ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events

Yuan Mei
MIT CSAIL
Cambridge, MA, USA
meiyuan@csail.mit.edu

Samuel Madden
MIT CSAIL
Cambridge, MA, USA
madden@csail.mit.edu

ABSTRACT

Composite (or Complex) event processing (CEP) systems search sequences of incoming events for occurrences of user-specified event patterns. Recently, they have gained more attention in a variety of areas due to their powerful and expressive query language and performance potential. Sequentiality (temporal ordering) is the primary way in which CEP systems relate events to each other. In this paper, we present a CEP system called ZStream to efficiently process such sequential patterns. Besides simple sequential patterns, ZStream is also able to detect other patterns, including conjunction, disjunction, negation and Kleene closure.

Unlike most recently proposed CEP systems, which use non-deterministic finite automata (NFA's) to detect patterns, ZStream uses tree-based query plans for both the logical and physical representation of query patterns. By carefully designing the underlying infrastructure and algorithms, ZStream is able to unify the evaluation of sequence, conjunction, disjunction, negation, and Kleene closure as variants of the join operator. Under this framework, a single pattern in ZStream may have several equivalent physical tree plans, with different evaluation costs. We propose a cost model to estimate the computation costs of a plan. We show that our cost model can accurately capture the actual runtime behavior of a plan, and that choosing the optimal plan can result in a factor of four or more speedup versus an NFA based approach. Based on this cost model and using a simple set of statistics about operator selectivity and data rates, ZStream is able to adaptively and seamlessly adjust the order in which it detects patterns on the fly. Finally, we describe a dynamic programming algorithm used in our cost model to efficiently search for an optimal query plan for a given pattern.

Categories and Subject Descriptors

H.4.m [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

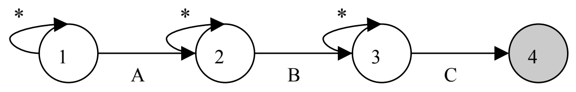

Figure 1: An example NFA for processing the sequential pattern A followed by B followed by C

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Complex Event Processing, Streaming, Optimization, Algorithm

1. INTRODUCTION

Composite (or Complex) event processing (CEP) systems search sequences of incoming events for occurrences of user-specified event patterns. They have become more popular in a number of areas due to their powerful and expressive query language and performance potential [1]. Sequential queries (based on temporal ordering) are the primary way in which CEP systems relate events to each other. Examples of sequential queries include tracing a car's movement in a predefined area (where a car moves through a series of places), detecting anomalies in stock prices (where the rise and fall of the price of some stocks is monitored over time), and detecting intrusion in network monitoring (where a specific sequence of malicious activities is detected). However, purely sequential queries are not enough to express many real world patterns, which also involve conjunction (e.g., concurrent events), disjunction (e.g., a choice between two options) and negation, all of which make the matching problem more complex.

Currently, non-deterministic finite automata (NFA) are the most commonly used method for evaluating CEP queries [7, 8, 15]. As shown in Figure 1, an NFA represents a query pattern as a series of states that must be detected. A pattern is said to be matched when the NFA transitions into a final state. However, the previously proposed NFA-based approaches have three limitations that we seek to address in this work:

Fixed Order of Evaluation. NFA's naturally express patterns as a series of state transitions. Hence, current NFA-based approaches impose a fixed evaluation order determined by this state transition diagram. For example, the NFA in Figure 1 starts at state 1, transits to state 2

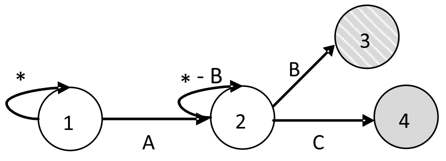

Figure 2: An example NFA for processing the negation pattern A followed by C without any B in between

when event A occurs, then to state 3 when B occurs, and finally to the output state when C occurs. A typical order that NFA-based model evaluates queries is by performing backward search starting from the final state 4. However, there are several other detection orders that are possible — for example, the system may wait for an instance B to occur and then search backward in time to find previous A instances and forward to find C instances. This would be a more efficient plan if B is much less common than A and C. As in traditional database join planning, a fixed evaluation order can be very inefficient. Though it might be possible to build different NFA's, or multiple sub-NFA's, to evaluate a given query in a different order, to the best of our knowledge prior work has not explored this possibility.

Negation. It is not straightforward to efficiently model negation (events that do not occur) in an NFA when there exist predicates between the negated and non-negated events. Figure 2 shows an NFA that could be used to process negation pattern "A followed by C without any event B in between". State 3 in Figure 2 is a terminal state reached after a B event is received; no output is produced in this case. State 4 is a final state indicating a match for the pattern. However, if there is a predicate involving B and C (e.g., as in the pattern "A followed by C such that there is no interleaving B with $B.price > C.price$"), there is no simple way to evaluate the predicate when a B event arrives, since it requires access to C events that have yet to arrive. Hence, it has difficulty to decide a B event transition. As a result, this NFA cannot be used for negation queries with predicates. For this reason, existing NFA-systems perform negation as a post-NFA filtering step [15].

Concurrent Events. As with negation, it is also hard to support concurrent events, such as conjunction queries (e.g., A and B), in an NFA-based model because NFA's explicitly order state transitions.

In this paper, we describe the design and implementation of a CEP system called ZStream. ZStream addresses the above limitations of previously proposed approaches, including the following key features:

1. We develop a tree-based query plan structure for CEP queries that are amenable to a variety of algebraic optimizations.
2. We define a notion of an *optimal plan* and show that ZStream can effectively search the space of possible plans that correspond to a specified sequential pattern. We show that choosing the optimal plan ordering can improve performance by a factor of four or more over a fixed plan as would be chosen by an NFA approach.
3. We show that ZStream is able to unify the evaluation of sequence, conjunction, and disjunction as variants of the join operator and also enables flexible operator ordering.
4. We propose a formulation of negation queries so they can be incorporated into this tree-based model just like other operators, rather than being applied as a final filtration step, which we find improves the throughput of negation queries by almost an order of magnitude.
5. We demonstrate how ZStream can adaptively change the evaluation plan as queries run and that an adaptive plan can outperform a static plan significantly when characteristics of incoming streams change (e.g., event rates or selectivity).

Our goal in this paper is not to demonstrate that an NFA-based approach is inherently inferior to a tree-based query-plan approach, but to show that

1. by carefully designing the underlying infrastructure and algorithms, a tree-based approach can process most CEP queries very efficiently, and,
2. without some notion of an optimal plan, as well as statistics and a cost model to estimate that optimal plan, any approach that uses a fixed evaluation order (whether based on NFA's or trees) is suboptimal.

The rest of the paper is organized as follows: Section 2 and 3 introduces related work and language respectively. Section 4 presents the system architecture and operator algorithms; Section 5 discusses the cost model and optimization; Section 6 shows evaluation results.

2. RELATED WORK

CEP systems first appeared as trigger detection systems in active databases [5, 6, 9, 10] to meet the requirements of active functionality that are not supported by traditional databases. Examples of such work include HiPAC [6], Ode [10], SAMOS [9] and Sentinel [5]. FSA-based systems, such like HiPAC and Ode, have difficulty supporting concurrent events, because transitions in FSAs inherently incorporate some orders between states. Petri Nets-based systems such like SAMOS are able to support concurrency, but such networks are very complex to express and evaluate. Like ZStream, Sentinel evaluates its event language using event trees, but it arbitrarily constructs a physical tree plan for evaluation rather than searching for an optimal plan, which, as we show, can lead to suboptimal plans.

Other research has tried to make use of string-based matching algorithms [13] and scalable regular expressions [12] to evaluate composite event patterns. These methods only work efficiently for strictly consecutive patterns, limiting the expressive capability of the pattern matching language. In addition, they typically focus on searching for common sub-expressions amongst multiple patterns rather than searching for an optimal execution plan for a single pattern.

To support both composite event pattern detection and high rate incoming events, SASE [4, 15], a high performance CEP system, was recently proposed. It achieves good performance through a variety of optimizations. SASE is, however, NFA-based, inheriting the limitations of the NFA-based model described in Section 1. The Cayuga [7] CEP system is another recently proposed CEP system. Since it is developed from a pub/sub system, it is also focused on common sub-expressions searching amongst multiple patterns. In addition, Cayuga is also NFA-based; hence it too suffers from the limitations of the NFA-based model.

Recently, commercial streaming companies like Stream-

Base [2] and Coral8 [3] have added support for SQL-like pattern languages as well. These companies apparently use join-style operators to support pattern detection. However, their approach is not documented in the literature, and, as of this writing, our understanding is that they do not have a cost model or optimization framework specifically for pattern queries.

3. LANGUAGE SPECIFICATION

In this section, we briefly review the CEP language constructs we use in ZStream. The language we adopt here is a simplified version of the languages used in other CEP systems [2, 4, 15].

We begin with a few basic definitions. **Primitive events** are predefined single occurrences of interest that can not be split into any smaller events. **Composite events** are detected by the CEP system from a collection of primitive and/or other composite events. **Single-class predicates** are predicates that involve only one event class and **multi-class predicates** are predicates that involve more than one event class. Primitive events arrive into the CEP system from various external event sources, while composite events are internally generated by the CEP system itself. CEP queries have the following format (as in [15]):

```
PATTERN   Composite Event Expressions
WHERE     Value Constraints
WITHIN    Time Constraints
RETURN    Output Expression
```

The *Composite Event Expressions* describe an event pattern to be matched by connecting event classes together via different event operators; *Value Constraints* define the context for the composite events by imposing predicates on event attributes; *Time Constraints* describe the time window during which events that match the pattern must occur. The RETURN clause defines the expected output stream from the pattern query.

