e_o, then $e_o.t_s = e_a.t_s$ and $e_o.t_e = e_l.t_e$. Please note that, the total order for PCQ outputs (i.e., design goal #2) is guaranteed when: (i) all e_l's have unique t_e values, and (ii) all e_a's have unique t_s values. (i) is assured by the INCREMENTAL match mode in MATCH-RECOGNIZE for the live pattern specifications. In some sense, this is natural, as INCREMENTAL was specifically designed to be used with live data sources [25]. (ii) is guaranteed using the MAXIMAL match mode in MATCH-RECOGNIZE for the archive pattern specifications.

Definition 10 (Incremental Match) *Given a set with all possible match results for a pattern P as M, the incremental match mode reports only sets of the longest matches ending with each event (i.e., each match has a unique t_e). Results of incremental match mode pattern search M_i can be defined more formally as follows:*

$$\forall m, n \in M_i, m.t_e = n.t_e \textit{ iff } m.t_s = n.t_s$$

Definition 11 (Maximal Match) *Given a set of all possible match results for pattern P as M, the maximal match mode reports only sets of the longest matches starting with each event (i.e., each match has a unique t_s). Results of maximal match mode pattern search M_x can be defined more formally as follows:*

$$\forall m, n \in M_x, m.t_s = n.t_s \textit{ iff } m.t_e = n.t_e$$

As a result of Definitions 9-11, output events that result from joins between live and archive events always get unique pairs of (t_s, t_e) values, maintaining the total order property given in Definition 5.

With the above PCQ semantics, our last design goal is also met, as the user has to only specify the recency correlation distance in time units, without worrying about total order requirements on input and output streams. This is quite intuitive as the three properties of the happened-before relation guarantees the expected semantics.

3. PROCESSING PCQ

In the previous section, we have established the formal semantics of recency-based PCQs. The next step is now to investigate algorithms that can perform this correlation in an efficient manner. In addition, these algorithms should integrate well with typical CEP/data stream processing environments, as to leverage existing standards, operations, and optimizations as much as possible. There are a few considerations on complexity and cost when evaluating these algorithms: Pattern computation is an expensive operation, involving many computations and a significant amount of state [18]. Nearly all of the existing pattern matching algorithms (e.g., [10]) need to "touch" all possibly relevant tuples at least once, thus raising the bar for efficient data management and indexing.

In this section, we will present three basic approaches for implementing PCQs, while investigating further optimizations in the next section:

1. Expressing recency-based correlation with the means of existing CEP or stream processing environments, in particular window-based joins followed by pattern processing.
2. Eagerly performing pattern processing on live and archive streams first, followed by a pattern-aware correlation operator which understands pattern output structure and recency.
3. Lazily performing the second-side (i.e., archive-side) of the pattern processing, driven by the pattern correlation operator and the need for these pattern instances.

As we will see throughout this section, (3) is the most promising.

3.1 State of the Art

Correlating two streams is a well-researched problem for which many approaches have been proposed (e.g., [23]). Most of these differ in their specific approach and semantics, but share a common idea: The possibly infinite stream on both sides is partitioned into a succession of finite subsequences, typically sliding windows. The contents of the windows on each side are joined/correlated to find matching data items. Once the join over this pair of windows is complete, the windows are advanced to handle newly arriving data items.

The policies how to exactly advance the windows are the main differentiator for these joins, but again there is similarity: Most approaches use item counts or timestamps as the underlying concepts. Precisely these concepts have a strong impedance mismatch with pattern correlation: Patterns could start/end with almost any arbitrary item, not just the next item or the item with the next higher timestamp.

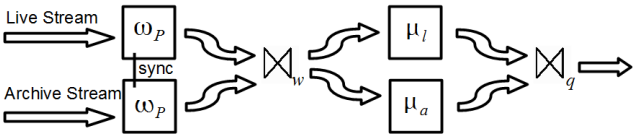


Figure 3: PCQ state of the art: Window correlation first

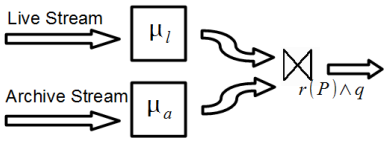


Figure 4: PCQ Pattern First

Despite the similarity of recency regions with sliding windows, only a very rough translation is therefore possible. We show this in Figure 3: Live and archive matches for a given recency region happen in a window with the same size as this recency region (ω_P). By placing such windows on live and archive streams and joining them on size/time ($\bowtie_W$), we can create the recency regions, on which we execute both patterns (μ_l and μ_a). Given the "advancing" mismatch, these windows can only slide by one item/timestamp value, while the pattern matches need to be computed anew for each window. As a result, the effort to compute all correlated matches for N items with a recency region of size P is $(N-1) * P * (c_l + c_a)$, where c_l and c_a are the cost to process a live/archive item on a pattern (see Table 2). While recency could provide significant savings in PCQ computation, using state of the art approaches to express it actually incurs an additional overhead.

3.2 Eager Pattern Processing

There are several ways to overcome the efficiency problems incurred by the semantic mismatch of using state of the art window-based stream correlation methods with PCQ. One approach would incorporate more and more pattern processing knowledge into window correlation in order to overcome the mismatch. Since this would lead to increased code complexity and runtime overhead in a CEP system, we are pursuing a different approach, which is shown in Figure 4: We first execute pattern matching on both live and archive streams on all input elements (μ_l and μ_a). On the pattern matching output, we use a special correlation operator that expresses the recency correlation logic ($\bowtie_{r(P) \wedge q}$, where $r(P)$ denotes a special recency correlation predicate that captures the recency correlation distance P). With this approach, we can ensure a complexity of $N * (c_l + c_a)$ for all pattern computation, since each pattern instance of both streams is computed exactly once, saving the factor P compared to the state of the art approach.

3.3 Lazy Pattern Processing

Eager Pattern Processing has the disadvantage that it needs to compute all patterns present in the data, even though they might never contribute to the result of a correlation. We therefore introduce a third option which has the potential to combine the best aspects of the previous ones: Compute all patterns on one side (e.g., live), but just the necessary patterns on the other side (e.g., archive). The data flow (as shown in Figure 4) does not change compared to the eager version, but there is a significant potential for cost savings. The baseline approach presented in this section can compute recency-based PCQ without additional memory overhead (other than pattern computation itself), but re-introduces some of the additional work of the state of the art approach. In the next

Input (t_s,p)	Live (t_s,t_e,p_{li},p_{lm})	Archive (t_s,t_e,p_{ai},p_{am})	Result (t_s,t_e,p_{li},p_{lm},p_{ai},p_{am})
(02:00,10)			
(02:01,6)	(02:00,02:01,10,6)		
(02:02,6)			
(02:03,5)	(02:02,02:03,6,5)		
(02:04,7)			
(02:05,6)	(02:04,02:05,7,6)	(02:02,02:04,6,7)	(02:02,02:05,7,6,6,7)
(02:06,11)			
(02:07,8)	(02:04,02:06,7,11)	(02:02,02:04,6,7) (02:04,02:06,7,11)	(02:02,02:07,11,8,6,7)
(02:08,8)			
(02:09,3)	(02:08,02:09,8,3)	(02:02,02:04,6,7) (02:04,02:06,7,11)	(02:02,02:09,8,3,6,7) (02:04,02:07,11,8,7,11) (02:04,02:09,8,3,7,11)
(02:10,3)			

Table 1: Lazy pattern matching - Baseline

section, we will study optimizations that reduce this additional cost by using additional memory.

