ZStream: A Cost-based Query Processor for Composite Event Detection

by

Yuan Mei

B.S. in Electrical Engineering and Computer Science
Peking University, Beijing, China, 2004

B.S. in China Center for Economic Research
Peking University, Beijing, China, 2005

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology
June 2008

©2008 Massachusetts Institute of Technology. All rights reserved.

Signature of Author

Department of Electrical Engineering and Computer Science
May 23rd, 2008

Certified by

Samuel Madden
Associate Professor, EECS
ITT Career Development Professor
Thesis Supervisor

Accepted by

Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses
ZStream: A Cost-based Query Processor for 

Composite Event Detection

by

Yuan Mei

Submitted to the Department of Electrical Engineering and Computer Science on May 23rd, 2008 in Partial Fulfillment of the Requirements for the Degree of Master of Science in Electrical Engineering and Computer Science

ABSTRACT

Composite (or Complex) event processing (CEP) systems search sequences of incoming primitive events for occurrences of user-specified event patterns. Recently, they are gaining more and 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 relates events to each other. Examples 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), detecting intrusion in network monitoring (where a specific sequence of malicious activities is detected) or catching break points in debugging systems (where a sequence of function calls are made). But even searching for a simple sequence pattern involving only equality constraints between its components is an NP-complete problem. Furthermore, simple sequentiality is 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, making the matching problem even more complex.

In this thesis, we present a CEP system called ZStream to efficiently process such sequential patterns. Besides simple sequentiality, ZStream is also able to support other relations such as conjunction, disjunction, negation and Kleene Closure. ZStream uses a tree-based plan for both the logical and physical representation of query patterns. Using this tree-based infrastructure, ZStream is able to unify the evaluation of sequence, conjunction, disjunction, negation, and Kleene Closure as variants of the join operator. A single pattern may have several equivalent physical tree plans, with different evaluation costs. Hence a cost model is proposed to estimate the computation cost of a plan. Experiments show that our 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, ZStream is able to adjust the order in which it detects patterns. In addition, we design a dynamic programming algorithm and propose equivalent transition rules to automatically search for an optimal query plan for a given pattern.
Thesis Supervisor: Samuel Madden
Associate Professor, EECS
ITT Career Development Professor
ACKNOWLEDGEMENTS

This thesis would not have been possible without the contributions and support of a number of people, especially my thesis supervisor, Prof. Samuel Madden.

I would like to extend my deepest thanks to Prof. Samuel Madden. Since I came to MIT, he has given me enormous help in both my life and study. Meanwhile, He is also the person who influences me most on my research and encourages me to make progress in every aspect. From the early days of elaborating the research question to the final stages of thesis proofreading, Sam has always been a source of inspiration and invaluable guidance. This thesis is written in clear English only thanks to Sam’s careful and patient revision of my work. To catch up my time, he even sacrifices his weekends and holidays to do the final revision.

I would like to extend my sincere thanks to Professor Michael Stonebraker and Professor Hari Balakrishnan. Mike has patiently listened to my talk, supplied thoughtful suggestions and encouraged me to go further. Hari offered generous help on related resources and suggested many applications for our system.

I would like to extend my special thanks to Eugene Wu, who provided many details on the SASE project. I would also thank to Arvind Thiagarajan, Bret Hull, Vladimir Bychkovskiy, and Sejoon Lim. They supply CarTel data set to experiment on.

Many thanks to my office mates Lewis Girod and Stanislav Rost. They are so kind people and always ready to help. I also need to thank all my friends at MIT and elsewhere for their wonderful friendship which makes my life valuable and enjoyable.

Finally, I would like to dedicate this thesis to my parents and my boyfriend Wen Feng for providing me endless courage to surmount all the difficulties on my way. Without their love, I would have never made it this far.
**TABLE OF CONTENTS**

List of Figures ........................................................................................................................... 11

List of Tables ............................................................................................................................ 13

List of Queries ............................................................................................................................ 14

List of Algorithms ....................................................................................................................... 15

List of Formulas ........................................................................................................................... 16

Chapter 1: Introduction ............................................................................................................... 17

1.1 Motivation ............................................................................................................................... 17

1.2 Related Work .......................................................................................................................... 19

1.3 Contributions ........................................................................................................................ 22

1.4 Thesis Organization ................................................................................................................. 23

Chapter 2: Language Specification and Applications ............................................................... 25

2.1 Language Specification .......................................................................................................... 25

2.1.1 Basic Concepts .................................................................................................................... 25

2.1.2 Event Operators .................................................................................................................. 27

2.2 Applications ............................................................................................................................. 29

2.2.1 Car Tracking ......................................................................................................................... 29

2.2.1.1 Semantic Model .............................................................................................................. 31

2.2.1.2 Aggregation ..................................................................................................................... 33

2.2.1.3 Other Query Examples .................................................................................................. 34

2.2.2 Stock Market Monitoring .................................................................................................. 35

2.2.3 Distributed System Debugging .......................................................................................... 37

2.2.4 Network Monitoring .......................................................................................................... 38

2.3 Conclusions ............................................................................................................................ 39

Chapter 3: System Architecture and Operator Evaluation ....................................................... 41

3.1 Introduction ............................................................................................................................. 41

3.2 Tree-Based Model ................................................................................................................... 41
Chapter 4: Cost Model and Optimization ................................................. 69
   4.1 Cost Model .............................................................................. 69
      4.1.1 Operator Cost .................................................................. 73
      4.1.2 Tree Plan Cost .................................................................. 76
   4.2 Equivalent Query Plans ............................................................... 77
      4.2.1 Rule-Based Transformations ............................................... 78
      4.2.2 Operator Reordering ........................................................... 80
         4.2.2.1 Optimal Operator Order Algorithm .............................. 80
   4.3 Conclusions ............................................................................. 84

Chapter 5: Performance Evaluation ...................................................... 85
   5.1 Experiment Setup ....................................................................... 85
   5.2 Batch-Iterator Model Evaluation.................................................. 86
   5.3 System Scalability ..................................................................... 89
   5.4 Parameters Affecting Costs ........................................................ 90
      5.4.1 Parameterized Predicates Selectivity .................................. 90
      5.4.2 Event Rates ........................................................................ 94
   5.5 Negation Pushed Down ............................................................... 96
   5.6 Conclusions ............................................................................. 99

Chapter 6: Conclusions and Future Work .............................................. 101
6.1 Conclusions........................................................................................................101
6.2 Future Work.......................................................................................................102

References.............................................................................................................103

Appendix...................................................................................................................105
LIST OF FIGURES