Query 1. Sequence Pattern

PATTERN	$T1; T2; T3$
WHERE	$T1.name = T3.name$
AND	$T2.name =$ *'Google'*
AND	$T1.price > (1 + x\%) * T2.price$
AND	$T3.price < (1 - y\%) * T2.price$
WITHIN	10 *secs*
RETURN	$T1, T2, T3$

Query 1 shows an example stock market monitoring query that finds a stock with its trading price first $x\%$ higher than the following *Google* tick, and then $y\%$ lower within 10 seconds. The stock stream has the schema *(id, name, price, volume,ts)*. The symbol ";" in the PATTERN clause is an operator that sequentially connects event classes, meaning the left operand is followed by the right operand.

In our data model, each event is associated with a start-timestamp and an end-timestamp. For primitive events, the start-timestamp and end-timestamp are the same, in which case we refer to a single timestamp. Some CEP systems make the assumption that a composite event occurs at a single point in time (the end-timestamp for instance), and ignore the event duration. This assumption makes the semantics of composite event assembly unclear. For example, suppose the composite event results generated from Query 1 are further used as inputs to another sequential pattern "$A; B$ WITHIN tw". Then simply using the end-timestamps (to satisfy the new pattern's time window tw) may result in the total elapsed time between the start of A and the end of B exceeding the time bound tw. In other words, simply matching on end-timestamp can result in composite events with an arbitrarily long occurrence duration. Hence, in ZStream, we require that composite events have a total **duration** less than the time bound specified in the WITHIN clause.

3.1 Event Operators

Event operators connect primitive or composite events together to form new composite events. This section briefly describes the set of event operators supported in our system, and presents more example queries.

Sequence $(A; B)$**:** The sequence operator finds instances of event B that follow event A within a specified time window. The output is a composite event C such that $A.end\text{-}ts < B.start\text{-}ts$, $C.start\text{-}ts = A.start\text{-}ts$, $C.end\text{-}ts = B.end\text{-}ts$ and $C.end\text{-}ts - C.start\text{-}ts \leq time\ window$.

Negation $(!A)$**:** Negation is used to express the non-occurrence of event A. This operator is usually used together with other operators; for example "$A; !B; C$" indicates that C follows A without any interleaving instances of B.

Conjunction $(A\&B)$**:** Conjunction (i.e., concurrent events) means both event A and event B occur within a specified time window, and their order does not matter.

Disjunction $(A|B)$**.** Disjunction means either event A or event B or both occurs within a specified time window. Since we allow both A and B to occur in disjunction, it is simply a union of the two event classes and also satisfies the time constraint.

Kleene Closure $(A^*/A^+/A^{num})$**.** Kleene closure means that event A can occur zero or more (*) or one or more (+) times. ZStream also allows the specification of a closure count to indicate an exact number of events to be grouped. For example, A^5 means five A instances will be grouped together.

3.2 Motivating Applications

Sequential patterns are widely used in a variety of areas, from vehicle tracking to system monitoring. We have illustrated a typical sequential pattern in Query 1. In this section, we show several more sequential patterns from stock market monitoring. The input stream has the same schema as that of Query 1.

Query 2. Negation Pattern

PATTERN	$T1; !T2; T3$
WHERE	$T1.name = T2.name = T3.name$
AND	$T1.price > x$
AND	$T2.price < x$
AND	$T3.price > x * (1 + 20\%)$
WITHIN	10 *secs*
RETURN	$T1, T3$

Query 3. Kleene Closure Pattern

PATTERN	$T1; T2^5; T3$
WHERE	$T1.name = T3.name$
WHERE	$T2.name =$ *'Google'*
AND	$sum(T2.volume) > v$
AND	$T3.price > (1 + 20\%) * T1.price$
WITHIN	10 *secs*
RETURN	$T1, sum(T2.volume), T3$

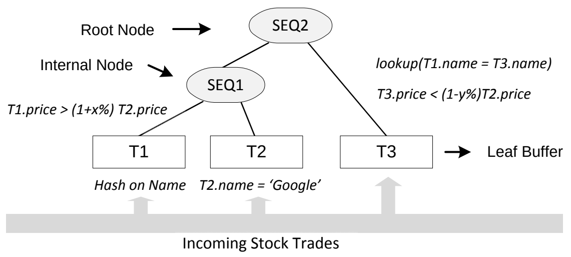

Figure 3: A left-deep tree plan for Query 1

Query 2 illustrates a negation pattern to find a stock whose price increases 20% from some threshold price x without any lower price in between during a 10 second window. Query 3 shows a Kleene closure pattern to aggregate the total trading volume of *Google* stock. This pattern is used to measure the impact on other stocks' prices after high volume trading of *Google* stock. The expression $T2^5$ constrains the number of successive *Google* events in the Kleene closure to 5; the aggregate function $sum()$ is applied to the attribute *volume* of all the events in the closure.

4. ARCHITECTURE AND OPERATORS

In this section, we describe the system architecture and evaluation of query plans, as well as algorithms for operator evaluation.

4.1 Tree-Based Plans

To process a query, ZStream first parses and transforms the query expression into an internal tree representation. *Leaf buffers* store primitive events as they arrive, and *internal node buffers* store the intermediate results assembled from sub-tree buffers. Each internal node is associated with one operator in the plan, along with a collection of predicates. ZStream assumes that primitive events from data sources continuously stream into leaf buffers in time order. If disorder is a problem, a reordering operator may be placed just after the leaf buffer. ZStream uses a *batch-iterator model* (see Section 4.3) that collects batches of primitive events before processing at internal nodes; batches are processed to form more complete composite events that are eventually output at the root. Figure 3 shows a tree plan for Query 1. This is a *left-deep plan*, because $T1$ and $T2$ are first combined, and their outputs are matched with $T3$. A right-deep plan, where $T2$ and $T3$ are first combined, and then matched with $T1$, is also possible.

Single-class predicates (over just one event class) can be pushed down to the leaf buffers, preventing irrelevant events from being placed into leaf buffers. For example, "$T2.name$ = *'Google'*" can be pushed down to the front of the $T2$ buffer such that only the events with their names equal to *'Google'* are placed into the buffer. More complicated constraints specified via multi-class predicates are associated with internal nodes. For example, the first sequential relation in Query 1 has a multi-class predicate "$T1.price > (1 + x\%) * T2.price$" associated with it; hence this predicate is attached to the $SEQ1$ node. Events coming from the associated sub-trees are passed into the internal nodes for combination, and multi-class predicates are applied during this assembly procedure. Notice that equality multi-class predicates can be further pushed to the leaf buffers by building hash partitions on the equality attributes. For instance, the equality predicate "$T1.name = T3.name$" of Query 1 can be expressed as hashing on "$T1.name$" and performing $name$ lookups on the $SEQ2$ node. Other operators can be represented in the tree model similarly; we describe ZStream's operator implementations in Section 4.4.

4.2 Buffer Structure

For each node of the tree plan, ZStream has a buffer to temporally store incoming events (for leaf nodes) or intermediate results (for internal nodes). Each buffer contains a number of records, each of which has three parts: a vector of event pointers, a start time and an end time. If the buffer is a leaf buffer, the vector just contains one pointer that points to the incoming primitive event, and both the start time and end time are the timestamp of that primitive event. If the buffer contains intermediate results, each entry in the event vector points to a component primitive event of the assembled composite event. The start time and the end time are the timestamps of the earliest and the latest primitive event comprising this composite event.

One important feature of the buffer is that its records are stored sorted in end time order. Since primitive events are inserted into their leaf buffers in time order, records in leaf buffers are automatically sorted by end time. For each internal non-leaf buffer, the node's operator is designed in such a way that it extracts records in its child buffers in end time order and also generates intermediate results in end time order.

This buffer design facilitates time range matching for sequential operations. In addition, The explicit separation of the start and end time facilitates time comparison between two buffers A and B in either direction (i.e., A to B and B to A), so that:

1. It is possible for ZStream to efficiently locate tuples in leaf buffers, so that ZStream can flexibly choose the order in which it evaluates the leaf buffers;
2. It is possible for ZStream to efficiently locate tuples also in internal node buffers matching a specific time range so that materialization can be used; and
3. ZStream can support conjunction by efficiently joining events from either buffer with the other buffer.

In addition, by keeping records in the end-time order in buffers, ZStream does not perform any unnecessary time comparisons. Since each operator evaluates its input in end-time order, events outside the time range can be discarded once the first out-of-time event is found. Other buffer designs, based, for example, on pointers from later buffers to matching time ranges in earlier buffers (such as the *RIP* used in [15]), make flexible reordering difficult for sequential patterns.

4.3 Batch-Iterator Model

For sequential patterns an output can only be possibly generated when an instance of the final event class occurs (e.g., B in the pattern $A; B$). If events are combined whenever new events arrive, previously combined intermediate results may stay in memory for a long time, waiting for an event from the final event class to arrive. If no events arrive from the final event class for a long time, these intermediate results are very likely to be discarded without even being used. Hence, ZStream accumulates events in leaf buffers in

Idle rounds, and performs evaluation to populate internal buffers and produce outputs in *Assembly rounds* only after there is at least one instance f in the final event class's buffer.