A direct implementation of lazy pattern processing results in the following join algorithm, where `L` is the live source, `A` is the archive source, `P` is the recency region size, and `q` is the join predicate in the `WHERE` clause:

```
Hybrid_Loop_Join_PCQ(DStream L, DArchive A,
                     int P, Predicate q)
{
  FOR EACH match m_l over L
    FOR EACH match m_a over A[m_l.t_e-P,m_l.e-1]
        WHERE m_a.t_s < m_l.t_s
          AND m_a.t_s >= m_l.t_e-P
          AND m_a.t_e < m_l.t_e
      IF q(m_l, m_a) THEN
        append (m_l, m_a) to the result
}
```

Table 1 shows the execution of this algorithm on data from the running example of Figure 1. The first column (**Input**) shows the incoming simple events, each carrying a timestamp t_s and a price p. To clarify the presentation, we only show a single timestamp (start and end are the same) and have omitted the symbol information. The next two columns (**Live** and **Archive**) show the complex events produced as matches from the respective pattern specifications. The start (t_s) and end (t_e) timestamps are shown explicitly, since the computed complex events now cover a time period. Following the query specification in Figure 2, we show the initial price for each pattern: p_{li} and p_{ai}. Furthermore, the live events contain the minimum price (p_{lm}), and the archive events contain the maximum price (p_{am}). In the last column (**Result**), we show how the live and archive events are correlated to form the complete PCQ result.

Given the execution strategy of this lazy algorithm, we compute live events continuously in the outer loop. They are produced when the complete matching input is available. In this example, we have 5 matches of the live pattern (fall) in total showing up at 02:01, 02:03, ..., 02:09, each also covering the previous simple event. Based on these live matches, we can determine the recency region and compute the archive matches as required. Archive ("tick") events are produced at 02:05, 02:07, 02:09, when requested for the related live patterns. Their timestamps, however, correspond to the contributing simple events (such as 02:02 to 02:04). As one can see, archive pattern matches are computed several times due to the overlapping recency regions. When correlating live and archive events, the generated events are ordered first by start time and then by end time, as specified in Definition 5. The start time is determined by the archive match (following Definition 7), while processing occurs in live event order. We therefore need to sort archive/live match combinations, which incurs delays until the remaining live matches for an archive match have been handled. This can be seen in particular on result (02:02,02:09,8,3,6,7) and (02:04,02:07,11,8,7,11). The latter would have been already available with the arrival of the input tuple (02:07,8), but needs to go after the former, which is only available at the arrival of the input tuple (02:09,3).

Name	Description
N	total number of input tuples
c_l	average cost of live pattern processing per tuple (time)
c_a	average cost of archive pattern processing per tuple (time)
c_j	average cost of join per result (time)
M_l	total number of live matches
M_a	total number of archive matches
M_j	total number of join results
P_l	average number of input tuples inside the recency region of a live match
P_a	average number of input tuples inside the recency region of an archive match
c'_l	average cost of live pattern processing per tuple over recency region (time)
c'_a	average cost of archive pattern processing per tuple over recency region (time)
$\triangle_l$	average difference between two consecutive recency regions of live matches (# tuples)
$\triangle_a$	average difference between two consecutive recency regions of archive matches (# tuples)

Table 2: Workload cost factors for PCQs

When categorizing this join algorithm one can say that it is a hybrid between a nested-loop join (when P is large) and a sort-merge join (when $P << |A|$), since the matches in both streams are generated in an ordered fashion, and the recency region is also advancing monotonically. Other join types are not applicable for the following reasons: Hash joins only work for equi-joins, and band-joins [14] are typically designed to establish the order we already get for free from our processing model. The main step keeping us from an optimal join execution is the overlap among recency regions, for which we introduce the result cache in Section 4.3. Another small overhead is incurred by having to sort result events according to their start time, while the current algorithm produces them by end time. This overhead is fairly small, however, and sorting is not needed when changing the join order (Section 4.4).

From this algorithm, we can formulate a basic cost formula consisting of three main parts: (i) cost of the outer pattern processing, (ii) cost of the inner pattern processing, and (iii) cost of joining the results.

$$Cost_{proc} = N * c_l + M_l * P_l * c'_a + M_j * c_j \quad (1)$$

As shown in Table 2, we assume a constant average cost for processing a tuple in the first stream (c_l), in the active region of the second stream (c'_a) and in the join (c_j). The number of items processed in the second pattern depends on the selectivity of the first pattern (M_l) and the average number of tuples inside the recency region of a live match (P_l). $M_l * P_l$ can become larger than N if there are many live matches and/or a large recency region, so that the cost could exceed that of eager pattern processing. The optimizations in the next section will show methods to overcome this problem. The number of actual join results (M_j) or the cost of computing each of them (c_j) does not depend on the join/pattern processing order and can be treated as a constant in this work.

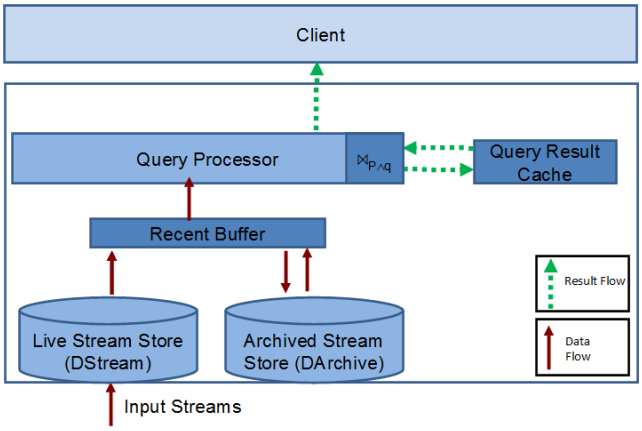


Figure 5: DejaVu CEP system architecture

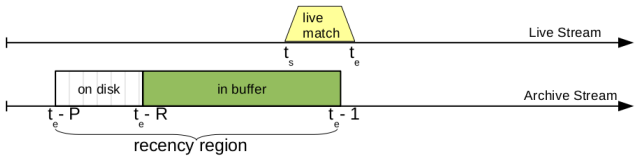


Figure 6: Recent Buffer

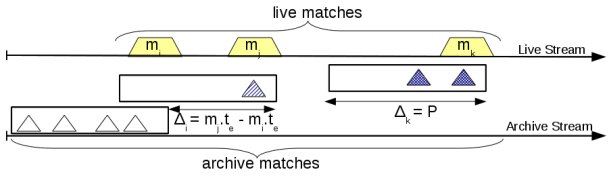


Figure 7: Query Result Cache

4. OPTIMIZING PCQ

As seen in the previous section, we need to address two main challenges in order to optimize recency-based PCQ: (i) managing and accessing the data efficiently, in particular creating a good working set from the archive, and (ii) efficiently processing the individual patterns and the correlations between them.