Figure 1-1: Example for NFA-based Model to Process Sequential Pattern .................. 18
Figure 2-1: Two Traces in the CarTel Portal ................................................ 30
Figure 2-2: Illustration of the Car Tracking Points along the Time Line ................. 33
Figure 2-3: Distributed System Debugging Example ....................................... 37
Figure 3-1: Left-deep Tree Plan for Query 2-7 ............................................... 42
Figure 3-2: Tree Plan for Query 2-8 ............................................................ 44
Figure 3-3: Tree Plan for Query 2-5 with NEG on top ..................................... 45
Figure 3-4: Tree plan for Query 2-5 with NEG Pushed Down ............................. 47
Figure 3-5: Example Buffer Structure ............................................................ 48
Figure 3-6: Example to Show the 1st Round of the Batch-Iterator Model with the Batch Size Equal to 6 ................................................................. 53
Figure 3-7: Example to Show the 6th Round of the Batch-Iterator Model with the Batch Size Equal to 1 ................................................................. 53
Figure 3-8: Example of the Sequence Operator Assembling Events Together ........ 56
Figure 3-9: Example of the Conjunction Operator Evaluation .......................... 57
Figure 3-10: Example of the Disjunction Operator Evaluation .......................... 59
Figure 3-11: Example of the K-Seq Operator Evaluation .................................. 60
Figure 3-12: Example of the K-Seq Operator Evaluation with only Middle Event Class ................................................................. 61
Figure 3-13: Example Sequential Negation Evaluation Using a Left N-Seq ............ 64
Figure 3-14: Illustration How to Decide the Extra Time Constraints for a Left N-Seq 65
Figure 3-15: Example Sequential Negation Evaluation Using a right N-Seq .......... 66
Figure 4-1: Two Query Plans for Query 2-7, the Left-deep Plan and Right -deep Plan 72
Figure 4-2: Example Tree Plan to Show Optimal Substructure for Sequential Pattern 81
Figure 4-3: Example to Illustrate the Algorithm 4-1 ......................................... 82
Figure 4-4: An Optimal Tree Plan Resulted from the Example Given in Figure 4-3 ... 83
Figure 4-5: Example to Illustrate Holes in a Composite Event ........................... 84
Figure 5-1: Throughputs of the Left-deep Plan of Query 5-1 with Increasing Batch Size ................................................................. 87
Figure 5-2: Throughputs of the Left-deep Plan and the Right-deep Plan of Query 5-1 with Increasing Input Size ................................................................. 89
Figure 5-3: Throughputs of the Left-deep Plan and the Right-deep Plan of Query 5-1 with Various Selectivities for the Predicate Trade1.price > Trade2.price .......... 91
Figure 5-4: Throughputs of the Left-deep Plan and the Right-deep Plan of Query 5-2 with Various Selectivities for the Predicate Trade4.price > Trade5.price .......... 91

Figure 5-5: Running Time, Estimated Costs and the Revised Costs of the Left-deep Plan and the Right-deep Plan of Query 5-1 with Various Selectivities for the Predicate Trade1.price > Trade2.price ........................................ 93

Figure 5-6: Running Time and Estimated Costs of the Left-deep Plan and the Right-deep Plan of Query 5-3 with Various Event Rates between Relevant Events (IBM: Sun: Oracle) ........................................................................ 95

Figure 5-7: Running Time and Estimated Costs of the Left-deep Plan and the Right-deep Plan of Query 5-3 with Various Event Rates between Relevant events (IBM: Sun: Oracle), and Huge Rate Differences ........................................... 95

Figure 5-8: Running Time of Two Plans of Query 5-4 with Various Relative Event Rates. One with Negation Pushed down, and the Other with Negation on top......... 98
LIST OF TABLES

Table 2-1: Schema for CarTel Tracking Points...................................................30
Table 2-2: Schema for Stock Data........................................................................35
Table 4-1: Estimated Statistics...........................................................................73
Table 4-2: Input Cost Formula and Output Cost Formula for Individual Operators.....74
Table 4-3: Equivalent Transition Rules.............................................................78
Table 5-1: Parameter Settings for Input Data......................................................88
LIST OF QUERIES

Query 2-1: Find all the event pairs where an event A is followed by an event B having the same id within 2 hours using consumption policy MATCH-MANY........27
Query 2-2: Get all the drives from Boston to Framingham via I-90 within two hours....31
Query 2-3: Find all locations on I-90 where a car exceeded the average speeds via Boston and Framingham within two hours..................................................32
Query 2-4: Find all the drives from Boston to Framingham via I-90 within two hours where its average speed on I-90 exceeds 90m/h..........................34
Query 2-5: Find all the drives from Boston to Framingham not via I-90 within two hours.................................................................34
Query 2-6: Find all the drives from Boston to Framingham via either I-90 or I-93 within two hours........................................................................35
Query 2-7: Find a trade of IBM with price x% higher than the following Google trade, followed by another IBM trade with price y% lower within 10 seconds......36
Query 2-8: Find a trade of IBM with price x% higher than the average price of the next 5 successive Google trades, followed by another IBM trade with price y% lower within 10 seconds........................................................................36
Query 2-9: Find a trace that completes a web service from the client................38
Query 2-10: Find the IP address that constructs a TCP connection with the desip......39
Query 5-1: Find a trade of IBM followed by Sun with lower price, followed by an Oracle trade within 200 time units.........................................................86
Query 5-2: Find a sequential pattern of the form “IBM trade; Sun trade; Oracle trade; Google trade; Yahoo trade” where the price of Google is higher than that of Yahoo within 100 time units.....................................................90
Query 5-3: Find a trade of IBM followed by Sun, followed by an Oracle trade within 200 time units.............................................................94
Query 5-4: Find a trade of IBM followed an Oracle trade within 200 time units where no Sun trade occurs in between...........................................97
LIST OF ALGORITHMS

Algorithm 3-1: Sequence Evaluation for Each Batch Round.............................55
Algorithm 3-2: Conjunction Evaluation for Each Batch Round............................58
Algorithm 3-3: Disjunction Evaluation for Each Batch Round.............................59
Algorithm 3-4: K-Seq Evaluation for Each Batch Round..............................62
Algorithm 3-5: Left N-Seq Evaluation for Each Batch Round............................66
Algorithm 4-1: Searching for the Optimal Operator Order...............................81
LIST OF FORMULAS

Formula 4-1: Estimated Cost Formula.................................................................70
Formula 4-2: Tree Plan Cost Formula.................................................................76
Formula 4-3: Operator Cost Inequality Formula...............................................79
Formula 5-1: Revised Cost Formula.................................................................92
CHAPTER 1: INTRODUCTION

1.1 Motivation

Composite (or Complex) event processing (CEP) systems search sequences of incoming primitive events for occurrences of user-specified event patterns. They are gaining more and more attention in a variety of areas due to their powerful and expressive query language and performance potential [CEPW08]. Sequentiality (temporal ordering) is the primary way in which CEP relates events to each other. Examples 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), detecting intrusion in network monitoring (where a specific sequence of malicious activities is detected) or catching break points in debugging systems (where a sequence of function calls are made). But even searching for a simple sequence pattern involving only equality constraints between its components is an NP-complete problem. Furthermore, simple sequentiality is 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, making the matching problem even more complex.

Currently, non-deterministic finite automata (NFA) is a commonly used method for evaluating CEP queries [WDR06] [DGH+07]. As shown in Figure 1-1, 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 the final state. However, the NFA-based model itself has three fundamental shortcomings:

1. It is hard to support concurrent event detection (events that occur simultaneously within a time window) in an NFA-based model.
2. It is hard to express negation (events that do not occur) in an NFA; existing systems require it to be done as a post-NFA filtering step.

3. An NFA-based model is typically evaluated in a fixed order determined by the state transition trace. For example, the NFA showed in Figure 1-1 starts from state 1 to its final state 4 via states 2 and 3; then the evaluation order of the operators connecting events in the NFA-based model is 4-3-2-1. As we will show in Chapter 2, applications that involve sequentiality usually have relatively long patterns (3 to 8), and evaluation in a fixed order is inefficient.

![Figure 1-1: NFA-based model to process the sequential pattern “Event A followed by Event B followed by Event C”. States of the NFA are marked by numbers and its final state is depicted by grey color. When the NFA reaches its final state, one pattern is said to be matched.](image)

In this thesis, we describe the design and implementation of a CEP system called \textit{ZStream}\(^1\) aimed at sequential applications that addresses all these limitations of the NFA-based approach. We define a notion of an optimal plan and show that ZStream can automatically choose an optimal or near-optimal plan to effectively detect composite events that match a specified pattern or query. To achieve this, we developed a tree-based infrastructure that is amenable to a variety of optimizations. This tree-based infrastructure allows ZStream to unify the evaluation of sequences, conjunctions and disjunctions, the three main operators in many CEP specifications, as variants of the \textit{join} operator. This formulation allows flexible operator ordering and intermediate result materialization. We also propose a new way to formulate and evaluate negation queries so they can be incorporated into the tree model and the cost model just like other operators, rather than being applied as a final filtration step. Using a simple set of statistics about operator

---

\(^1\) ZStream system is a younger sister of XStream system [GMN+08]. The name ZStream is taken from a Chinese word \textit{Zhi} meaning sequentiality and order, as this system is designed to detect sequential patterns efficiently.
selectivity and data rates, ZStream is also able to adjust the order in which it detects patterns.