During assembly, the system computes an *earliest allowed timestamp (EAT)* based on f's timestamp. More specifically, the EAT in each assembly round is calculated by subtracting the time window constraint from the earliest end-timestamp of the events in the final event class's buffer. Any event with start time earlier than the EAT cannot possibly satisfy the pattern and can be discarded without further processing.

Specifically, the batch-iterator model consists of the following steps:

1. A batch of primitive events is read into leaf buffers with the predefined batch size;
2. If there is no event instance in the final event's leaf buffer, go back to step 1; otherwise, go to step 3;
3. Calculate the EAT and pass it down to each buffer from the root;
4. Assemble events from leaves to root, storing the intermediate results in their corresponding node buffers, and removing out-of-date records, according to the implementation of each operator (see Section 4.4).

Notice that steps 1-2 belong to idle rounds, which are used to accumulate enough primitive events; and steps 3-4 belong to assembly rounds, which are used to perform composition work.

For in-memory pattern matching, memory consumption is considered an important issue. By taking the advantage of the batch-iterator model and evaluating the time window constraints as early as possible (EAT pushed to leaf buffers), ZStream can bound the memory usage efficiently. We will show this by reporting the peak memory usage in Section 6.

4.4 Operator Evaluation

In this section, we describe the algorithms for operator evaluation.

4.4.1 Sequence

Each sequence operator has two operands, representing the first and second event class in the sequential pattern. The algorithm used in each assembly round for sequence evaluation is given as follows:

```
   Input: right and left child buffers $RBuf$ and $LBuf$, $EAT$
   Output: result buffer $Buf$
1  foreach $Rr \in RBuf$ do
2      if $Rr.start\text{-}ts < EAT$ then remove $Rr$; continue;
3      for $Lr = LBuf[0]; Lr.end\text{-}ts < Rr.start\text{-}ts; Lr{+}{+}$ do
4          if $Lr.start\text{-}ts < EAT$ then remove $Lr$; continue;
5          if $Lr$ and $Rr$ satisfy value constraints then
6              Combine $Lr$ and $Rr$ and insert into $Buf$
7  Clear RBuf
```

Algorithm 1: Sequence Evaluation

To make sure the intermediate results are generated in end-time order, the right buffer is used in the outer loop, and the left buffer is in the inner one. In the algorithm, step 2 and 4 are used to incorporate the EAT constraint into the sequence evaluation. We assume that all events arriving in earlier rounds have smaller timestamps than those arriving in later rounds. This means that for a sequence node, after the assembly work is done, records in the node's right child buffer will not be used any more because all possible events that can be combined with them have already passed through the left child buffer and the combined results have been materialized. Hence, step 7 is applied to clear the sequence node's right child buffer.

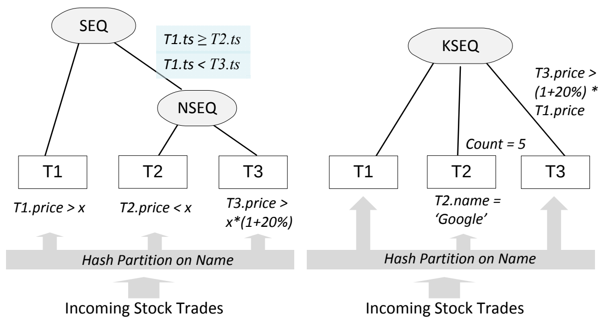

Figure 4: A right-deep tree plan for Query 2 (left); and a query plan for Query 3 (right).

4.4.2 Negation

Negation represents events that have not yet occurred. It must be combined with other operators, such as sequence and conjunction. ZStream does not allow negation to appear by itself as there is no way to represent the absence of an occurrence. Though we could add a special "did not occur" event, it is unclear what timestamp to assign such events or how frequently to output them. Semantically, it also makes little sense to combine negation with disjunction (i.e., $A|!B$) or Kleene closure (i.e., $!A^*$).

Negation is more complicated than other operators because it is difficult to represent and assemble all possible non-occurrence events. If events specified in the negation clause occur, they can cause previously assembled composite events to become invalid. One way to solve this problem is to add a negation filter on top of the plan to rule out composite events that have events specified in the negation part occurred, as has been done in previous work [15]. An obvious problem with the last-filter-step solution is that it generates a number of intermediate results, many of which may be filtered out eventually. A better solution is to push negation down to avoid unnecessary intermediate results, which is what we will discuss in this section.

Negation evaluation is performed using the $NSEQ$ operator. The key insight of the NSEQ operator is that it can find time ranges when non-negation events can produce valid results. Consider a simplified pattern "$A; !B; C$ WITHIN tw" where tw is a time constraint. For an event instance c of C, suppose there exists a negation event instance b of B such that

1. Any event instance of A that occurred before b does not match with c (because it is negated by b).
2. And conversely, any instance of A that occurred after b and before c definitely matches with c (because not negated by b).

In this case, we say b *negates* c. Thus, the NSEQ operator searches for a negation event that negates each non-negation event instance (each instance of C in the above example.)

Figure 4(left) illustrates a right-deep plan for Query 2. *Hash Partitioning* is performed on the incoming stock stream to apply the equality predicates on *stock.name*.

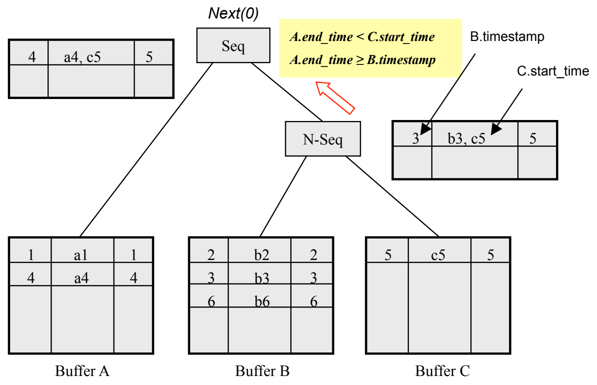

Figure 5: An example of NSEQ evaluation for the pattern "$A; !B; C$ WITHIN tw"

NSEQ is applied to find an event instance $t2$ of $T2$ that negates each event instance $t3$ of $T3$. The output from $NSEQ$ is a combination of such $t2$ and $t3$. In addition, the extra time constraints $T1.ts \geq T2.ts$ and $T1.ts < T3.ts$ are added to the SEQ operator which takes the output of $NSEQ$ as an input. These time constraints determine the range of event instances of $T1$ that can be combined with the events output from $NSEQ$ directly. Thus, this plan can reduce the size of unnecessary intermediate results dramatically.

We now discuss the evaluation of NSEQ and we assume that negation events are primitive. Figure 5 illustrates the execution of the simplified pattern "$A; !B; C$ WITHIN tw". In such cases, the event that negates c of C is the latest negation event instance b that causes c to become invalid. For instance, $b3$ negates $c5$, which is recorded as "$b3, c5$" in the $NSEQ$ buffer. The time predicates $A.end\text{-}ts < C.start\text{-}ts$ and $A.end\text{-}ts \geq B.ts$ are pushed into the SEQ operator. These indicate that, due to negation, only event instances of A in time range $[3, 5)$ should be considered (i.e. $a4$). Finally, the composite result "$a4, c5$" is returned.

The algorithm to evaluate NSEQ when its left child is a negation event class is shown in Algorithm 2. NSEQ works much the same as SEQ except that the left buffer is looped through from the end to the beginning; and only negation events that negate events from the right buffer are combined and inserted into the result buffer (steps 7–9). When no negation event can be found to negate the instance from the right buffer, (NULL, Rr) is inserted into the result buffer instead (steps 5 and 10). The algorithm to evaluate NSEQ where the right child is a negation class ("$B; !C$") can be constructed similarly. In this case, the event that negates each b should be the first event from C that arrives after b and also passes all predicates.

Notice that Algorithm 2 only works for the case where negation event class's multi-class predicates all apply to just one of the non-negation classes. However, if the negation event class has multi-class predicates over more than one non-negation event class, locating the direct matching range is more tricky. Some of the valid composite components may have been filtered out in NSEQ because NSEQ only contains the predicate information of two event classes. To solve this problem, more sophisticated extra predicates need to be added and more composed components need to be saved.

```
   Input: left negation child buffer $LBuf$, right child buffer
          $RBuf$, $EAT$
   Output: result buffer $Buf$
 1 foreach $Rr \in RBuf$ do
 2     if $Rr.start\text{-}ts < EAT$ then remove $Rr$; continue;
 3     for $i = LBuf.length - 1; i \geq 0; i\text{--}$ do
 4         if $LBuf[i].start\text{-}ts < EAT$ then
 5             insert($NULL$,$Rr$) to Buf;
 6             clear LBuf from position $i$ to 0; break;
 7         if $LBuf[i].end\text{-}ts < Rr.start\text{-}ts$
 8         AND $LBuf[i], Rr$ satisfy value constraints then
 9             insert($LBuf[i]$,$Rr$) to $Buf$;break;
10     if no $Lr$ was found then insert ($NULL$, $Rr$) to $Buf$
11 Clear RBuf
```
Algorithm 2: NSEQ Evaluation

This may cancel out the benefits of NSEQ. Hence, ZStream applies a negation operator at the top of the query plan rather than using NSEQ in such cases.