In this section, we will present three main optimization ideas (input buffering, query result caching, and join source ordering), that target these two challenges. We will also extend our cost model along the way, which helps us to analyze the impact of our proposed techniques.

We have studied and implemented our optimization techniques within the setting of the DejaVu CEP system. Therefore, we will first give a quick overview of DejaVu, before we dive into the details of our optimization techniques.

4.1 System Setting

DejaVu is a CEP system that integrates declarative pattern matching over live and archived streams of events [15]. DejaVu is built on top of MySQL [5] and implements a core subset of MATCH-RECOGNIZE [25].

Figure 5 shows the high-level architecture of DejaVu, including the three main extensions (colored in dark) that we have added for supporting and optimizing PCQs: (i) Recent Buffer, (ii) Query Result Cache, and (iii) PCQ operators. DejaVu's Live Stream Store (DStream) accepts live events into the system and feeds them to the Query Processor as they arrive, whereas its Archived Stream Store (DArchive) persists live events for long-term access. A fundamental design decision behind DejaVu was to route the live data to be archived through the Query Processor instead of directly storing it in a DArchive. As we will show, this approach not only enables important optimizations such as changing the order of join sources, but also simplifies working set maintenance.

4.2 Recent Input Buffering

To address our first challenge of managing and accessing the archive data during PCQ processing efficiently, we introduced the *Recent Buffer*. It is an in-memory data structure that mediates between the live and archived event stores (i.e., DStream as well as DArchive in DejaVu). By caching the most recent stream tuples, it provides the "hottest" subset of the DArchive with the same access costs and paths as for the DStream, thereby avoiding costly disk reads on recently archived data. Furthermore, it provides the means to perform bulk inserts into the DArchive.

Figure 6 depicts the structure of the *Recent Buffer*. P and R represent the recency region size (determined by the value of recency correlation distance defined in Section 2) and the *Recent Buffer* capacity, respectively. If $R \geq P$, then all the necessary tuples are stored in the *Recent Buffer*. If not, the *Recent Buffer* will only store the most recent R tuples. In such a case, the first portion of the recency region will be read from the disk by using a B+tree index on the start time values of the tuples.

The contents of the *Recent Buffer* need to be maintained as new inputs arrive into the DStream and as live matches are processed. The access properties of the patterns in our model (sequential scans, results only moving forward), and the recency region of size P lead to the following *Recent Buffer* maintenance approach: When there is a completed live match from time t_s to t_e, the region from the end of the match (i.e., $[t_e - P, t_e)$) needs to be considered for archive processing (see Definition 8). On the other hand, if no live match has been found yet and we are currently processing a tuple $T(t_s, t_e)$, we generally need to consider the region back from this tuple (i.e., $[T.t_e - P - 1, T.t_e]$), since a live match might be completed with the next arriving tuple. If a previous match m was found, however, we know that the next match will not end before $m.t_e$ (based on the `INCREMENTAL` mode discussion provided in Section 2). Therefore, we can take the maximum start points of these expected recency regions, ensuring that we never need more than P entries in the *Recent Buffer*. Given these properties, the *Recent Buffer* is implemented as a set of in-memory FIFO stores, with a separate one for each partition (specified with the `PARTITION BY` clause) in the query.

In addition to providing efficient memory management, our recent input buffering mechanism also facilitates our query optimization techniques, which we present next.

4.3 Query Result Caching

One important potential bottleneck of our baseline PCQ algorithm in Section 3 is the re-computation of pattern matching queries over the archived stream, because it leads to $\mathcal{O}(P_l * M_l)$ complexity.

One way to overcome this problem is based on the observation that the recency region of a live match intersects with the recency regions of other live matches, so that an archive match could be used by multiple live matches. This scenario resembles materialized views in traditional databases, where the access to the materialized results is faster than recomputing the view [17]. Figure 7 illustrates the relationship among such recency regions and the effect of using the *Query Result Cache* for different overlap scenarios. On the top lane, we see three live matches m_i, m_j, and m_k,

Live tuple (t_s,p)	Recent Buffer (t_s,p)	Query Result Cache (t_s,t_s,p_{ai},p_{am})	Result (t_s,t_e,p_{li},p_{lm},p_{ai},p_{am})
...			
(02:04,7)	(02:00,10) ... (02:04,7)		
(02:05,6)	(02:03,5) ... (02:05,6)	(02:02,02:04,6,7)	(02:02,02:05,7,6,6,7)
(02:06,11)	(02:03,5) ... (02:06,11)	(02:02,02:04,6,7)	
(02:07,8)	(02:05,6) ... (02:07,8)	(02:02,02:04,6,7) (02:04,02:06,7,11)	(02:02,02:07,11,8,6,7)
(02:08,8)	(02:05,6) ... (02:08, 8)	(02:02,02:04,6,7) (02:04,02:06,7,11)	
(02:09,3)	(02:05,6) ... (02:09,3)	(02:02,02:04,6,7) (02:04,02:06,7,11)	(02:02,02:09,8,3,6,7) (02:04,02:07,11,8,7,11) (02:04,02:09,8,3,7,11)
(02:10,3)	(02:05,6) ... (02:10,3)	(02:04,02:06,7,11)	

Table 3: Lazy Live First with Query Result Cache

(yellow tetragon) each spanning a recency region on the archive side (hollow rectangles) with archive matches (triangles). The recency regions of m_i and m_j overlap, making the effective distance $\triangle_j$ smaller than P_l and allow the reuse of the last archive match of m_i for m_j. In turn, $\triangle_k$ is equal to P_l, since there is no recency region overlap.