1.2 Related Work

In this section, we review related work on CEP systems. CEP systems first appeared as trigger detection systems in active databases [DAYAL88] [GJ91] [GD94] [CKAK94] to meet the requirements of active functionality that are not supported by traditional databases. These CEP systems were primarily designed to detect changes in the database’s state. They used finite state automata (FSA) or Petri Nets to capture the state changes and transitions.

Examples of such work include:

- HiPAC [DAYAL88] and Ode [GJ91] use FSAs to detect composite events. In these two prototype systems, composite events are expressed using regular expressions. Events are fed into an FSA according to their occurrence time. If the FSA enters an accepting state, then a composite event is said to be detected.

The disadvantages of the HiPAC and Ode are that they were pure FSAs. Hence,

1. It is difficult for them to support concurrent event detection, because transitions in FSAs inherently incorporate some orders between states.

2. Ode does not support parameterized predicates (predicates between event classes in the composite pattern), as it only allows constraints to be defined within a single event class.

- The SAMOS [GD94] project uses Colored Petri Nets to detect composite events. A
Petri Net consists of places, transitions, and arcs connecting places with transitions. Places may contain any numbers of tokens, which correspond to events that have occurred. At runtime, when an event occurs, a corresponding token is inserted into all places representing the event type. A transition “fires” if each of its input places has tokens. When a transition fires, it consumes the tokens from its input places, performs some tasks and then puts some tokens into each of its output places.

Colored Petri Nets can support concurrent event detection and parameterized predicates well, but are very complex to express and evaluate, making them poorly suited to supporting complicated patterns.

The Sentinel [CM93] [CKAK94] project evaluates its event language Snoop using event trees. Each composite event pattern in Snoop is represented as an event tree structure. Leaves of a tree stand for primitive events and internal nodes represent the intermediate composite events. These internal nodes are expressed as event operators along with relevant parameter settings; operand events are supplied by child nodes.

Snoop is interesting because it supports different consumption policies for different applications, which are surprisingly ignored by most other research. A consumption policy influences the semantics of an operator by specifying what happens to the input events after they are used in a pattern. For example, the input events of each operator can match with other events just once or multiple times. As shown in Chapter 2, these different consumption policies are very important and useful in a variety of real applications.

The problem with Sentinel is that it is not designed to deal with high rate incoming events. Most of the events it is supposed to accept are database events, such as inserts,
deletes, updates, etc. These events are much less frequent than events in many high rate CEP applications; for example, stock market monitoring may require processing thousands of events per second.

Additionally, Sentinel has shortcomings in the following three aspects:
1. It does not support the negation operator in its language specification.
2. It does not take advantage of temporal relations between events, hence is inefficient for sequential event pattern detection.
3. It arbitrarily chooses a tree structure for evaluation rather than carefully searching for the optimal plan, which, as we show, is particularly bad when event rates and predicate selectivity vary substantially.

Other research [SZZA04] has tried to make use of string-based matching algorithms, including the KMP, GSW and OPS algorithms to evaluate composite event patterns. All these regular expression or string-based matching algorithms only work efficiently for strictly consecutive patterns, limiting the expressive capability of the pattern matching language.

More recently, streaming processing engines (SPEs) [ACC+03] [CCD+03] [CORAL] [SBASE] and publish/subscribe systems [FJL+01] have emerged as a result of requirements for more robustness and efficiency in real time data management. But SPEs by themselves are not suitable for CEP in that, in most cases, their language specifications were adapted from traditional database algebra. Pub/sub systems suffer from a similar problem because they were designed for filtering on individual events, rather than relating multiple events as in CEP.

To support both composite event pattern detection and high rate incoming events, SASE
[WDR06], a high performance CEP, was recently proposed. It simplifies the composite pattern expression language, and achieves good performance through a variety of optimizations. But SASE is essentially NFA-based, inheriting all the limitations of the NFA-based model listed in Section 1.1. Also, it does not distinguish between different event consumption policies, and hence its semantics are somewhat ambiguous.

Recently, the Cayuga [DGH+07] CEP system was also proposed. Since it is developed from a pub/sub system, it is focused on finding and efficiently evaluating common sub-expressions amongst multiple patterns instead of on optimization issues related to individual pattern evaluation. In addition, Cayuga is also NFA-based; hence it suffers all the limitations of the NFA-based model.

1.3 Contributions

In this thesis, we present a high performance CEP system ZStream that is extremely efficient for applications dominated by sequential queries. Specifically, we make the following contributions:

1. We design a tree-based data structure that can take advantage of the temporal relations between individual events. Using this data structure, it is efficient to locate the range of events needed for query evaluation.
2. We propose a cost model to estimate the total cost for a given query plan. By employing a dynamic programming algorithm, we can search for a query plan with minimal cost automatically. Our experiments show that the performance of different plans varies dramatically, suggesting plan optimization is an important issue.
3. We propose a new way to formulate and evaluate the negation operator such that we
can incorporate it into the tree model and the cost model just like other operators. In that sense, we successfully unify the evaluation of sequences, conjunctions, disjunctions and negations as variants of the join operator.

4. We design a batch-iterator model to evaluate query plans, which allows us to bound the intermediate buffer size and memory utilization of our plan evaluation algorithm.

5. We make use of buffered intermediate results to avoid repeated combination of primitive events.

6. We develop and adopt a suite of consumption policies to eliminate semantic ambiguity.

1.4 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 describes some basic concepts and terms used in composite event pattern detection, which will be used later in this thesis. It also presents some examples of sequential queries in real world applications to show the broad applicability of such queries.

Chapter 3 describes the system architecture. It illustrates the tree-based model used to internally express composite event patterns and describes the batch-iterator model used for evaluation. It also discusses the details of the evaluation algorithms for the five main relational operators supported in ZStream: sequence, conjunction, disjunction, Kleene Closure and negation.

Chapter 4 presents the cost model used to select an optimal query plan and discusses two
optimizations performed in ZStream: rule level optimization and operator order optimization. Finally it describes the dynamic programming algorithm that ZStream uses to automatically search for the optimal plan.

Chapter 5 demonstrates the effectiveness of ZStream through a series of experiments. It shows that our cost model described in Chapter 4 can capture the real system performance behavior accurately. Finally, it verifies our claim that the performance of different query plans for a single pattern may vary dramatically, suggesting the importance of plan optimization.

Chapter 6 concludes.
CHAPTER 2: LANGUAGE SPECIFICATION AND APPLICATIONS

2.1 Language Specification

This chapter introduces several real applications that use CEP queries to express sequential relations between individual events. In the rest of the thesis, we refer to such applications as sequential applications. Before elaborating on details of these applications, we first discuss some generally accepted CEP terminology and our language specification. Since our work is not focused on designing a complete CEP language, we will only mention the language features used in this thesis.

2.1.1 Basic Concepts

We begin with a few basic definitions:

- **Primitive Event**: predefined single occurrence of interest that can not be split into any smaller events.
- **Composite Event**: event detected by the CEP system from a collection of primitive and/or other composite events.
- **SARGable Predicates**: predicates that involve only one event class.
- **Parameterized Predicates**: predicates that involve more than one event class.

In addition, it is necessary to differentiate between the concepts of event class and event instance. As with schemas and records in traditional databases, the event class declares the schema of a group of event instances. Each event class is associated with several
attributes, such as speed, price, etc; these attributes are instantiated with real values at runtime when the event instances actually occur.

Each event is associated with a start-timestamp and an end-timestamp, with the end-timestamp stands for its time of occurrence. For primitive event, its start-timestamp and end-timestamp are equal to each other, so they will just be referred as timestamp. Some CEP systems like SASE make the assumption that a composite event occurs at a time point (end-timestamp), and ignores its occurrence duration. This assumption makes it vague to further assemble these composite events. For example, suppose $A$ and $B$ are two composite events and they satisfy the pattern “$A$ followed by $B$ in a time window $tw$” according to their end-timestamps. However, the real occurrence time of “$A$ followed by $B$” may exceed the time window constraint. Since $A$ and $B$ themselves may be generated in a similar way, the resulting composite event may have arbitrarily long occurrence duration, and the time window constraint does not constrain anything in this case.