4.4.3 Conjunction

Conjunction is similar to sequence except that it does not distinguish between the orders of its two operands, as it assembles events in both directions. The evaluation algorithm is shown in Algorithm 3. It is designed to work like a sort-merge join. It maintains a cursor on both input buffers (Lr and Rr), initially pointing to the oldest not-yet-matched event in each buffer. In each step of the algorithm, it chooses the cursor pointing at the earlier event e, (lines 3– 7), and combines e with all earlier events in the other cursor's buffer (lines 8–9). This algorithm produces events in end-time order, since it processes the earliest event at each time step.

```
   Input: right and left child buffers $RBuf$ and $LBuf$, $EAT$
   Output: result buffer $Buf$
 1 Set $Lr = LBuf.initial$ and $Rr = RBuf.initial$
 2 while $Lr != LBuf[end]$ OR $Rr != RBuf[end]$ do
 3     if $Lr.start\text{-}ts < EAT$ then remove $Lr$;$Lr$++;continue;
 4     if $Rr.start\text{-}ts < EAT$ then remove $Rr$;$Rr$++;continue;
 5     if $Lr.end\text{-}ts > Rr.end\text{-}ts$ then
 6         $Pr = Rr; Rr{+}{+}; Cr = Lr; CBuf = LBuf$
 7     else $Pr = Lr; Lr{+}{+}; Cr = Rr; CBuf = RBuf$
 8     for $Br = CBuf[0]; Br != Cr; Br{+}{+}$ do
 9         if $Br$, $Pr$ satisfy value and time constraints then
10             $Br$ and $Pr$ are combined and inserted into $Buf$
11 Set $LBuf.initial = Lr$, $RBuf.initial = Rr$, for next round
```
Algorithm 3: Conjunction Evaluation

4.4.4 Disjunction

Disjunction simply outputs union of its inputs, so its evaluation is very straightforward: either input can be directly inserted into the operator's output buffer if it meets both time and value constraints. Specifically, the output of disjunction is generated by merging events in the left buffer and the right buffer according to their end time. ZStream does not materialize results from the disjunction operator because most of the time, they will simply be a copy of the inputs.

4.4.5 Kleene Closure

Figure 4(right) shows a tree plan for Query 3. The input stock stream is first hash partitioned on *stock.name*, and the *Google* buffer can be shared with all the partitions. Kleene

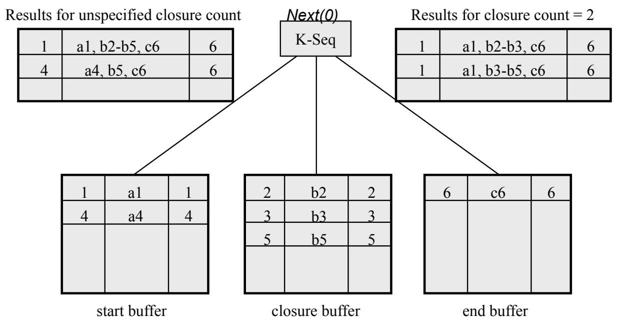

Figure 6: Example KSEQ evaluation for the patterns "$A; B^2; C$" and "$A; B^*; C$"

closure is represented as a trinary *KSEQ* node in the tree plan. *count* = 5 on the top of the $T2$ buffer indicates that it will group 5 successive $T2$ events together. If the number is not specified, the symbol '+' or '$*$' can be used instead to group the maximal number of possible successive $T2$ events together. The KSEQ operator is trinary because a closure pattern (in general) needs an event class to start and end the closure. The first operand acts as a start event class, the last operand acts as an end event class and the Kleene closure detection is performed on the middle operand. The start and end classes may be omitted if the Kleene closure appears at the start or end of a pattern.

The KSEQ operator looks for closure matches in the middle buffer. If the closure count is not specified, the maximal number of the middle buffer events between the start point and the end point is found; only one result is generated for this pair of start and end points. If the closure count is specified, a window is applied on the middle buffer with its window size equal to the count. In each time step, the window is moved forward one step; one or more results are generated for each start-end pair in this case. To ensure intermediate results are arranged in the end-time order, the end buffer is put in the outer loop of the evaluation algorithm.

Figure 6 illustrates an example of the evaluation of KSEQ. The buffer on the upper left shows the results when the closure count is not specified; the one on the upper right shows the results when the closure count is 2. In both cases, $c6$ from the end buffer is first picked up to fix the end point; then $a1$ from the start buffer is chosen to fix the start point. When the closure count is not specified, all the events in the closure buffer between the end time of $a1$ and the start time of $c6$ are grouped together. When the closure count is set to 2, after $a1$ and $c6$ are chosen, two groups of "$b2, b3$" and "$b3, b5$" are formed to match with them. The algorithm to evaluate KSEQ in each assembly round is shown as Algorithm 4.

5. COST MODEL AND OPTIMIZATIONS

This section presents the cost model and optimization techniques used in ZStream. Based on the cost model, ZStream can efficiently search for the optimal execution plan for each sequential query. We also show that our evaluation model can easily and seamlessly adapt to a more optimal plan on the fly, and present optimizations for using hashing to evaluate equality predicates.

```
Input: start buffer $SBuf$, middle buffer $MBuf$, end buffer
       $EBuf$, closure count $cc$, $EAT$
Output: result buffer $Buf$
foreach $Er \in EBuf$ do
    if $Er.start\text{-}ts < EAT$ then remove $Er$; continue;
    foreach $Sr \in SBuf$ and $Sr.end\text{-}ts < Er.start\text{-}ts$ do
        if $Sr.start\text{-}ts < EAT$ then remove $Sr$; continue;
        foreach $Mr \in MBuf$ do
            if $Mr.start\text{-}ts < EAT$
            OR $Mr.start\text{-}st <= Sr.end\text{-}ts$ then
                remove $Mr$; continue;
            if $cc$ is not specified then
                Group $Mr$s where $Mr.start\text{-}ts > Sr.end\text{-}ts$
                and $Mr.end\text{-}ts < Er.start\text{-}ts$ and $Mr$
                satisfies the value constraints
                Insert the composite event into $Buf$
            else if $cc$ is specified then
                foreach $Mr$, $Mr.start\text{-}ts > Sr.end\text{-}ts$ do
                    Group from $Mr$ where $Gr \in Group$
                    having $Gr.end\text{-}ts < Er.start\text{-}ts$ and
                    satisfies value constraints
                    until $Group.length == cc$ or
                    $Gr.end\text{-}ts \geq Er.start\text{-}ts$
                    Insert the composite event into Buf
                    if $Gr.end\text{-}ts \geq Er.start\text{-}ts$ then break
```
Algorithm 4: KSEQ Evaluation

5.1 Cost Model

In traditional databases, the estimated cost of a query plan consists of I/O and CPU costs. In ZStream, I/O cost is not considered because all primitive events are memory resident. ZStream computes CPU cost for each operator from three terms: the cost to access the input data, the cost of predicate evaluation and the cost to generate the output data. These costs are measured as the number of input events accessed and the number of output events combined. Formally, the cost C is:

$$C = C_i + (nk)C_i + pC_o \tag{1}$$

Here, the cost consists of three parts: the cost of accessing the input data C_i, the cost to generate the output data pC_o and the cost of predicate evaluation $(nk)C_i$, where n is the number of multi-class predicates the operator has (which cannot be pushed down), and k and p are weights.

C_i and C_o stand for the cost of accessing input data and assembling output results respectively. Since both of them are measured based on the number of events touched, the weight p is set to 1 by default, which we have experimentally determined to work well. $(nk)C_i$ stands for the cost of predicate evaluation. Since predicate evaluation is performed during the process of accessing the input data, its cost is proportional to C_i. Based on our experiments, k is estimated to be 0.25 in ZStream.

Table 1 shows the terminology that we use in the rest of this section. R_E determines the number of events per unit time. Hence $R_E * TW_p * P_E$ can be used as an estimate of $CARD_E$ (all instances of E that are active within the time period TW_p). Consider the plan shown in Figure 3; the $CARD$ of $T2$ is $R_{STOCK} * P_{T2} * (10sec)$, where P_{T2} is the selectivity for the predicate "$T2.name$ = '*Google*'".

Sequential operators such as SEQ, KSEQ, NSEQ have implicit time constraints. $Pt_{E1,E2}$ is used to measure the selectivity of such predicates. For example, the pattern "$E1; E2$" implicitly includes a time predicate "$E1.end\text{-}ts < E2.start\text{-}ts$", indicating that only event instances $e1$ of $E1$

Table 1: Terminology Used in Cost Estimation

Term	Definition
R_E	Rate of primitive events from the event class or partition E. This is the cardinality of E per unit time.
TW_p	Time window specified in a given query pattern p
P_E	Selectivity of all single-class predicates for event class or partition E. This is the product of selectivity of each single-class predicate of E.
$CARD_E$	Cardinality of the events from event class E that are active in time window TW_p. This can be estimated as $R_E * TW_p * P_E$
$Pt_{E1,E2}$	Selectivity of the implicit time predicate between event class or partition $E1$ and $E2$, with $E1.end\text{-}ts < E2.start\text{-}ts$. The default value is set to 1/2.
$P_{E1,E2}$	Selectivity of multi-class predicates between event class or partition $E1$ and $E2$. This is the product of selectivity of the multi-class predicates between $E1$ and $E2$. If $E1$ and $E2$ do not have predicates, it is set to 1.
C_{iO}	The cost of the operator O to access its input data.
C_{oO}	The cost of the operator O to access its output data.
$CARD_O$	Cardinality of output of operator O. It is used to measure the output cost of the operator, i.e. C_{oO}
C_O	Total estimated cost (in terms of the number of events touched) of the operator O, it is the sum of C_{iO} and $CARD_O(C_{oO})$.

that occur before instances $e2$ of $E2$ can be combined with $e2$. $Pt_{E1,E2}$ does not apply to the cost formula for conjunction and disjunction as they are not sequential. $P_{E1,E2}$ is similar to $Pt_{E1,E2}$ except that it includes the selectivity of all multi-class predicates between $E1$ and $E2$.