We can now adapt Equation (1) to only cover the computation of archive matches that have not been previously computed, covering a smaller portion of the recency region $\triangle_l$. Since cache retrieval is very cheap compared to pattern computation, we do not include it in our cost formula:

$$Cost_{proc} = N * c_l + M_l * Min(P_l, \triangle_l) * c'_a + M_j * c_j \quad (2)$$

Since we are only targeting the correlation of pattern queries, we do not need to burden ourselves with the overhead of general view maintenance [17], but can use a much simpler approach instead: The underlying data sources are append-only, and the recency regions advance monotonically. Therefore, a FIFO-based data structure is sufficient. In addition, the pattern correlation (Section 2) demands that there is only one pattern starting per time unit in the recency region; therefore, the maximum size of the *Query Result Cache* has a linear correlation with the size of the recency region.

Similar to the *Recent Buffer*, the *Query Result Cache* is also implemented as a set of stores, one for each partition in the query. According to our measurements and analysis, we expect the number of *Query Result Cache* entries to be much smaller than the number of *Recent Buffer* entries, since a complex event pattern can easily aggregate dozens or more input tuples. Nonetheless, using both of these data structures is still beneficial, since the *Recent Buffer* speeds up the computation of archive results at their first access, and can be shrunk to just the $\triangle_l$ regions. Furthermore, both can share a memory pool, since the *Query Result Cache* needs most capacity when there is matching, whereas the *Recent Buffer* needs most memory when not matching.

Table 3 illustrates the contents of the Recent Buffer and Query Result Cache for a snapshot of the execution trace of our running day trader example when our baseline algorithm (i.e., Lazy *Live First*) is enhanced with a *Query Result Cache*. The results show the expected behavior: *Query Result Cache* entries are added when they are computed on demand (e.g., live match (02:02,02:04,6,7) at input (02:05,6)) and stay in the cache until no recency region can

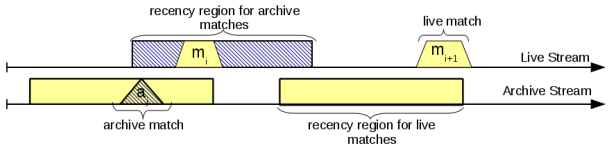


Figure 8: Join Source Ordering & Selectivities

cover them any more (e.g., at (02:10,3)). The *Recent Buffer* is now only needed for input items which are not covered by the *Query Result Cache*, so it only needs to start one item after the start of the most recent *Query Result Cache* entry (e.g., (02:03,5) at (02:05,6), when the archive match starts at 02:02).

4.4 Join Source Ordering

Traditional relational query optimization approaches have established several ways to optimize join queries with expensive predicates or nested subexpressions [19]. Yet, most of these approaches are not applicable to our problem, since they assume that the expensive operations can be freely moved around through the query plan. This is not possible in our case, since attributes of the patterns are used in the join.

Nonetheless, we can still exploit the opportunities for changing the order of join sources in our hybrid loop join algorithm, when the selectivity of live and archive patterns differs. This way, using the less selective partner on the outer side will also reduce the overall workload, as fewer recency regions will need to be inspected. A second factor comes from the observation that pattern queries have significant variances in their cost depending on their input data, e.g., when the number of match candidates varies greatly. This effect is amplified by the correlation operations we are performing, since the recency regions will change when changing the join order. In this case, changing the recency region might help skip the processing of the hot spots. Figure 8 gives an example of such a tradeoff: The upper lane with tetragons shows the live matches, the lower with triangles shows the archive matches. Rectangles shaded in the same way as the matches show the respective recency regions, where the archive recency regions is constructed in a forward manner. As one can see, *Live First* would compute two live matches and accordingly two recency regions, whereas *Archive First* would just compute a single archive match, and accordingly a single recency region.

As a result, our cost formula will have two variants:

$$Cost_{proc\_live\_first} = N * c_l + M_l * Min(P_l, \triangle_l) * c'_a + M_j * c_j \quad (3)$$

$$Cost_{proc\_archive\_first} = N * c_a + M_a * Min(P_a, \triangle_a) * c'_l + M_j * c_j \quad (4)$$

The above equations are conceptually symmetric, but they capture the different cost and selectivity distributions.

Since recency by itself is not symmetric, several design decisions were made so data source re-ordering would work efficiently: Our recency correlation function (Definition 8) covers the region from start of archive match to end of live match, and does not allow any part to exceed this range. Therefore, we can also easily process forward from an archive match to discover the relevant live matches. The recency regions actually processed will now be slightly different (starting from an archive match forward), but the same set of combined events will be produced. What will change, however, is the order in which the results are computed: Since archive matches drive the "outer loop", we will receive combined events ordered by their start time, not their end time (as in *Live First*). This is actually an advantage, since we can avoid reordering to achieve the desired start time ordering. The *Recent Buffer* can now be attached to the first data source of the join.

Input Tuple (t_s,p)	Recent Buffer (t_s,p)	Query Result Cache (t_s,t_e,p_{li},p_{lm})	Result (t_s,t_e,p_{li},p_{lm},p_{ai},p_{am})
	. . .		
(02:03,5)	(02:03,5)		
(02:04,7)	(02:03,5) (02:04,7)		
(02:05,6)	(02:03,5) . . . (02:05,6)	(02:04,02:05,7,6)	(02:02,02:05,7,6,6,7)
	. . .		
(02:07,8)	(02:03,5) . . . (02:07,8)	(02:04,02:05,7,6) (02:06,02:07,11,8)	(02:02,02:07,11,8,6,7)
	. . .		
(02:09,3)	(02:03,5) . . . (02:09,3)	(02:04,02:05,7,6) (02:06,02:07,11,8) (02:08,02:09,8,3)	(02:02,02:09,8,3,6,7) (02:04,02:07,11,8,7,11) (02:04,02:09,8,3,7,11)
(02:10,3)	(02:07,8) . . . (02:10,3)	(02:06,02:07,11,8) (02:08,02:09,8,3)	

Table 4: Lazy Archive First with Query Result Cache

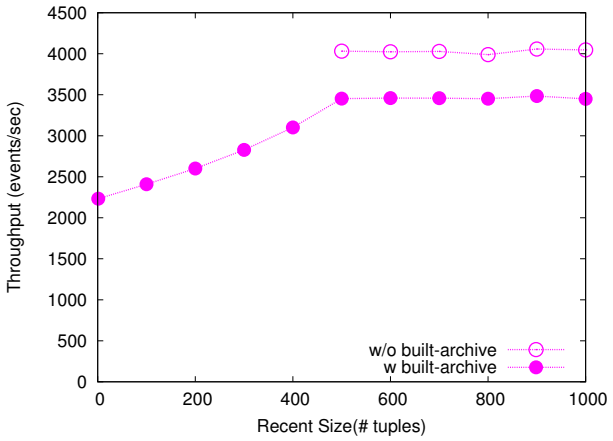


Figure 9: Archive storage and Recent Buffer, Q1, P_l=500

Table 4 illustrates the contents of the Recent Buffer and Query Result Cache for a snapshot of the execution trace of our running day trader example when the order of the join sources in our baseline algorithm are swapped (i.e., Lazy Archive First) is addition to using a *Query Result Cache*. The *Query Result Cache* now contains live matches which are collected while performing forward processing starting from the archive matches. They stay in the cache until archive processing overtakes them (e.g., at (02:10,3)). The *Recent Buffer* is also filled in a forward fashion, covering the start of the current archive match until the current data item. In contrast to *Live First* processing, restarting the archive computation after the previous starting point (e.g., (02:03,5) for the archive match from (02:02,6)) requires keeping the full recent region, not just the difference.