Generally, primitive events arrive into the CEP system from various external event sources, while composite events are internally generated from the CEP system itself by specifying various composite event patterns, like:

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>Composite Event Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>Value Constraints</td>
</tr>
<tr>
<td>WITHIN</td>
<td>Time Constraints</td>
</tr>
<tr>
<td>RETURN</td>
<td>Output Expression</td>
</tr>
<tr>
<td>MATCH</td>
<td>Consumption Policy</td>
</tr>
</tbody>
</table>

Composite Event Expressions describe an event pattern to be matched by connecting various event classes together via different event operators; Value Constraints define the context for the composite events by imposing SARGable and/or parameterized predicates on their corresponding attributes; Time Constraints describe the time window during
which events that match the pattern must occur. Time constraints can be specified either absolutely (e.g. "2008-5-7 3:00 pm – 4:00 pm") or relatively (e.g. "WITHIN 2 hours").

Since absolute time is very straightforward, we focus on supporting relative time in this thesis. The RETURN clause defines the expected output stream from the pattern query. Finally, the MATCH clause specifies the consumption policy used by this query. By default, ZStream uses the MATCH-MANY policy, as we will discuss more later. Query 2-1 shows a simple example pattern query to “find all the event pairs where an event A is followed by an event B having the same id within 2 hours using the consumption policy MATCH-MANY”. Notice that the symbol ";" in the PATTERN clause is an operator to sequentially connect event classes, meaning the left operand is followed by the right operand. We will discuss more operators in the next section.

Query 2-1: Find all the event pairs where an event A is followed by an event B having the same id within 2 hours using consumption policy MATCH-MANY.

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>A; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>A.id = B.id</td>
</tr>
<tr>
<td>WITHIN</td>
<td>2 hours</td>
</tr>
<tr>
<td>RETURN</td>
<td>A.id</td>
</tr>
<tr>
<td>MATCH</td>
<td>MATCH-MANY</td>
</tr>
</tbody>
</table>

2.1.2 Event Operators

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

- Sequence: A; B

As indicated in Query 2-1, the sequence operator finds instances of event B that
follow event A (within the specified time window). We define a sequential composite event C as
1. \( A.end-timestamp < B.start-timestamp \) and
2. \( C.start-timestamp = A.start-timestamp \) and
3. \( C.end-timestamp = B.end-timestamp \) and
4. \( C.end-timestamp - C.start-timestamp \leq \text{time window} \)

The sequence operator is the primary way for queries in ZStream to express sequentiality. Hence in this thesis, we focus on optimization and evaluation of the sequence operator.

- **Conjunction:** \( A \& B \)
  Conjunction means both event A and event B occur (within the specified time window), and their order does not matter. We define a conjunction composite event C as
  1. \( C.start-timestamp = \text{MIN}(A.start-timestamp, B.start-timestamp) \) and
  2. \( C.end-timestamp = \text{MAX}(A.end-timestamp, B.end-timestamp) \) and
  3. \( C.end-timestamp - C.start-timestamp \leq \text{time window} \)

- **Disjunction:** \( A \mid B \)
  Disjunction means either event A or event B or both occurs (within the specified time window). Since we allow both A and B to occur in disjunction, it can be just considered as a union of the two event classes. In other words, the generated composite events are simply event instances coming from A or B, except that the duration of these instances must be within the specified time window.

- **Kleene Closure:** \( A^{\ast} / A^{+} / A^{\text{num}} \)
Kleene Closure means that event A needs to 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 5 successive $A$ event instances will be grouped together.

- **Negation:** $!A$

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

### 2.2 Applications

This section illustrates the language concepts described above and further discusses different consumption policies via several examples from real applications.

#### 2.2.1 Car Tracking

This application is motivated by the CarTel project [HBC+06], a distributed, mobile sensor network built at MIT. It consists of an embedded computer connected with a variety of sensors in each car. It processes and transmits the collected data back to a central server via a combination of WiFi, Bluetooth, and cellular connectivity. Its portal organizes the collected data into traces, which include a time stamped series of sensor readings and GPS coordinates. As an illustration, two traces are shown in Figure 2-1, one in red and the other in green.
Table 2-1 shows the schema of the CarTel data. We will consider several CEP queries over this data.

**Table 2-1: Schema for CarTel Tracking Points**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>carid</td>
<td>Integer</td>
<td>The identification for the traveling car</td>
</tr>
<tr>
<td>Lat</td>
<td>Float</td>
<td>The latitude of the tracking point</td>
</tr>
<tr>
<td>Long</td>
<td>Float</td>
<td>The longitude of the tracking point</td>
</tr>
<tr>
<td>Roadname</td>
<td>String</td>
<td>The road name the tracking point belongs to</td>
</tr>
<tr>
<td>speed</td>
<td>Float</td>
<td>The speed of the car detected at this point</td>
</tr>
<tr>
<td>timestamp</td>
<td>Integer</td>
<td>The time stamp of the tracking point. It is transformed to the standard millisecond time value.</td>
</tr>
</tbody>
</table>

Interstate 90 in Massachusetts is a toll road. One interesting question is whether cars try to avoid tolls by taking alternative routes. For example, the fastest way from Boston to
Framingham is via I-90. Hence it might be interesting for us to know the percentage of cars that go from Boston to Framingham using I-90. To get all the drives from Boston to Framingham via I-90 within two hours, we can issue a query like:

**Query 2-2: Get all the drives from Boston to Framingham via I-90 within two hours.**

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>B; I; F</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>B.carid = I.carid = F.carid</td>
</tr>
<tr>
<td>AND</td>
<td>B.lat = 'Blat' AND B.long = 'Blong'</td>
</tr>
<tr>
<td>AND</td>
<td>I.roadname = 'I-90'</td>
</tr>
<tr>
<td>AND</td>
<td>F.lat = 'Flat' AND F.long = 'Flong'</td>
</tr>
<tr>
<td>WITHIN</td>
<td>2 hours</td>
</tr>
<tr>
<td>RETURN</td>
<td>B.carid, F.timestamp - B.timestamp</td>
</tr>
</tbody>
</table>

In the above query, the pattern "B; I; F" stands for three sequential tracking points captured by the GPS. Once the system matches the pattern, it returns the pair of carid and the time the car took to travel from Boston to Framingham. To ease expression and understanding, we use the phrases 'Blat', 'Blong', 'Flat' and 'Flong' in place of the corresponding latitudes and longitudes for Boston and Framingham.

### 2.2.1.1 Semantic Model

Notice that Query 2-2 does not specify the consumption policy it uses; thus raises an interesting semantic issue: there may be many GPS points along I-90 in a trace; hence the pattern "B; I; F" may produce many results for each B-F pair if we allow this pair to match with several I events. Thus, the meaning of this sequential pattern is ambiguous. In other words, we need to clarify the kind of tracking points (event instances) allowed or not allowed to exist in between tracking points B and F. To do this, we define two related concepts:
- Relevant Events: These are events that belong to any event class in the query pattern and also pass all the SARGable predicates associated with that event class.

- Irrelevant Events: all other events.

We allow any number of irrelevant events to appear in the pattern; negation needs to be applied explicitly to prevent irrelevant events from occurring. For relevant events, we provide several different consumption models:

- Match-once
  
  Once an event is matched with other events to form a pattern, it is deleted from the buffer and can not be used again. For example, in the query given above, the user is only interested in one match per drive.

- Match-many
  
  Each event can match multiple times with other events to form a pattern. In other words, it allows all possible combination of qualified events. For example, we need this match-many model to express the query “find all locations on I-90 where a car exceeded the average speeds via Boston and Framingham”.