Table 2 summarizes the input cost formulas (C_{iO}) and output cost formulas (C_{oO}) for each individual operator. The input cost C_{iO} is expressed in terms of the number of input events that are compared and/or combined; and the output cost C_{oO} is measured as the number of composite events generated by the operator (i.e., $CARD_O$). The total cost of an individual operator is the sum of its input cost, predicate cost and output cost as indicated in Formula 1.

The implicit time selectivity $Pt_{A,B}$ is attached to the sequence operator's input cost formula because the buffer structure automatically filters out all event instances a of A with $a.end\text{-}ts \geq b.start\text{-}ts$ for each event instance b of B. The evaluation of conjunction and disjunction is independent of the order of their inputs; hence time predicates do not apply to their cost formulas. For disjunction, multi-class predicates are not included because an event on either of the two inputs can result in an output.

The cost for Kleene closure is more complicated. If the closure count *cnt* is not specified, exactly one group of the maximal number of closure events is output for each start-end pair. The number of accessed event instances N from the middle input B can be estimated as the number of events from B that match each start-end pair. So $N = CARD_B * Pt_{A,B} * Pt_{B,C}$, where A and C represent the start and end event class. If the closure count *cnt* is specified, then for any start-end pair, each event instance from B that occurs in between this pair will be output *cnt* times on average. Hence $N = CARD_B * Pt_{A,B} * Pt_{B,C} * cnt$.

ZStream has two ways to evaluate negation. One is to put a negation filter on top of the whole plan to rule out negated events. The other is to use an NSEQ operator to push the negation into the plan. The cost of the first method ($NEG(SEQ(A,C),B)$) includes two parts: the cost of the SEQ operator and that of NEG. The cost of SEQ can be estimated as above. The input cost for NEG is $CARD_{SEQ}$. It is not related to $CARD_B$ because the composite results from SEQ can be thrown out once an instance b of B between A and C is found. ZStream can find such a b by finding the event that negates each c of C, and hence it does not need to scan B. The cost of the second approach ($Seq(A, NSEQ(B,C))$) also contains two parts: the cost to evaluate the NSEQ and the cost to evaluate the SEQ operator. The input cost of the NSEQ is $CARD_C$ and not related to $CARD_B$ because ZStream can find each c's negating event (which is just the latest event in B before c) directly, without searching the entire B buffer.

The cost formulas shown in Table 2 assume that the operands of each operator are primitive event classes. They can be easily generalized to the cases where operands themselves are operators by substituting the cardinality of primitive event classes with the cardinality of operators. Then, the cost of an entire tree plan can simply be estimated by adding up the costs of all the operators in the tree.

5.2 Optimal Query Plan

Our goal is to find the best physical query plan for a given logical query pattern. To do this, we define the notion of an equivalent query plan — that is, a query plan p' with a different ordering or collection of operators that produces the same output as some initial plan p. In particular, we study three types of equivalence: rule-based transformations, hashing for equality multi-class predicates and operator reordering.

5.2.1 *Rule-Based Transformations*

As in relational systems, there are a number of equivalent expressions for a given pattern. For example, the following two expressions are semantically identical:

1. $Expression1$: "$A; (!B\&!C); D$"
2. $Expression2$: "$A; !(B|C); D$"

Their expression complexity and evaluation cost, however, are substantially different from each other. ZStream supports a large number of such algebraic rewrites that are similar to those used in most database systems; we omit a complete list due to space constraints. Based on these equivalence rules, we can generate an exponential number of equivalent expressions for any given pattern. Obviously, it is not practical to choose the optimal expression by searching this equivalent expression space exhaustively. Instead, we narrow down the transition space by always trying to simplify the pattern expression; a transition is taken only when the target expression:

1. has a smaller number of operators or,
2. the expression has the same number of operators, but contains lower cost operators.

Plans with fewer operators will usually include fewer event classes, and thus are more likely to result in fewer intermediate composed events being generated and less overall work. If the alternative plan has the same number of operators, but includes lower cost operators, it is also preferable. The cost of operators is as shown in Table 2, which indicates that $C_{DIS} < C_{SEQ} < C_{CON}$ (NSEQ and KSEQ are not substitutable for other operators).

Returning to the two expressions given at the beginning of this section, the optimizer will replace $Expression1$ with $Expression2$ because $Expression2$ has fewer operators and the cost of Disjunction is smaller than that of Conjunction.

5.2.2 *Hashing for Equality Predicates*

Table 2: Input and Output Cost formula for Individual Operators

Operator	Description	Input Cost C_i	Output Cost C_o
Sequence $(A;B)$	A and B are two input event classes or partitions. The cost is expressed as the number of input combinations tried. $Pt_{A,B}$ captures the fact that the sequence operator does not try to assemble any a of A with b of B where b occurs before a.	$CARD_A * CARD_B * Pt_{A,B}$	$CARD_A * CARD_B * Pt_{A,B} * P_{A,B}$
Conjunction $(A\&B)$	A and B are two input event classes or partitions. Unlike Sequence, Conjunction can combine event a of A with any b of B within the time window constraint.	$CARD_A * CARD_B$	$CARD_A * CARD_B * P_{A,B}$
Disjunction $(A\|B)$	A and B are two input event classes or partitions. Since Disjunction simply merges its inputs, its cost is just the cost of fetching each input event.	$CARD_A + CARD_B$	$CARD_A + CARD_B$
Kleene Closure $(A;B^{cnt};C)$	A and C are the start and the end event class, respectively. B is the closure events class. If A or C is missing, the parameters related to A or C are set to be 1. $N = CARD_B * Pt_{A,B} * Pt_{B,C} * cnt$, where cnt is the closure number; $N = CARD_B * Pt_{A,B} * Pt_{B,C}$, if cnt is missing; $N = 1$, if event class B is missing	$CARD_A * CARD_C * Pt_{A,C} * N$	$CARD_A * CARD_C * Pt_{A,C} * N * P_{A,C} * P_{A,B} * P_{B,C}$
Negation $(A;!B;C)$ (top)	Negation on top, expressed as: $NEG(SEQ(A,C),!B)$	$C_{iSeq} + CARD_{SEQ}$	$CARD_{SEQ} + CARD_{SEQ} * (1 - Pt_{A,B} * Pt_{B,C}) * Pt_{A,C}$
Negation $(A;!B;C)$ (pushed down)	Negation pushed down, expressed as $SEQ(A,NSEQ(B,C))$	$CARD_C + CARD_A * CARD_C * Pt_{A,C}$	$CARD_C + CARD_A * CARD_C * Pt_{A,C} * (1 - Pt_{A,C} * Pt_{B,C})$

As a second heuristic optimization step, ZStream replaces equality predicates with hash-based lookups whenever possible. Hashing is able to reduce search costs for equality predicates between different event classes; otherwise, equality multi-class predicates can be attached to the associated operators as other predicates.

As shown in Figure 3 (a tree plan for Query 1), the incoming stock stream is first hash partitioned on "*name*" as $T1$. Internally, $T1$ is represented as a hash table, which is maintained when $T1$ is combined with $T2$ in the $SEQ1$ node. When the equality predicate $T1.name = T3.name$ is applied during $SEQ2$, this lookup can be applied directly as a probe for $T3.name$ in the hash table built on $T1$.

More formally, suppose $P(A,B,f)$ denotes a predicate $A.f = B.f$. If A and B are sequentially combined in the order $A;B$, the hash table is built on $A.f$ (because B is used in the outer loop in Algorithm 1). If A and B are conjunctively connected, hash tables are built on both $A.f$ and $B.f$ (because both A and B may have chances to be in the outer loop).

Hash construction and evaluation can be easily extended to the case where there are multiple equality predicates. Suppose $P_1(A_1,B_1,f_1)$ and $P_2(A_2,B_2,f_2)$ are two equality predicates for a sequential pattern where A_i is before B_i $(i = 1,2)$. Then,

1. If $A_1 \neq A_2$, build hash tables on both $A_1.f_1$ and $A_2.f_2$
2. If $A_1 = A_2$ and $f_1 = f_2$, build a hash table on $A_1.f_1$
3. Otherwise $A_1 = A_2$ and $f_1 \neq f_2$; build the primary hash table on $A_1.f_1$ and a secondary hash table on $A_1.f_2$.

5.2.3 Optimal Reordering

Once a query expression has been simplified using algebraic rewrites and hashing has been applied on the equality attributes, ZStream is left with a logical plan. A given logical plan has a number of different physical trees (e.g., left-deep or right-deep) that can be used to evaluate the query. In this section, we describe an algorithm to search for the optimal physical tree for the sequential pattern. We first observe that the problem of finding the optimal order has an optimal substructure, suggesting it is amenable to dynamic programming, as in Selinger [14].