5. PERFORMANCE EVALUATION

5.1 Experimental Setup

DejaVu is built as an extension of MySQL-6.0.3-alpha, modifying its query parser, optimizer, run-time and storage engine. All experiments were executed on an Intel Xeon X3360 (2.8Ghz) with 8 GB RAM and a single 1 TB S-ATA disk, running Redhat Enterprise 5.4 64bit; the stock GCC 4.2.4 at optimization level -O2 was used for compilation, following the default setting for MySQL.

In order to evaluate the effectiveness of our optimization techniques and understand the factors in our cost model, we performed experiments in three directions:

- maintenance and access of archive data (recent buffer)
- query processing parameters (query result cache + join order)
- verification of our results on a real-life data set (NYSE TAQ)

We ran two variants of the financial query presented in Section 1.1. Both employ complex and costly patterns on the live and the archive clauses:

- *Q1* looks for a fall pattern on the live stream and correlates it with a "tick"-shaped pattern on the archive stream (see Figure 1 and Section 1.1).
- *Q2* also looks for a fall pattern on live stream but correlates it with a rise pattern on the archive stream. This query is used to model workloads where live and archive patterns are anti-correlated.

Workload variance in our setups comes from the input streams, which we adapt according to the parameters discussed in Section 4. We use a single input data stream with schema (t_{start}, t_{end}, $symbol$, $price$) and one tuple recorded at every time unit. This stream is pre-generated and pushed completely into the live store, from where it is pulled by DejaVu for query processing and archiving.

In our study, we focus on overall throughput, since it is most indicative of the processing cost. More specifically, throughput is defined as the number of input tuples processed per second, and compute it by dividing the total number input tuples by the total query processing time.

5.2 Evaluation of Recent Input Buffering

The first part of our analysis focuses on the overhead of storage management, both from creating the archive and reading data out of it. Figure 9 shows (i) the impact of different recent buffer sizes when an archive is maintained, and (ii) the gain in throughput when not building an archive. Query processing is done in a naive way (i.e., live-first, no query result cache); we will study query processing parameters in the next section. Figure 9 verifies that, to avoid having to read from disk when performing archive pattern processing, the recent buffer should be at least as big as the recency region size (P_l), in this case 500 tuples (and cannot be smaller than this when "w/o built-archive"). If a smaller recent buffer is used (due to, e.g., limited memory), the performance degrades proportionally to the available buffer size, accounting for the tuples having to be fetched from the portion of the archive that is on disk. In this case, since fetching can by done using an index scan and the cache replacement policy of the recent buffer always discards the oldest tuples, we still achieve a throughput value better than 2500 tuples/second. Archive building causes a more-or-less constant overhead; it can only be avoided if the recent buffer is large enough to cover the recency region completely. Overall, archive reading is clearly a bottleneck when doing naive query execution. On the other hand, the focus of this work is not to optimize stream archiving, but to optimize query processing for PCQs that utilize an underlying stream archive. By introducing an in-memory recent buffer component into our architecture, we support the query processor in accessing the more recent archive data faster. We therefore run the rest of our experiments with a sufficiently big recent buffer.

5.3 Evaluation of Query Processing

The main part of our performance study focuses on query processing, as it has the most profound impact on throughput and a larger set of parameters to explore. We vary the input data to stress

P_l		10	50	100	200	300	400	500	600	700	800	900	1000
No Result Cache	Max Recent Items	10	50	100	200	300	400	500	600	700	800	900	1000
Query Result Cache	Max Recent Items	10	49	49	49	49	49	49	49	49	49	49	49
	Max Cache Entries	0	1	3	7	11	15	19	23	27	31	35	39

Table 5: Item count for Recent Buffer and Query Result Cache, varying P_l (Figure 10)

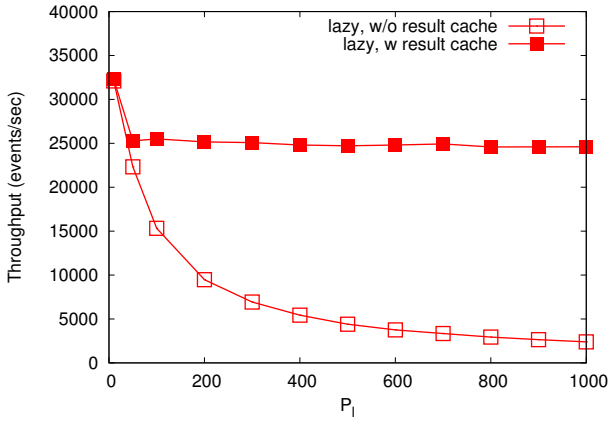


Figure 10: Varying P_l; Q1; fixed M_l, $\triangle_l$, and c'_a

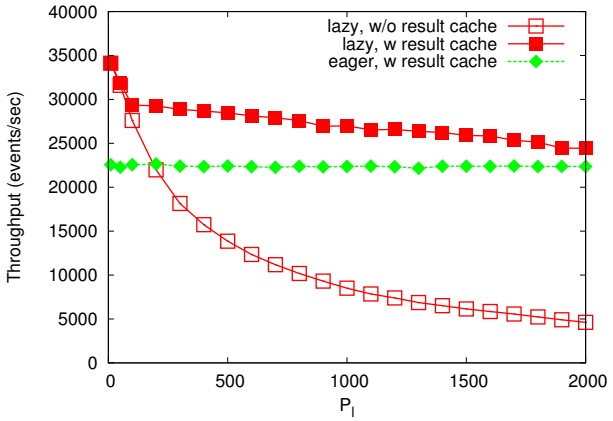


Figure 11: Varying P_l; Q1; fixed M_l, $\triangle_l$, and c'_a

particular cost factors, in particular investigating the impact of the recency region size, the presence of the query result cache, and join source ordering.

Figure 10 illustrates the performance impact of varying the size of the archive correlation region, which is the core of our correlation model. As predicted by the cost model, we see a quadratic cost increase (and accordingly a throughput decrease) as soon as the recency region size is large enough to actually produce archive pattern matches. Hence, we need to recompute all these matches for every live match (similar to a nested-loop join). Using the result cache, we can reduce this cost, since each of these matches only needs to be computed once. Therefore, the cost is now proportional to the number of results, which results in a linear throughput decrease (similar to a hash join).