**Query 2-3:** Find all locations on I-90 where a car exceeded the average speeds via Boston and Framingham within two hours

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>B; I; F</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>B.carid = I.carid = F.carid</td>
</tr>
<tr>
<td>AND</td>
<td>B.lat = 'Blat' AND B.long = 'Blong'</td>
</tr>
<tr>
<td>AND</td>
<td>I.roadname = 'I-90'</td>
</tr>
<tr>
<td>AND</td>
<td>F.lat = 'Flat' AND F.long = 'Flong'</td>
</tr>
<tr>
<td>AND</td>
<td>I.speed &gt; ( B.speed + F.speed ) / 2</td>
</tr>
<tr>
<td>WITHIN</td>
<td>2 hours</td>
</tr>
<tr>
<td>RETURN</td>
<td>B.carid, I.pos</td>
</tr>
<tr>
<td>MATCH</td>
<td>MATCH-MANY</td>
</tr>
</tbody>
</table>
As an example of these two consumption policies, consider Figure 2-2, which illustrates the GPS locations of a car along a time line. The capital letter denotes the event class the tracking point belongs to, and the integer beside it denotes the corresponding timestamp. If we match the pattern “B; I; F” using the match-once policy, the detected composite event would be “B1; I4; F5”; using the match-many policy, the detected events would be “B1; I2; F5”, “B1; I3; F5” and “B1; I4; F5”.

![Time Line](image)

Figure 2-2: Illustration of the car tracking points along the time line. The capital letter denotes the event class the tracking points belonging to and the integer denotes its corresponding timestamp.

The match-once policy has the limitation that sometimes it is unclear which event to match with. In the example shown in Figure 2-2, we match with the most recent tracking point, but the user may prefer to match with the oldest event or some arbitrary event. The match-many policy does not have this problem if its time constraints are assigned properly.

The match-once policy is well-supported by the DFA model [GJ91] and the tree model [CM93], while match-many policy can be supported by the NFA model [WDR06] and also the tree model. In this thesis, we will focus the discussion on evaluation and optimization of queries that use the match-many policy. If not specified explicitly, ZStream adopts the match-many policy by default.

2.2.1.2 Aggregation

Another important feature of ZStream is that it supports aggregates in the pattern. Query 2-4 illustrates an example. In the PATTERN clause, \( I^+ \) means successive occurrences of
GPS points on I-90; In the value constraints clause, the aggregation function $\text{avg}()$ is applied to the attribute $\text{speed}$ of all the events in the Kleene Closure $I'$. 

Query 2-4: Find all the drives from Boston to Framingham via I-90 within two hours where its average speed on I-90 exceeds 90$m/h$.

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>$B; I'; F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>$B.carid = I.carid = F.carid$</td>
</tr>
<tr>
<td>AND</td>
<td>$B.lat = 'Blat'$ AND $B.long = 'Blong'$</td>
</tr>
<tr>
<td>AND</td>
<td>$I'.roadname = 'I-90'$</td>
</tr>
<tr>
<td>AND</td>
<td>$F.lat = 'Flat'$ AND $F.long = 'Flong'$</td>
</tr>
<tr>
<td>AND</td>
<td>$\text{avg}(I'.speed) &gt; 90 \text{ m/h}$</td>
</tr>
<tr>
<td>WITHIN</td>
<td>2 hours</td>
</tr>
<tr>
<td>RETURN</td>
<td>$B.carid, \text{avg}(I'.speed)$</td>
</tr>
</tbody>
</table>

In the stock application described in Section 2.2.2, we will show how we can control the length of the aggregated events, which is particularly useful for stock market monitoring with sliding time constraints.

2.2.1.3 Other Query Examples

In this section, we provide examples of queries that are negation and disjunction. To find all the drives from Boston to Framingham not via I-90 within two hours, we need to make use of the negation operator:

Query 2-5: Find all the drives from Boston to Framingham not via I-90 within two hours

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>$B; \neg I; F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>$B.carid = I.carid = F.carid$</td>
</tr>
<tr>
<td>AND</td>
<td>$B.lat = 'Blat'$ AND $B.long = 'Blong'$</td>
</tr>
<tr>
<td>AND</td>
<td>$I'.roadname = 'I-90'$</td>
</tr>
<tr>
<td>AND</td>
<td>$F.lat = 'Flat'$ AND $F.long = 'Flong'$</td>
</tr>
<tr>
<td>WITHIN</td>
<td>2 hours</td>
</tr>
<tr>
<td>RETURN</td>
<td>$B.carid, F.timestamp - B.timestamp$</td>
</tr>
</tbody>
</table>
To find all the drives from Boston to Framingham via either I-90 or I-93 within two hours, the operator disjunction is used:

Query 2-6: Find all the drives from Boston to Framingham via either I-90 or I-93 within two hours

| PATTERN          | B; (II | I2); F |
|------------------|-----------|
| WHERE            | B.carid = II.carid = I2.carid = F.carid |
| AND              | B.lat = 'Blat' AND B.long = 'Blong' |
| AND              | II.roadname = 'I-90' |
| AND              | I2.roadname = 'I-93' |
| AND              | F.lat = 'Flat' AND F.long = 'Flong' |
| WITHIN           | 2 hours |
| RETURN           | B.carid, F.timestamp - B.timestamp |

2.2.2 Stock Market Monitoring

In this section, we describe several CEP queries for stock market monitoring. The schema of the stock data used is listed in Table 2-2.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Integer</td>
<td>The identification of the trade</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>The company name of this trade</td>
</tr>
<tr>
<td>price</td>
<td>Double</td>
<td>The price at which the trade occurs</td>
</tr>
<tr>
<td>volume</td>
<td>Integer</td>
<td>The volume of the trade</td>
</tr>
<tr>
<td>timestamp</td>
<td>Integer</td>
<td>The time at which the trade occurs. It is transformed to the standard millisecond time value.</td>
</tr>
</tbody>
</table>

Query 2-7 is a simple sequential pattern to find a trade of IBM with price x% higher than the following Google trade, followed by another IBM trade with price y% lower within 10 seconds. Query 2-8 is more complex. It tries to find a group of 5 successive Google
trades instead of just one Google trade in between two IBM trades. Notice that the group number 5 is specified in the query. So in the value constraints clause, the expression “Google^5” is used to constrain the number of successive Google events in the Kleene Closure to 5.

Query 2-7: Find a trade of IBM with price x\% higher than the following Google trade, followed by another IBM trade with price y\% lower within 10 seconds.

\[
\text{PATTERN } IBM1; \text{Google; IBM2}
\]
\[
\text{WHERE } IBM1.name = IBM2.name = 'IBM'
\]
\[
\text{AND } Google.name = 'Google'
\]
\[
\text{AND } IBM1.price > (1 + x\%) \text{Google.price}
\]
\[
\text{AND } IBM2.price < (1 - y\%) \text{Google.price}
\]
\[
\text{WITHIN } 10 \text{ secs}
\]
\[
\text{RETURN } IBM1, Google, IBM2
\]

Query 2-8: Find a trade of IBM with price x\% higher than the average price of the next 5 successive Google trades, followed by another IBM trade with price y\% lower within 10 seconds.

\[
\text{PATTERN } IBM1; \text{Google}^5; IBM2
\]
\[
\text{WHERE } IBM1.name = IBM2.name = 'IBM'
\]
\[
\text{AND } Google.name = 'Google'
\]
\[
\text{AND } IBM1.price > (1 + x\%) \text{avg(Google.price)}
\]
\[
\text{AND } IBM2.price < (1 - y\%) \text{avg(Google.price)}
\]
\[
\text{WITHIN } 10 \text{ secs}
\]
\[
\text{RETURN } IBM1, Google, IBM2
\]

Stock market monitoring is an example where a match-many policy is useful. For example, in Query 2-7, it is hard to say that which Google trade has influence on the relative price fall of the IBM trades; hence the match-many model, which generates all possible combinations, is better because it finds more possible correlations between trades.
2.2.3 Distributed System Debugging

As a third example, we illustrate a sequential query from a distributed debugging system. The behavior of distributed systems with multiple communication components is usually very hard to understand. Situations can be even more difficult when the system has "black-box" components, e.g. commercial software without source code. In such cases, distributed debugging tools can help monitor traces or search for errors. In event-based debugging, models reflect the expected behavior of the system, and thus are used to detect anomalies in the actual behavior of the system. If a system's behavior matches its model, the users have some assurance that the distributed system is functioning correctly.