THEOREM 5.1. *For a given query, if the tree plan T is optimal, then all the sub trees T_i of T must be optimal for their corresponding sub patterns as well.*

PROOF. We prove this by contradiction. Suppose the theorem is not true; then it should be possible to find a sub tree plan T'_i with lower cost than T_i, but with the same output cardinality. Using T'_i as a substitute for T_i, we would then obtain a better tree plan T' with lower total cost for the pattern, which contradicts the assumption that T is optimal. $\square$

Based on the optimal substructure observation in Theorem 5.1, we can search for an optimal tree plan by combining increasingly larger optimal sub plans together until we have found an optimal plan for the whole pattern. The algorithm to search for the optimal operator order is shown as Algorithm 5. The algorithm begins by calculating the optimal sub plans from the event sets of size 2. In this case, there is only one operator connecting the event classes; hence its operator order is automatically optimal. In the outermost loop (line 2), the algorithm increases the event set size by 1 each time. The second loop (line 3) goes over all possible event sets of the current event set size. The third loop (line 4) records the minimal cost plan found so far by searching for all possible optimal sub-trees. The root of each optimal sub-tree chosen for the current sub-pattern is recorded in the *ROOT* matrix. In the end, the optimal tree plan can be reconstructed by walking in reverse from the root of each selected optimal sub-tree. The two function calls: *calc_inputcost*() and *calc_CARD*() are used to estimate the input cost and output cost of an operator according to Table 2.

Algorithm 5 is equivalent to the problem of enumerating the power set of the operators, and hence generates $O(n^2)$ subsets in total. For each subset, it performs one pass to find the optimal root position. Hence the total time complexity is $O(n^3)$. Compared to Selinger [14], Algorithm 5 also takes the bushy plans into consideration. In practice, this algorithm is very efficient, requiring less than 10 *ms* to search for an optimal plan with pattern length 20.

Figure 7 illustrates an example of an optimal plan generated when the pattern length is 4. During the final round, where the only event set to consider is the pattern $(1,2,3,4)$, the algorithm tries all possible combination of sub-lists: 1 with $(2,3,4)$; $(1,2)$ with $(3,4)$; $(1,2,3)$ with 4. The root of each optimal sub tree is marked. The final optimal (bushy) plan selected for this pattern is shown on the right.

```
**Input**: number of event classes $N$, statistics info
**Output**: buffer $ROOT$ recording roots of optimal sub trees
1 Initialize two dimensional matrices $Min$, $Root$, and $CARD$
2 for $s = 2; s \le n; s{+}{+}$ do  // $s$ is sub tree size
3     for $i = 1; i \le n - s + 1; i{+}{+}$ do  // $i$ is sub tree index
4         for $r = i + 1; r < i + s; r + +$ do  // $r$ is root pos
5             $opc$ = cacl_inputcost($CARD[r - i][i], CARD[s - r + i][r], r$);
6             $cost = Min[r - i][i] + Min[s - r + i][r] + opc$;
7             if $Min[s][i] > Cost$ then
8                 $Min[s][i] = Cost; ROOT[s][i] = r$;
9                 $CARD[s][i]$ = calc_CARD($opc, r$);
```

Algorithm 5: Searching for Optimal Operator Order

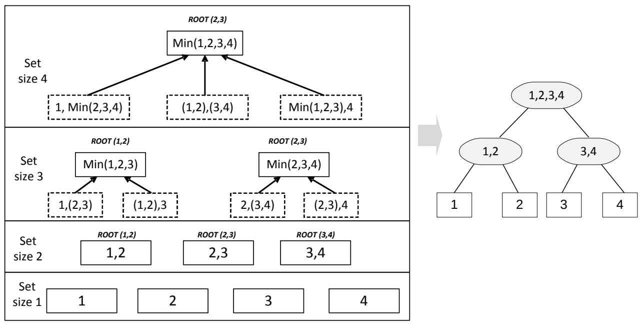

Figure 7: Illustration of searching for the optimal operator order when pattern length = 4

5.3 Plan Adaptation

As a result of potentially high variability in input stream rates and selectivities, an initially optimal plan may no longer be optimal after running for some time. To recompute the plan on the fly, we maintain a running estimate of the statistics in Table 1, using sampling operators attached to the leaf buffers. In our implementation, we use simple windowed averages to maintain the rates of each input stream and the selectivity of each predicate. When any statistic used in a plan varies by more than some error threshold t, we re-run Algorithm 5 and install the new plan if the performance improvement predicted by the cost model is greater than a performance threshold c.

The use of the batch-iterator model simplifies the task of switching plans on the fly. Notice that in each assembly round, newly arriving batches can rebuild all the previous intermediate results they need from the leaf buffers. To make this work, we modified our algorithms to not discard tuples from leaf buffers after tuples have been consumed (e.g., we do not perform Line 7 of Algorithm 1 for leaf buffers). Switching plans will not produce any duplicate results as each round is independent of the previous round. Hence, ZStream can change a running query plan on assembly round i using the following two steps:

1. Discard all the intermediate results for the old plan after finishing assembly round i;
2. Rebuild all the intermediate results for the new plan in assembly round $i + 1$ as if it were the first round.

Even though this plan adaptation strategy is simple, it can outperform the static plan dramatically, as we show in Section 6. More sophisticated adaptive strategies may try to reuse some of the intermediate results already stored and minimize the recalculation [11] or incorporate parallelism [16] when plans are changed. We leave these problems for future work.

6. PERFORMANCE EVALUATION

In this section, we describe a number of experiments we have run to evaluate the performance of ZStream. Our primary objectives were to understand to what extent the reordering optimizations described in the previous sections affect overall query performance and by how much ZStream can outperform a previously-proposed NFA-based approach [15]. We also look at the performance of negation push down and the efficiency of plan adaptation. Finally, we test ZStream on some real world web log data in Section 6.5.

ZStream is implemented in C++ using STL:*list* to maintain the buffer structure. We separately implemented the NFA-based approach described in [15]. It is also C++ based, and uses STL:*deque* to support *RIP* pointers on its stack structure. In our experiments, STL:*deque* (for random lookups) proved to be about 1.5 times faster than STL:*list*. Note that materialization is not supported in our NFA implementation because *RIP* implementation has difficulty to support materialization for multi-class range predicates (e.g., $A.price > B.price$ for sequential pattern $A; B$)

All experiments were run on a dual core CPU 3.2 GHz Intel Pentium 4 with one core turned off and 2 GB RAM. We ran ZStream on a pre-recorded data file; data was pulled into the system at the maximum rate the system could accept. System performance was measured by the rate at which input data was processed, i.e.:

$$rate = \frac{|Input|}{t_{elapsed}}$$

where $|Input|$ is the size of the input and $t_{elapsed}$ is the total elapsed processing time, not counting time to deliver the output. The input data is stock trade data with the schema described in Section 3.2. We generate synthetic stock events so that event rates and the selectivity of multi-class predicates could be controlled. All experiments are the average of 30 runs. Peak memory usage are also reported for some experiments.

6.1 Parameters Affecting Costs

In this section, we experiment on various factors that affect the costs of query plans, showing that the cost model proposed in Section 5 accurately reflects the system performance.

6.1.1 *Multi-Class Predicate Selectivity*

We ran experiments on Query 4, a sequential pattern with a single predicate on the first two event classes, using a left-deep plan, a right-deep plan, and the NFA based approach. Here, incoming ticks have a uniform distribution over stock names, meaning relative event rates are 1 : 1 : 1 (that is, one *Sun* quote arrives for each *IBM* quote, and one *Oracle* quote arrives for each *Sun* quote).

Query 4. Sequence Pattern "$IBM; Sun; Oracle$" with a predicate between IBM and Sun

```
PATTERN   $IBM; Sun; Oracle$
WHERE     $IBM.price > Sun.price$
WITHIN    200 units
```

Figure 8 shows the throughput of the two alternative plans (the left-deep and the right-deep plan) and the NFA approach for Query 4. The left-deep plan outperforms the right-deep plan because it evaluates the operator with the

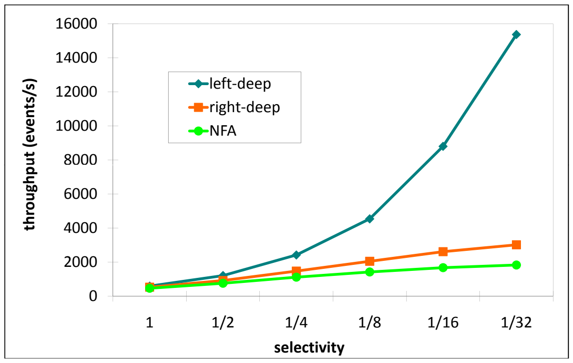

Figure 8: Throughputs of different plans for Query 4 with varying selectivity

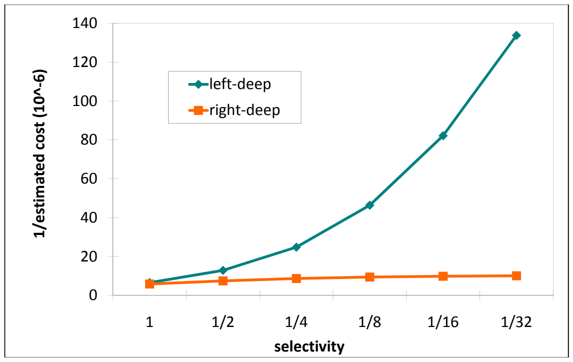

Figure 9: $1/estimated\ cost$ of different plans for Query 4 with varying selectivity

multi-class predicate between IBM and Sun first, such that it generates fewer intermediate results. The lower the selectivity, the fewer intermediate results the left-deep plan produces. Hence, the gap between the performance of the two plans increases with decreasing selectivity. When the predicate is very selective (1/32 for instance), the left-deep plan outperforms the right-deep plan by as much as a factor of 5. We also note that the NFA-based approach has similar performance to the right-deep plan in Figure 8. This is because the NFA constructs composite events using a backwards search on a DAG (Directed Acyclic Graph) [15]. This results in the NFA evaluating expressions in a similar order to the right deep plan. Our results show similar results for varying selectivity in longer sequential patterns.