We also measured the maximum utilization of both the recent buffer and the result cache, to see the required capacity. The results are shown in Table 5: When no result cache is used, the recent buffer usage is as big as P_l, since we always have to keep the full recency region available. When a result cache is used, this changes significantly, since the recent buffer is used only when "sliding" between live matches. In our test dataset, it is dominated by the difference between live matches, which is 49 tuples. The number of entries in the result cache is linear to the size of P_l, since our model ensures distinct starting points for the matches (Section 2). Furthermore, it is often much smaller than the number of input tuples, since a pattern clause consumes many input tuples to produce a single output tuple.

As a result, the total memory consumption is linear to P_l, ensuring good scalability for higher values of P_l.

5.3.1 Lazy vs. Eager Pattern Processing

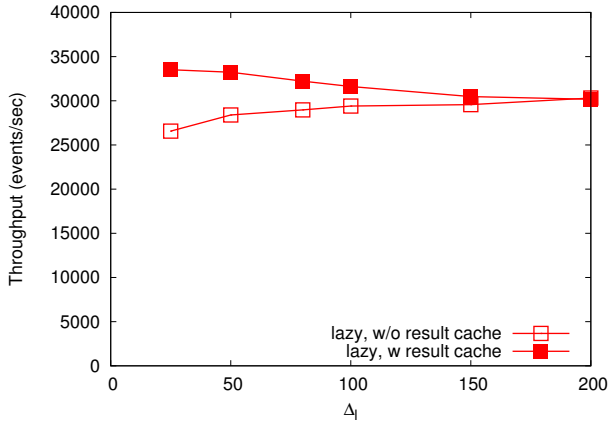
Figure 11 compares the throughput of the system, when different processing strategies are used. As explained in Section 3, the *Eager* strategy computes both live and archive patterns over the whole dataset, which makes its performance independent of the recency correlation distance (P_l). This can be attributed to the observation that the join between live and archive matches is significantly cheaper than the pattern processing itself. Due to the independence from P_l, the throughput of the *Eager* strategy stays constant with varying recency correlation values. As the figure shows, the *Eager* strategy performs better than *Lazy w/o result cache* since the latter has quadratic cost on the archive side (as the cost formula from Section 3 and 4 predicts). On the other hand, *Lazy with result cache* outperforms *Eager* for small P_l values since it can skip unneeded parts on the archive side, whereas *Eager* needs to compute patterns on these. As P_l increases and the gaps on the archive side shrink, the cost of *Lazy with result cache* converges with *Eager*, since both need to perform the same workload. Yet as the cost model predicts, *Eager* never actually outperforms *Lazy with result cache* in terms of throughput. Given this result, we have focused on the *Lazy* strategy for our remaining experiments.

5.3.2 Query Result Cache Sensitivity

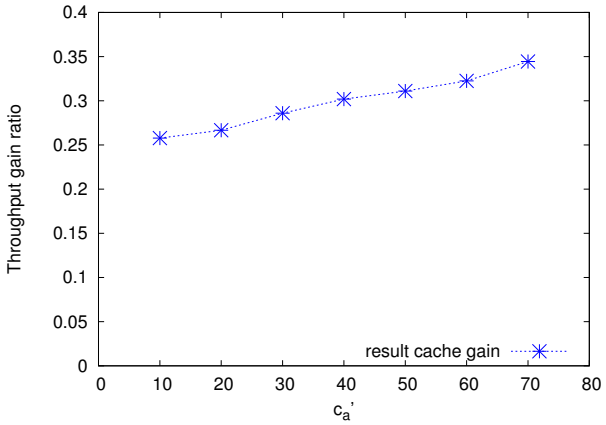
The effectiveness of the query result cache depends on two factors: The overlap between the consecutive recency regions and the cost of archive processing. As Figure 12a shows, a decreasing overlap $\triangle_l$ (i.e., the distance between two consecutive recency regions) entails a decrease in throughput, due to the lower cache utilization. When $\triangle_l$ is as big as P_l (and thus no overlap exists), cache and no-cache performance converges, showing that there is no performance overhead in employing an unused cache. This can be explained by the very small cost of a maintaining and probing an in-memory queue compared to pattern matching operations.

Since the query result cache eliminates the cost of recomputing the pattern on the second data source, its benefit becomes more pronounced the more expensive the second pattern query is. Figure 12b illustrates this trend by comparing the relative throughput gain $((TP_{cache} - TP_{nocache})/TP_{nocache})$ of the cache-based approach when modifying the cost of archive processing (c'_a).

As a result of our query result cache experiments, we can clearly see that a result cache should be used whenever there is no extreme memory shortage. It enables significant performance gains on appropriate workloads, does not carry a measurable performance overhead in the worst case, and has very modest memory consumption.



(a) Varying $\triangle_l$; Q1; fixed M_l, $P_l = 200$, and c'_a



(b) Varying c'_a; Q1; fixed M_l, $\triangle_l = 80$, and $P_l = 200$

Figure 12: Impact of Query Result Cache when $\triangle_l$ or c'_a is varied

5.3.3 Evaluation of Join Source Ordering

As outlined in Section 4.4, changing the join order is beneficial if there is a significant selectivity or cost difference between the two join inputs. If such a difference exists, the more selective/less costly input is best used as the outer loop. In our case, there is a potential for optimization even if the two join partners have the same overall selectivity: The correlation of live and archive matches can create "empty" regions or "hotspots", which can then be skipped.

The join order experiments are performed in both workloads where the result cache is beneficial, as well as in workloads where it is not. Doing both yields additional insight, since join order and result cache can have competing effects: Join order works best in bringing the most selective join partner to the outer loop, whereas result caching reduces the cost of the inner loop.

We used *Q2* in these experiments, since the archive pattern of *Q1* contains the live pattern, thus fixing both the selectivity and cost ratios: There is a fall pattern instance (live) inside every "tick"-shaped (fall followed by increase) pattern (*Q1* archive) instance, which prohibits varying relative cost and selectivities among the two sources. Section 5.4 shows the results of running *Q1* in both join orders on our real-life data set, showing that our results for *Q2* also apply for *Q1* on a realistic data set.

In the first set of the experiments, the cost of recency region processing is the same for live- and *Archive First* processing ($c'_l = c'_a$), and the cost of first pattern processing is set to a value that is proportional to the number of the first pattern matches ($c_l \propto M_l$and$c_a \propto M_a$). Hence, according to our cost formula, the total processing cost depends on the selectivity of archive and live matches (M_l and M_a).