One interesting feature for distributed system debugging is that it usually has long sequential patterns due to the long code traces it needs to track. For example, in a multi-tier distributed system, such as the one illustrated in Figure 2-3, one possible expected access pattern would be:

1. Q1: Connection request from client to web-server
2. Q2: Authentication check request from web-server to database
3. R2: Authentication response from database to web-server
4. Q3: Application request from web-server to application server
5. Q4: Data request from application server to database
6. R4: Data response from database to application server
7. R3: Application response from application server to web-server
8. R1: Connection response from web-server to client

The model has eight sequential classes, and it can be simply expressed as:

Query 2-9: Find a trace that completes a web service from the client.

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>WHERE</th>
<th>WITHIN</th>
<th>RETURN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1; Q2; R2; Q3; Q4; R4; R3; R1</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

2.2.4 Network Monitoring

Event Pattern Matching is also broadly applicable in network monitoring applications that need to capture various network behaviors. For example, a common application is to count the number of TCP connections from the same IP address during a certain amount of time. Such TCP connections can be detected on edge gateways by examining every packet passing through them.

A TCP connection between Source S and Destination D involves a three-phase handshake. First, Source S sends an IP datagram with the SYN bit set. Then Destination D sends a packet back to acknowledge the SYN packet also with the SYN bit set. Finally Source S
sends a message back to Destination D to acknowledge the second SYN message. Hence we can express the TCP connection pattern as follows:

Query 2-10: Find the IP address that constructs a TCP connection with the desip.

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>P1; P2; P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>P1.ip = P3.ip</td>
</tr>
<tr>
<td>AND</td>
<td>P2.ip = desip</td>
</tr>
<tr>
<td>AND</td>
<td>P1.set = 'SYN'</td>
</tr>
<tr>
<td>AND</td>
<td>P2.set = 'SYN' + 'ACK'</td>
</tr>
<tr>
<td>AND</td>
<td>P3.set = 'ACK'</td>
</tr>
<tr>
<td>WITHIN</td>
<td>100 ms</td>
</tr>
<tr>
<td>RETURN</td>
<td>P1.ip</td>
</tr>
</tbody>
</table>

2.3 Conclusions

In this chapter, we introduced the query language used in ZStream. Based on this specification, we examined four different applications and discussed possible sequential queries in them. Furthermore, we clarify an ambiguity in existing stream processing languages by introducing two classes of consumption policies: match-once and match-many. For the match-once policy, we can further specify the classes of events to match with, such as the most recent event or the oldest event. In Chapter 3 and 4, we will discuss the issues of query evaluation and optimization respectively.
CHAPTER 3: SYSTEM ARCHITECTURE AND OPERATOR EVALUATION

3.1 Introduction

This chapter describes the design of ZStream. To process a pattern query, ZStream first parses and transforms the query expression to 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. The operator associated with each internal node specifies the way to assemble its child events. Primitive events from the sources continuously arrive; ZStream uses a batch-iterator model to decide the best time to start evaluation. Within each assembling batch round (a round really performing combination computation), the tree-plan is evaluated in a bottom-up fashion. The rest of this chapter is organized as follows: we begin by introducing the tree-based representation used for the composite event pattern specification described in Chapter 2; then we describe the buffer structure and the batch-iterator model designed to efficiently process sequential relations; finally we describe the algorithms used to evaluate the operators defined in Chapter 2, and discuss possible optimizations for each of these operators.

3.2 Tree-Based Model

As with query plans in relational databases, ZStream internally represents composite event patterns as tree structures (plans) that are used for evaluation and optimization. In these tree structures, operators in the pattern along with their predicates are represented as
internal nodes and primitive events are represented as leaves. ZStream assumes that primitive events coming from external data sources are inserted into leaf buffers in time order. If disorder is a problem, a reordering operator may be inserted just after the incoming data streams. These primitive events are passed up the tree, forming more complete composite events, and are finally output at the root.

### 3.2.1 Binary Operators

![Diagram of a left-deep tree plan for Query 2-7](image)

**Figure 3-1: A left-deep tree plan for Query 2-7 “find a trade of IBM with price x% higher than the following Google trade, followed by another IBM trade with price y% lower within 10 seconds”. All its SARGable predicates are pushed down to the leaf buffers. Sequential relations and their corresponding constraints on events are captured via the internal Seq nodes.**

Figure 3-1 shows a tree plan for Query 2-7 mentioned in Chapter 2. This query is a sequential pattern query of the form “find one IBM trade ‘IBM’ followed by a Google trade ‘Google’ followed by another IBM trade ‘IBM’”. As mentioned in Chapter 2,
SARGable predicates over just one primitive event class define all the relevant events of the query. Such SARGable predicates can be directly pushed down to the leaf buffers, preventing irrelevant events from being placed into leaf buffers. For example, “Google.name = 'Google'” is a SARGable predicate of Query 2-7. In its tree plan shown in Figure 3-1, the SARGable predicate is pushed down to the front of the Google buffer such that only the events with their names equal to ‘Google’ are placed into the buffer.

In Figure 3-1, sequential relations between IBM and Google events are captured by the Seq nodes. More complicated constraints can be further specified via parameterized predicates associated with each of these Seq nodes. For example, the first sequential relation in Query 2-7 has a parameterized predicate “IBM1.price > (1+x%) Google.price” associated with it; hence this predicate is attached to the Seq1 node. Events coming from the associated sub-trees are assembled in the Seq1 nodes, and parameterized predicates are applied during the assembly procedure. Other binary relations like conjunction and disjunction can be represented in the tree model similarly. However, the expression for Kleene Closure and negation are more complex; we will discuss these two operators in the following two sections.

### 3.2.2 Kleene Closure

Figure 3-2 shows a tree plan for Query 2-8 to find a group of five successive Google trades in between two IBM trades. In the tree plan, Kleene Closure is represented as a K-Seq node. This is because Kleene Closure is only meaningful when applied as a part of a sequential pattern. In the tree plan, the number 5 listed on the right top of the Google buffer indicates that it will group 5 successive Google events together. If the number is not specified, the symbol ‘+’ or ‘*’ can be used instead to group the maximal number of possible successive Google events together.
Generally, a Kleene Closure pattern needs an event class to start and end the closure counting respectively; hence we define the K-Seq operator as a trinary operator (we discuss situations in which the first and the third parameters may be omitted in Section 3.5.4 below). The first operand acts as a start event class, the last operand act as an end event class and the Kleene Closure detection is performed in the middle event class.

![K-Seq Diagram]

**Figure 3-2**: A tree plan for Query 2-8 “find a trade of IBM with price x% higher than the average price of the next 5 successive Google trade, followed by another IBM trade with price y% lower within 10 seconds”. The K-Seq node performs Kleene Closure detection on Google events. The number 5 on the right top of the Google buffer indicates the expected count of the Kleene Closure group is 5. Users may use the symbol ‘+’ or ‘*’ if the number is not specified.

### 3.2.3 Negation

Operator negation represents events that have not yet occurred. It is more complicated in the sense that it is difficult to represent and assemble all events that have not occurred.
The difficulty is that if later the events corresponding to the negation part occur, all previous assembled composite events become invalid. A straightforward way to solve this problem is to add a negation operator as a filter on top of the plan to rule out those composite events that contain events specified in the negation part of the pattern expression [WDR06]. Figure 3-3 illustrates an example negation plan that places negation as a last filter step for Query 2-5 “find all the drives from Boston to Framingham not via I-90”. In this plan, the Boston and Framingham GPS points are first combined in the Seq node. Later, the Neg node filters out those composite events in which negated events occurred.