Figure 9 shows the estimates produced by our cost model for the left-deep plan and the right-deep plan of Query 4 with varying selectivities. This shows that our cost model can accurately predict the system behavior with varying selectivity.

6.1.2 Event Rates

In this section, we study how varying the relative event rates of different event classes affects the cost of query plans for different queries. The intuition is that query plans that combine event classes with lower event rates first will generate a smaller number of intermediate results. Hence such query plans have better performance. To exclude the effect of selectivity, we experiment on a simple sequential pattern (Query 5) without any predicates.

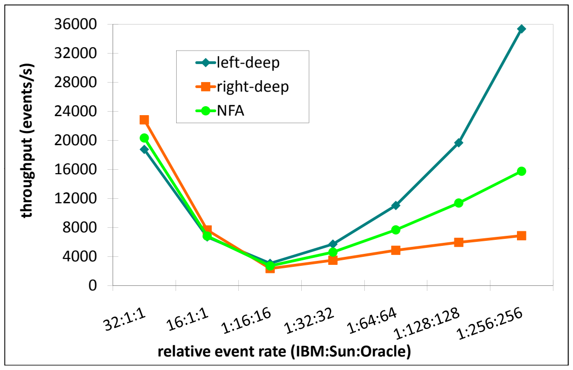

Figure 10: Throughputs of different plans for Query 5 with varying relative event rates

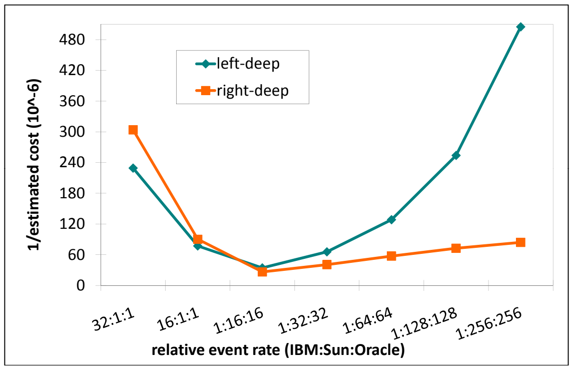

Figure 11: $1/estimated\ cost$ of different plans for Query 5 with varying relative event rates

Query 5. Sequence Pattern "$IBM; Sun; Oracle$"

PATTERN $IBM; Sun; Oracle$
WITHIN 200 *units*

Figure 10 shows the throughput for three plans (left-deep, right-deep and NFA-based) for Query 5 where we vary the relative event rate between IBM and the other two event classes. When IBM has a higher event rate, the right-deep plan performs the best, since IBM is joined later in this plan. The left-deep plan becomes best when IBM's event rate starts drops lower than the other two, since IBM is joined earlier in this plan. Figure 11 shows the estimated cost of the left-deep and the right-deep plan with varying relative rates, which turns out with the similar performance behavior as the real running throughput.

One additional observation from Figures 10 and 11 is that the performance gap between the best and worst performing plans on the right side of the figures is greater. These represent plans where there is a lower relative event rate for a single event class. Consider the case where event rate is $1 : 1 : 1$. In this case, the performance of all the plans is the same. Now, decreasing a single event class's rate by a factor of k is equivalent to increasing each of the other event classes' event rates by a factor of k. This results in a factor of k^{N-1} (where N is the total number of event classes) skew in the event distribution. In comparison, on the left side of the figure, increasing the rate of one stream only increases skew by a factor of k.

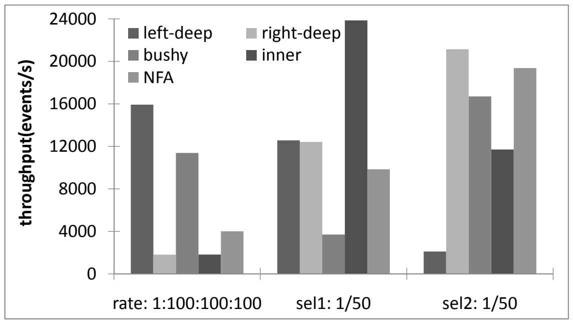

Figure 12: Throughput of different plans for Query 6

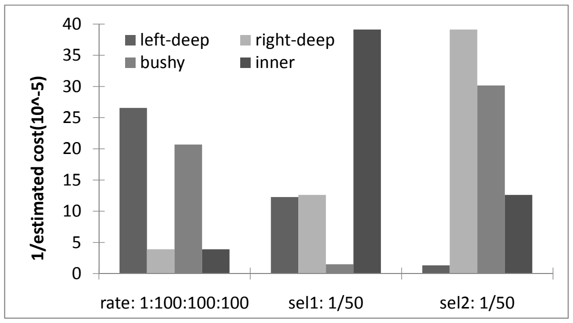

Figure 13: $1/estimatedcost$ of different plans for Query 6

6.2 Optimal Plans in More Complex Queries and Memory Usage

In this section, we show that the performance of different physical plans can vary dramatically when statistics change. The experiment is conducted using Query 6, running four different query plans and the NFA based approach.

Query 6. More Complex Query

PATTERN	$IBM; Sun; Oracle; Google$
WHERE	$Oracle.price > Sun.price$
AND	$Oracle.price > Google.price$
WITHIN	100 $units$

The plans are:

1. **Left-deep Plan**: [[[IBM; Sun]; Oracle]; Google]
2. **Right-deep Plan**: [IBM; [Sun; [Oracle; Google]]]
3. **Bushy Plan**: [[IBM; Sun]; [Oracle; Google]]
4. **Inner Plan**: [IBM; [[Sun; Oracle]; Google]]
5. **NFA**

We varied the event rate and selectivity of the different streams to show that the optimal plan changes quite dramatically. Figure 12 illustrates the throughput of the four plans with varying selectivity and relative event rate. For these experiments, we vary the proportion of inputs from different streams (from its default of $1:1:1:1$), as well as the selectivities of the two query predicates (from their default of 1). When the IBM event rate is low, as shown in the left most cluster of bars ($rate = 1:100:100:100$), the left-deep plan does best. The bushy plan also does well because it also uses IBM in the first (bottommost) operator. The right-deep plan, inner plan and NFA perform poorly because they combine with IBM in a later operator. In the second case where the first predicate (between Sun and $Oracle$) is very selective ($sel1 = 1/50$), the inner plan (which evaluates the first predicate first) does best, and it is almost two times faster than the other plans. The bushy plan in this case does extremely poorly because it defers the evaluation of the first predicate until the final processing step. The third case is good for the right-deep plan and the NFA-based approach because the predicate between the last two event classes is selective ($sel2 = 1/50$). As expected, the left-deep plan does poorly in this case. Figure 13 shows the estimates from our cost model (for all plans except NFA), showing that it predicts the real performance behavior well. Based on the cost model, our dynamic programming algorithm (Algorithm 5) should be able select the optimal plan efficiently.

Table 3 shows the peak memory consumption by the five plans for Query 6 in two cases: 1). when the IBM event rate is very low ($rate = 1:100:100:100$); and 2). when the predicate between Sun and $Oracle$ is very selective ($sel1 = 1/50$). As can be indicated form Table 3, the peak memory consumption is relatively stable among different plans (much more stable than the throughput of these different plans). In general, the memory consumption is independent of the input data size. It is the type of the query (pattern length, operator type and time window constraints) and data features (selectivity and event rate) that affect and bound the memory usage.