We varied the match ratio in the first experiment (see Figure 13a). Let us first discuss the case where the workload does not benefit from a result cache (empty square and triangle in the graph). If there is no difference in the selectivity of live and archive matches, the performance is the same. When there is a selectivity difference, then the pattern which is less likely to be matched should be processed first.

When the workload is amenable for result cache usage (filled square and triangle in Figure 13a), the trend is similar to the one in the *without result cache* scenario, but the relative benefits are smaller. Although the cost for the first pattern stays the same, processing the second pattern becomes cheaper, since the result cache eliminates the quadratic overhead.

In the second set of experiments, we keep the selectivity of both patterns at the same level ($M_l = M_a$), but change the correlations between the recency regions and thus introduce variance in the "local" cost. In other words, changing the join order might help to skip the computationally expensive areas in the stream. As a result, the average cost of processing a tuple over the whole stream is much lower than processing it in the recency region.

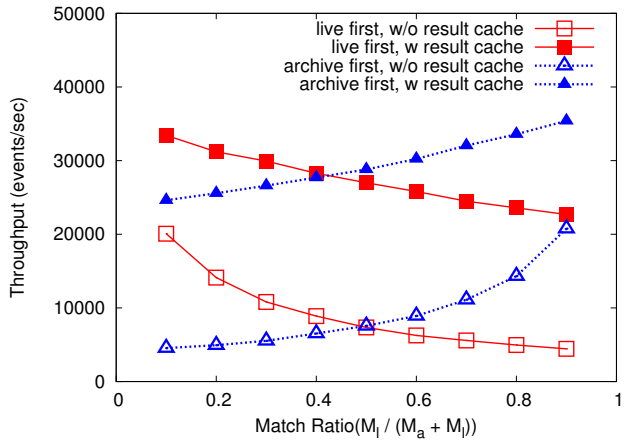
Figure 13b shows our results. If the result cache is not effective (empty square and triangle in Figure 13b), the performance of the system increases when the less costly recency region is preferred. On the other hand, when a result cache is actually applicable, it leads to a significant cost reduction in processing the second data source, so that the join order has almost no impact (filled square and triangle in Figure 13b). Archive-first processing always performs slightly better than life-first, since it can access the recent buffer more efficiently, it can benefit from forward processing in the pattern, and does not need to re-order the final results.

These experiments clearly demonstrate the benefits of join ordering in PCQs. The performance factors are orthogonal, so we expect to see even better results when there is a skew in both selectivity and cost/correlation.

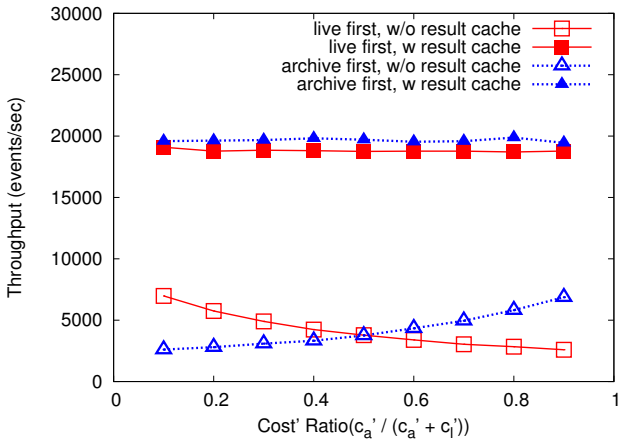
5.4 Results on Real-Life Data

Our experiments so far have been performed on synthetic data which was geared towards evaluating the cost model factors. In order to understand how relevant our optimizations are in real-world workloads, we took several days of stock-market data from NYSE (January 26 to 31, 2006), and selected the most heavily traded stock (Exxon Mobile, symbol XOM). We again ran *Q1*, and extended P_l to cover the equivalent of several hours. As Figure 14 shows, the total throughput is somewhat lower than for the synthetic data (Figure 10), whereas the benefit of result caching is even more pronounced on larger P_l, giving a factor of 54 performance gain for P_l=500 between the baseline approach (*Live First without cache*) and the best method (*Archive First with Cache*).

Although the actual numbers are different, we see the same trend regarding the benefit of result caching. The main reason behind the different throughput numbers is that real life data is more likely to produce matches than synthetic data. This means more recency



(a) Varying selectivity; Q2; c_l'=c_a', $c_l \propto M_l$, $c_a \propto M_a$, $P_l = P_a = 500$



(b) Varying cost ratio; Q2; $c_l << c_a'$, $c_a << c_l'$, $M_l = M_a$, $P_l = P_a = 500$

Figure 13: Impact of Join Source Ordering when match selectivities or recency region processing cost is varied

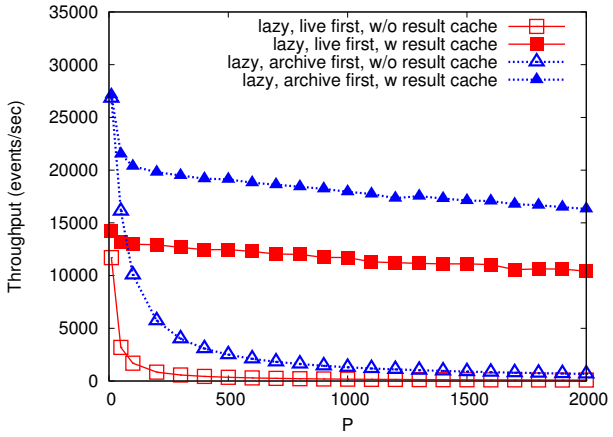


Figure 14: Real-life data, Q1, varying P, Lazy Live First & Lazy Archive First, with or without Query Result Cache

regions overlap more and more matches in the recency regions. Another important observation regarding the real life experiment is that *Archive First* processing achieves a better throughput. The reason behind it is that the archive pattern is more selective than the live pattern. As a reminder, each "tick"-shaped pattern instance includes at least one fall pattern instance, and "tick"-shaped pattern is less likely to be matched than a fall pattern. Here, the selectiviy difference dominates the join source ordering decision and makes *Archive First* with result cache the winner strategy.

5.5 Discussion

In our experiments, we have studied the impact of the different processing approaches and optimizations on throughput, both on synthetic and real-life data. The results lead to several clear conclusions: *Eager* is nearly always outperformed by *Lazy with Result Cache*, in the best case it manages to perform equally well. Therefore, there is little motivation to use *Eager* when focusing on throughput. Among the optimizations, using a result cache is always useful when there is at least some free memory available, as it significantly speeds up processing when overlaps exists yet has extremely low overhead when none exists. In addition, memory requirements are low and well-bounded. In a similar fashion, a recent input buffer should be used whenever there is enough memory, since it significantly reduces the cost of archive access. The usefulness of join order depends on a skew in selectivity or correlation among the two inputs. If such a skew exists, it also provides measurable benefits. To sum up, all the optimizations we have investigated provide significant benefits, sometimes improving results by an order of magnitude or more.