Figure 3-3: A tree plan for Query 2-5 “find all the drives from Boston to Framingham not via I-90 within two hours”. The negation operator is applied in the final NEG filter.

An obvious problem with the last-filter-step solution is that it generates many
intermediate results where most of them are filtered out in the last step. A better solution is to try to push negation down to avoid unnecessary intermediate results. Since our discussion is focused on sequential applications, we only consider the negation operator that involves sequential patterns, such as “Boston; !I-90; Framingham”. For ease of discussion, we define the following two concepts for this type of sequential negation pattern:

1. Negation event: the event specified in the negation part of the pattern expression
2. Non-negation event: the event specified in the non-negation part of the pattern expression

Now, we can introduce the concept of critical negation property. A typical sequential negation pattern is like “A; !B; C WITHIN tw” where tw stands for the time window during which pattern instances are allowed to occur. For each event instance \( c \) of \( C \), if there exists a corresponding negation event instance \( b \) of \( B \) such that

1. all event instances of \( A \) that occurred after \( b \) but before \( c \) can be matched with \( c \) to form a qualified pattern. And
2. all event instances of \( A \) that occurred before \( b \) are definitely not qualified to match with the instance \( c \).

Then \( b \) is said to have critical negation property for \( c \). The critical negation property for each of the non-negation event instance of \( A \) can be defined similarly. Based on this observation, ZStream introduces a new operator \( N-Seq \) that incorporates negation into the Seq operator by searching for an event having the critical negation property for each input non-negation event instance.

Figure 3-4 shows an example tree plan using N-Seq for the same Query 2-5. The N-Seq operator is applied to find an event instance \( I \) of ‘I-90’ which has the critical negation property for each event instance \( F \) of ‘Framingham’. The output from N-Seq is a
combination of \( F \) and \( I \). In addition, extra time constraints \( B.\text{timestamp} \geq I.\text{timestamp} \) and \( B.\text{timestamp} < F.\text{timestamp} \) are attached to the Seq operator which takes the output of N-Seq as an input. These time constraints can help directly determine the range of event instances of ‘Boston’ that can be combined with the composite events output from N-Seq. Hence, this plan can reduce the size of the intermediate results dramatically. The N-Seq operator can be put into the tree plan as an ordinary binary operator such that its cost estimation and evaluation are performed just like any other operator. In Section 3.5.5, we will discuss the algorithm to evaluate the N-Seq operator in detail.

![Figure 3-4: A tree plan for Query 2-5 “find all the drives from Boston to Framingham not via I-90 within two hours”. The negation operator is incorporated into the Seq operator represented as an N-Seq node.](image)
3.3 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). The buffer structure is important because it is used to join different event streams together and to quickly locate particular time ranges of events. We discuss the buffer design in this section.

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. Each entry in the vector of event pointers 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 in the collection, standing for the start-timestamp and the end-timestamp of the resulting composite event respectively. 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. Figure 3-5 illustrates an example buffer structure. One important feature of the buffer is that its records are stored sorted in the end time order. Since primitive events are inserted into their leaf buffers in time order as discussed in Section 3.2, records in leaf buffers are automatically stored in their end time order. 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 in return generates intermediate results also in end time order. The way the buffer is designed and arranged is to facilitate time range matching for sequential pattern operations.

Other CEP systems (like SASE) also use a buffer based event storage method, but associate each event \( e \) with a pointer connecting to the most recent position (MRP) in its previous buffer \( P \) such that \( e \) can be matched with all event instances in \( P \) prior to its MRP. The MRP for \( e \) is simply the last entry in \( P \) when \( e \) is added to its own buffer. In this way, SASE chains all of the event buffers together sequentially and evaluates the chain from beginning to end. However, this MRP method imposes many constraints on the buffer order, and loses some flexibility for further optimization:

1. For an event \( e \), it is hard to express the MRPs of multiple previous buffers except by keeping a list of MRPs for each of them. This problem makes combining non-successive buffers together complicated. In the cases when non-successive buffers have high selectivity predicates in between, combining non-successive buffers directly can be a better choice.

2. For an event \( e \), it is hard to calculate the MRPs for internal non-leaf buffers, because when \( e \) is added into its buffer, internal results are not in the internal buffers yet. Hence it is hard to make use of materialization using just MRPs.

3. It is hard to support the conjunction operator in the sequential chain. In a typical conjunction pattern “\( A \ & \ B \) WITHIN \( tw \)”, each event instance \( a \) of \( A \) needs to combine with event instances \( b \) of \( B \) before \( a \) as well as event instances \( b \) after \( a \) within \( tw \). However, the sequential chain is connected just in one direction; hence it can not locate the range of event instances of \( B \) before \( a \) quickly if buffer \( B \) is chained after buffer \( A \).

The design of our buffer structure solves all these problems without imposing any
additional computational or memory costs. Timestamps are represented as integers in ZStream. The explicit separation of the start and end time facilitates time comparison between arbitrary buffers in both directions so that ZStream can flexibly choose the order in which it evaluates the buffers and also can support conjunction. In addition, by keeping records in the end-time order in buffers, ZStream does not perform any unnecessary time comparison. Since each operator evaluates its input also in end-time order, events outside the time range can be discarded once the first out-of-range event is found.

3.4 Batch-Iterator Model

This section discusses the problem how ZStream evaluates a tree plan. A naive way is to process each event as soon as it arrives, as is done in many CEP systems [DGH+07] [CKAK94] [GD94] [GJ91]. However, this method is inefficient in that it will produce and store many intermediate results that may not be used before elapse due to the time window constraints. This problem is even worse for long sequential patterns because only after instances of the final event class in the pattern have occurred, can it be decided whether or not a pattern was detected. Thus, if events are combined whenever new events arrive, previously combined intermediate results may stay in the memory for a long time, waiting for the event from the final event class to arrive.

SASE [WDR06] is different. For each sequential pattern, it first creates an NFA representing the pattern. Then, SASE needs at least two steps before it can construct a sequential pattern instance: sequence scan and sequence construction. Sequence scan is used to scan the incoming events and forward them to the NFA. Once the NFA reaches a final state, sequence construction is used to construct a pattern instance by walking backwards through the NFA. Although the pattern construction starts after a final event is
detected, in the sequence scan step, preprocessing such as possible event combinations are still performed, causing similar problems that preprocessed results may stay in the system for a long time.

ZStream uses an alternative solution where it accumulates a batch of events before each round of evaluation. There are two sorts of rounds defined as follows:

- Assembly round: a round that has accumulated enough number of events. ZStream is performing evaluation in such rounds.
- Idle round: a round that has not accumulated enough number of events. ZStream is waiting for more events coming in such rounds.

*Enough* number of events means that there is at least one event instance of the final event class arriving in the buffer, so that its timestamp can be used to calculate the earliest allowed timestamp (EAT) in the system. More specifically, the EAT is calculated by subtracting the time window constraint from the earliest end-timestamp of the last leaf buffer. By using EAT, out-of-date events can be discarded from the system. However, the exact number of enough events is difficult to decide. Hence, ZStream reads in a predefined batch size of events in each idle round, and starts assembly round whenever it detects an event coming from the last event class. The batch-iterator model works as follows:

For each round,

1. Read a batch of primitive events into their leaf buffers with the predefined batch size.
2. If there is no event instance in the last leaf buffer, go back to step 1, otherwise, go to step 3.
3. Calculate the EAT, and a *Next()* call is performed to pass the EAT from the root to leaves.
4. Assemble the primitive events bottom-up, storing the intermediate results in their corresponding node buffers, and remove out-of-date records.