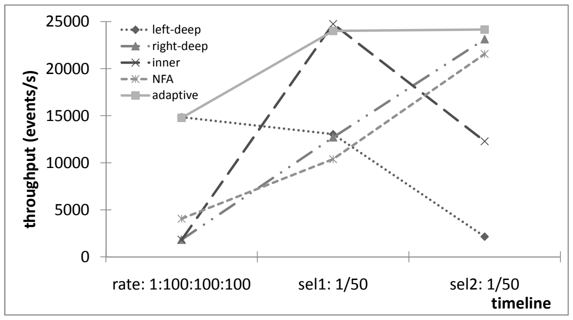

Figure 14: Throughput of different plans for Query 6 with concatenated input, vs. adaptive planner

6.3 Plan Adaptation

In this section, we describe experiments that test ZStream's plan adaptation features (described in Section 5.3), as well as our dynamic programming algorithm. For these experiments, we concatenated the three streams used in the previous experiment together and ran query 6 again. In this concatenated stream the rate of IBM is initially 100x less than the other stocks and the selectivities are both set to 1; then, the IBM rate become equal to the others but the selectivity of the first predicate goes to 1/50;

Table 3: Peak Memory Usage(in MB) for Query 6

		rate = $1:100:100:100$	**sel1 =** $1/50$
Peak Mem(MB)	Left-deep	7.36	7.06
	Right-deep	7.15	6.51
	Bushy	6.72	6.73
	Inner	7.58	6.47
	NFA	6.70	6.55

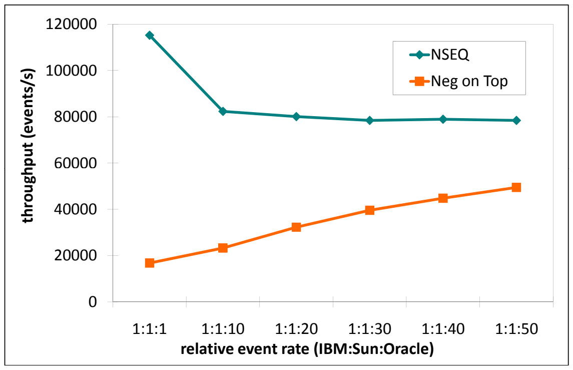

Figure 15: Throughputs of different plans for Query 7 varying *Oracle* rate

finally, the selectivity of the first predicate returns to 1, but the selectivity of the second predicate goes to 1/50. We compared the performance of our adaptive, dynamic programming based algorithm which continuously monitors selectivities and rates to the same fixed plans used in the previous experiment (we omit bushy for clarity in the figure.) Figure 14 shows the results, with throughput on the Y axis and the time on the X axis (with the stream parameters changing at the tick marks). We show three points for each approach. These points represent the average throughput for the three stream segments corresponding to the three varying sets of parameters. Notice that the adaptive algorithm is able to select a plan that is nearly as good as the best of the other plans. The performance of left-deep, right-deep, inner, and NFA is similar to the results shown in Figure 12.

6.4 Negation Push Down

As discussed earlier, one way to evaluate negation queries is to put a negation filter NEG on top of the entire query plan, and filter out the negated composite events as a post-filtering step. The alternative approach is to use the NSEQ operator to directly incorporate negation into the query plan tree. We compare the performance of these two methods in this section on Query 7:

Query 7. Negation Pattern "$IBM; !Sun; Oracle$"

```
PATTERN   IBM; !Sun; Oracle
WITHIN    200 units
```

The experiment is run on two query plans:

1. **Plan** 1: Use an NSEQ operator to combine Sun and $Oracle$ first. Then a SEQ operator is applied to combine IBM with the results from the NSEQ.
2. **Plan** 2: Use a SEQ operator to combine IBM and $Oracle$ first. Then a negation filter NEG is applied to rule out the $(IBM, Oracle)$ pairs where Sun events occurred in between.

Figure 15 and 16 illustrate the results for these two plans with varying relative event rates. In both figures, Plan 1 always outperforms Plan 2. When the event distribution is skewed, the performance increases faster for Plan 2 because it generates many fewer intermediate results compared to Plan 1. As shown in Figure 15, however, when $Oracle$ event rates increase, the throughput of Plan 1 decreases slightly. This is due to the way in which NSEQ works. As shown in Algorithm 2, NSEQ matches each $Oracle$ event o with

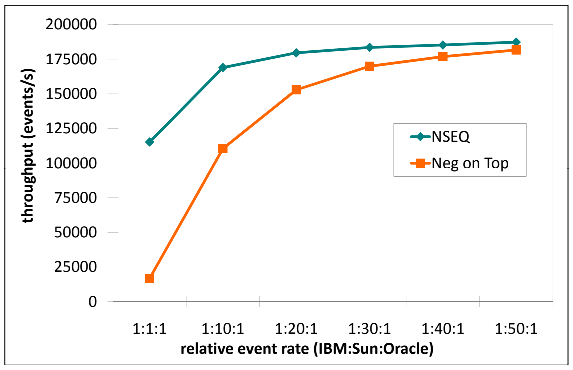

Figure 16: Throughputs of different plans for Query 7 varying *Sun* rate

Table 4: Number of Records Accessing Publications, Projects, and Courses

	publication	project	courses
# of accesses	6775	11610	16083

the latest Sun event before o. Hence, increasing the rate of Oracle will affect the amount of computation done by NSEQ, which counteracts the benefits introduced by the biased distribution here (we observe similar results when increasing the IBM event rates). Another observation is that the throughput of the Plan 2 increases much more quickly when the event distribution is biased towards Sun events. This is because the Plan 2 combines IBM and $Oracle$ first. The distribution biased on Sun will result in relatively fewer $(IBM, Oracle)$ pairs.

6.5 Web Access Pattern Detection

In this section, we demonstrate the efficiency of ZStream on real word web log data. The web log data contains one month period (from 22/Feb/2009 to 22/Mar/2009) of more than 1.5 million web access records from MIT DB Group web server. The records have the schema (Time, IP, Access-URL, Description).

Query 8. Web Access Pattern "$Publication; Project; Course$"

```
PATTERN   Publication; Project; Course
WHERE     same IP address
WITHIN    10 hours
```

In this data, we observed that some users who are downloading publications from our server are also interested in the web pages for research projects and courses offered by our group. To detect users with this access pattern, we wrote Query 8. We chose a sequential pattern here instead of a conjunction pattern because we wanted to compare with the performance of NFA, which doesn't support conjunction. The statistics of the number of records that access these different file types are shown in Table 4.

Figure 17 shows the performance of three different plans (left-deep, right-deep and NFA) on Query 8. Here a left-deep plan uses a SEQ operator with publications and projects, followed by a SEQ with courses, while a right deep plan accesses projects and courses first, and then combined with publications. The left-deep plan is much faster than the other two because the number of accesses to publications is

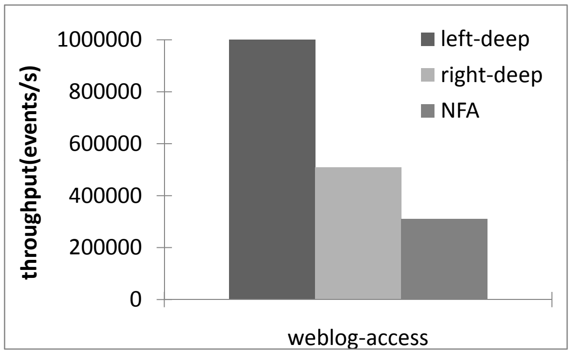

Figure 17: Throughputs of different plans for Query 8 on one month web access log data

Table 5: Peak Memory Usage (in MB) for Query 8

	left-deep	right-deep	NFA
Peak Mem(MB)	10.13	10.66	10.55

much smaller than the number of accesses to projects and courses as shown in Table 4. Hence many fewer intermediate results are generated by the left deep plan. This is consistent with the results from Section 6.1.2. NFA is a little slower than right-deep plan because we do not materialize for NFA, which is relatively important in this case because Query 8 has a very long time window (10 hours) and most of materialized intermediate tuples can be reused. The peak memory usage for these three plans is shown in Table 5.

7. CONCLUSION

This paper presented ZStream, a high performance CEP system designed and implemented to efficiently process sequential patterns. ZStream is also able to support other relations such as conjunction, disjunction, negation and Kleene Closure. Unlike previous systems that evaluate CEP queries in a fixed order using NFAs, ZStream uses a tree-based plan for both the logical and physical representation of query patterns. A single pattern may have several equivalent physical tree plans, with different evaluation costs. Hence, we proposed a cost model to estimate the computation cost of a plan. Our experiments showed that the cost model can capture the real evaluation cost of a query plan accurately. Based on this cost model and using a simple set of statistics about operator selectivity and data rates, we showed that ZStream is able to adjust the order in which it detects patterns on the fly. In addition to these performance benefits, a tree-based infrastructure allows ZStream to unify the evaluation of sequences, conjunctions, disjunctions, sequential negations and Kleene Closures as variants of the join operator. This formulation allows flexible operator ordering and intermediate result materialization.

8. ACKNOWLEDGMENTS

We thank Brian Cooper, Donald Kossmann and other reviewers for their invaluable suggestions to this paper. We also thank Michael Stonebraker for his insightful advices for us. This work was supported under NSF Grant NETS-NOSS 0520032.

9. REFERENCES

[1] Event Processing Workshop, March 2008. `http://complexevents.com/?page_id=87`.

[2] StreamBase corporate homepage, 2009. `http://www.streambase.com/`.

[3] Coral8 corporate homepage, 2009. `http://www.coral8.com/`.

[4] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, 2008.

[5] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, 1994.

[6] U. Dayal et al. The hipac project: Combining active databases and timing constraints. *SIGMOD RECORD*, 17(1):51–70, March 1988.

[7] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *CIDR*, January 2007.

[8] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, 2001.

[9] S. Gatziu and K. Dittrich. Events in an active object-oriented database. In *Workshop on Rules in Database Systems*, 1994.

[10] N. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *VLDB*, September 1991.

[11] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[12] A. Majumder, R. Rastogi, and S. Vanama. Scalable regular expression matching on data streams. In *SIGMOD*, 2008.

[13] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *ACM TODS*, 29(2), June 2004.

[14] P. Selinger et al. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[15] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.

[16] Y. Zhu, E. Rundensteiner, and G. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, 2004.