6. RELATED WORK

There are several CEP engines (e.g., Cayuga [13], SASE [10], ZStream [21]) that propose languages and processing techniques for efficient pattern matching over live streams, but none of them supports stream archiving or pattern correlation queries. On the other hand, the DataDepot stream warehousing system [16] has been designed to automate the ingestion of streaming data from a variety of sources, as well as to maintain complex materialized views over them, but it does not provide any facilities for PCQs.

The need for combining the processing of live and historical events has also been recognized by previous work (e.g., Moirae [11], TelegraphCQ [12, 22], NiagaraST/Latte [24]), and they mostly tackle the efficient archive access problem. Moirae [11] prioritizes the processing of the recent historical data and produces approximate results by using multi-level storage, recent event materialization, and context similarity metrics. TelegraphCQ proposes an overload-sensitive disk access method where multiple samples of a stream archive are created and are selectively accessed depending on the input rates [12], and a bitmap-based indexing technique for efficiently querying live and historical stream data [22]. NiagaraST/Latte demonstrates how hybrid queries can be used for travel time estimation in intelligent transportation systems [24]. The focus is on window-based hybrid queries (no pattern matching) and on efficient archive access based on different similarity notions. Our recency-based correlation criteria falls under Latte's structural similarity notion.

Typical commercial SPEs also support hybrid continuous queries (e.g., [1], [8], [9]), where for every new item on the stream, a

database query needs to be executed. We generalize this to finding correlated archive pattern matches for every new pattern match on the stream. As such, we face the additional challenges of variable processing window sizes on live and archive, variable archive scope due to variable slide on live, and more complex window computations.

Last but not least, Oracle CEP [7] and ESPER [2] also provide a basic implementation for MATCH-RECOGNIZE, though with some limitations that render their use not feasible for processing PCQs. More specifically, application-time based processing of MATCH-RECOGNIZE is not currently supported in Oracle CEP engine, whereas ESPER does not support joins across MATCH-RECOGNIZE clauses.

7. CONCLUSIONS AND FUTURE WORK

In this work, we have investigated the problem of efficiently correlating complex events over live and archived data streams, which we call Pattern Correlation Queries (PCQs). Applications for PCQs span various domains (such as financial or medical), which need real-time forecasting of events or identification of causal dependencies, while handling data at high rates and in massive amounts. In the paper, we first defined the formal semantics for recency-based PCQs, paying attention to usefulness, clarity, composability, and optimizability. After studying the state of the art and possible baseline algorithms for implementing PCQs according to their complexity, we have proposed three optimizations for efficient data management and query processing, including recent input buffering, query result caching, and join source ordering. An extensive performance study on synthetic and real-life data sets not only validates our cost model, but also shows that our optimizations are very effective, achieving more than two orders magnitude throughput improvements and much better scalability compared to a straightforward implementation.

In terms of future work, we plan to pursue additional optimizations such as result cache value indexing for further join optimizations and exploiting similarity of patterns to reduce matching cost. We also consider investigating how other performance criteria, such as response time, are affected by our design decisions and which optimizations are needed if they become more relevant. Another important direction to study is how other correlation criteria such as context similarity or temporal periodicity would work with our architecture and optimizations. We think that the role of the recent buffer will be reduced, whereas result caching will become even more important.

Acknowledgments.

We would like to thank Patrick Lau for his contribution in the development of DejaVu engine. This work has been supported in part by the Swiss NSF ProDoc PDFMP2-122971/1 grant.

8. REFERENCES

[1] Coral8, Inc. `http://www.coral8.com/`.

[2] ESPER. `http://esper.codehaus.org/`.

[3] MEDAN - Competence Center for Medical Data Warehousing and Analysis. `http://www.inf.unibz.it/dis/projects/medan/index.html`.

[4] Medical Use case. `http://www.mamashealth.com/Bloodpressure.asp`.

[5] MySQL. `http://www.mysql.com/`.

[6] NYSE Data Solutions. `http://www.nyxdata.com/nysedata/`.

[7] Oracle CEP. `http://www.oracle.com/technetwork/middleware/complex-event-processing/index.html/`.

[8] StreamBase Systems, Inc. `http://www.streambase.com/`.

[9] Truviso, Inc. `http://www.truviso.com/`.

[10] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient Pattern Matching over Event Streams. In *ACM SIGMOD Conference*, Vancouver, Canada, June 2008.

[11] M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee. Moirae: History-Enhanced Monitoring. In *CIDR Conference*, Asilomar, CA, January 2007.

[12] S. Chandrasekaran and M. Franklin. Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams. In *VLDB Conference*, Toronto, Canada, August 2004.

[13] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A General Purpose Event Monitoring System. In *CIDR Conference*, Asilomar, CA, January 2007.

[14] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An Evaluation of Non-Equijoin Algorithms. In *VLDB Conference*, Barcelona, Spain, September 1991.

[15] N. Dindar, B. Güç, P. Lau, A. Özal, M. Soner, and N. Tatbul. DejaVu: Declarative Pattern Matching over Live and Archived Streams of Events (Demo). In *ACM SIGMOD Conference*, Providence, RI, June 2009.

[16] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream Warehousing with DataDepot. In *ACM SIGMOD Conference*, Providence, RI, June 2009.

[17] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2), 1995.

[18] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. On Supporting Kleene Closure over Event Streams. In *IEEE ICDE Conference*, Cancun, Mexico, April 2008.

[19] J. M. Hellerstein. Optimization Techniques for Queries with Expensive Methods. *ACM TODS Journal*, 23(2), June 1998.

[20] A. Lerner and D. Shasha. The Virtues and Challenges of Ad Hoc + Streams Querying in Finance. *IEEE Data Engineering Bulletin*, 26(1), March 2003.

[21] Y. Mei and S. Madden. ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events. In *ACM SIGMOD Conference*, Providence, RI, June 2009.

[22] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. Enabling Real-Time Querying of Live and Historical Stream Data. In *SSDBM Conference*, Banff, Canada, July 2007.

[23] J. Teubner and R. Müller. How Soccer Players Would Do Stream Joins. In *SIGMOD*, 2011.

[24] K. Tufte, J. Li, D. Maier, V. Papadimos, R. L. Bertini, and J. Rucker. Travel Time Estimation using NiagaraST and Latte. In *ACM SIGMOD Conference*, Beijing, China, June 2007.

[25] F. Zemke, A. Witkowski, M. Cherniack, and L. Colby. Pattern Matching in Sequences of Rows. Technical Report ANSI Standard Proposal, July 2007.