5. Clear buffers when necessary

Step 2 is applied to make sure that only batch rounds having detected event instances from the last event class of the pattern are processed. Otherwise, no output results will be produced, so the algorithm goes back to step 1 to accumulate more events for the next round. Step 3 is performed during the evaluation procedure of each operator; hence does not actually require an extra pass over each buffer.

Figure 3-6 and Figure 3-7 show how the batch-iterator model works. They illustrate a simplified version of Query 2-7 with the pattern “IBM; Google; IBM2 WITHIN 4”. Suppose the incoming events are i1, i2, g3, g4, g5, i6. The letters i and g indicate these events are IBM and Google stock trade events respectively. The numbers indicate their corresponding timestamp. If the batch size is set to 6, then all of the six events will be processed in the first round, with \(EAT = -3\) passed into the \(\text{Next()}\) function call. All the intermediate results generated in the node Seq2 are listed on the upper left in Figure 3-6. If the batch size is set to 1, then there are six rounds, where each round is associated with one incoming event. During the first two rounds, \(i1\) and \(i2\) are inserted into the IBM and IBM2 buffer. Since there are no Google events to be matched with them, no intermediate results are generated. \(i1\) and \(i2\) will be cleared from the buffer IBM2 at the end of round 1 and 2 respectively because they will not be used in later rounds. Rounds 3 through 5 are idle rounds, so they can not start evaluation. But all events are accumulated in the leaf buffers until an IBM event occurs. Figure 3-7 illustrates the snapshot for round 6. The table on the upper left in Figure 3-7 shows all the intermediate results generated in round 6, and no intermediate results are generated in previous rounds. The generated intermediate results in this case are half less than that when batch size was equal to 6.
Figure 3-6: An example to show the 1st round of the batch-iterator model for the pattern “IBM1; Google; IBM2 WITHIN 4” with the batch size equal to 6.

Figure 3-7: An example to show the 6th round of the batch-iterator model for the pattern “IBM1; Google; IBM2 WITHIN 4” with the batch size equal to 1.
The example shown in Figure 3-6 and Figure 3-7 illustrates that using larger batch size will generate more intermediate results which indicates it may be less efficient. This is true because large batch sizes will weaken the impact of time window constraints on the pattern. If batch sizes are big, the time range allowed in each batch round is relatively large: from EAT to the latest event’s end-timestamp in the last leaf buffer. In the example, for batch size 1, the time range allowed in the sixth round is 2 – 6; while for batch size 6, the time range allowed in the first round is -3 – 6. Hence the larger batch size results in some unnecessary intermediate results compared to the situation there are few tuples in the batch and the time range allowed is relative smaller. Our experiments show that ZStream works most efficiently when each assembly round contains one or two event instances from the last event class of the sequential pattern.

Another issue is intermediate results materialization. ZStream makes use of materialized intermediate results to speed up evaluation and intermediate results are removed from the system when they will definitely not be needed by later rounds of evaluation. Since the number of accumulated events in each assembly round is kept as small as possible, ZStream has tried the best to limit the memory usage.

### 3.5 Operator Evaluation

In this section, we describe the algorithms used to evaluate the operators introduced in Chapter 2.

#### 3.5.1 Sequence

Each sequence operator has two operands, representing the fist and second event class in
the sequential pattern. Each of these classes has an event buffer associated with it. The
algorithm used in each batch round is given as follows:

Algorithm 3-1: Sequence Evaluation for Each Batch Round

| INPUT: | right child buffer pointer RBuf, left child buffer pointer LBuf, EAT |
| OUTPUT: | Seq node’s buffer pointer Buf |

1. For each record Rr in RBuf
2. If Rr.start_time < EAT, Rr is removed from RBuf; continue;
3. Loop through the records Lr in LBuf from the beginning.
4. If Lr.start_time < EAT, Lr is removed from LBuf; continue;
5. If Lr and Rr satisfy their value constraints
6. their combination result is inserted into Buf
7. Until Rr.start_time < Lr.end_time
8. Clear RBuf

To make sure the intermediate results are generated in the end time order, the right buffer
is used in the outer loop, and the left buffer is 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 that arriving in later
rounds. Hence 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 are temporally materialized. Hence, step 8 is applied to clear the sequence node’s
right child buffer. As shown in Algorithm 3-1, the way we evaluate the sequence is a
variant of the merge join in which the pointer in the right buffer is moved forwards, while
the one in the left buffer needs to move backwards sometimes. In principle, index based
techniques also can be applied for equality or range predicates, but we believe it is likely
to be too expensive to construct and maintain an index when real time events keep
arriving.
Figure 3-8 shows an example of how the sequence operator assembles events together in one batch round with \( EAT = 0 \). First, \( b3 \) is selected from the right buffer, and combined with both \( a1 \) and \( a2 \); \( a5 \) is not qualified because its end time is bigger than \( b3 \)'s start time. Then \( b4 \) is selected to combine with \( a1 \) and \( a2 \) similarly. At the end of this round, records in the right child buffer are cleared because all possible composite events that can be generated from these records have already been stored in the Seq node’s buffer for further usage. In the next round, new events arrive and are inserted into the end of their corresponding buffers. The EAT is recomputed, and the sequence evaluation algorithm is applied again.

### 3.5.2 Conjunction

Conjunction is similar to sequence except that it does not distinguish between the order of its two operands, as it assembles events in both directions. Since intermediate results still need to be arranged in the end-time order, the algorithm is designed to iteratively pick up
an event $e$ corresponding to the last event (if $e$ is a primitive event) or containing the last event (if $e$ itself is a composite event) of the composite events to be assembled. If $e$ is selected in its end-time order, the intermediate results generated are automatically in end-time order as well. Unlike Seq’s evaluation, for the conjunction operator, neither the right nor the left child buffer can be cleared at the end of each round because records in them may be matched with events that arrive later.

![Diagram of the conjunction operator evaluation](image)

**Figure 3-9:** An example of the conjunction operator evaluation

Figure 3-9 shows an example of the conjunction operator evaluation in one round with $EAT = 0$. Initially, two pointers are set to an initial position\(^2\) in the left child buffer and the right child buffer respectively ($a1$ and $b3$ in this example). Since $a1$ has a smaller end time, $a1$ is picked up. But there is no record in the right buffer with its end time smaller than $a1$’s start time, so $a1$ is skipped because no record can be matched with it; $a2$ is skipped for the same reason. Then $b3$ is selected, and it can be matched with both $a1$ and $a2$; similarly, $b4$ is selected and matched with $a1$ and $a2$. Finally, $a5$ is selected, and it can

---

\(^2\) The initial position is different from the beginning of the buffer due to the batch-iterator model. For each batch round, the initial position stands for the position from which new events coming in this round are stored.
be matched with both \( b3 \) and \( b4 \). It is not correct to remove \( b3 \) and \( b4 \) at the end of this round because there may be new events coming in the next round that can match with \( b3 \) and \( b4 \), \( a6 \) for example. In the next round, new events and assembled results are inserted into the end of corresponding buffers. The algorithm to evaluate the conjunction operator in each batch round is as follows:

**Algorithm 3-2: Conjunction Evaluation for Each Batch Round**

| INPUT: right child buffer pointer RBuf, left child buffer pointer LBuf, EAT |
| OUTPUT: Con node’s buffer pointer Buf |

1. Set \( Lr \) and \( Rr \) to point to the initial record of LBuf and RBuf, respectively
2. While \( Lr \neq \text{end of LBuf} \) or \( Rr \neq \text{end of RBuf} \)
3. If \( Lr.\text{start\_time} < EAT \), remove \( Lr \) from LBuf, \( Lr++ \); continue;
4. If \( Rr.\text{start\_time} < EAT \), remove \( Rr \) from RBuf, \( Rr++ \); continue;
5. If \( Lr.\text{end\_time} > Rr.\text{end\_time} \)
6. \( Pr = Rr; Rr++; Cr = Lr; CBuf = LBuf; \)
7. Else \( Pr = Lr; Lr++; Cr = Rr; CBuf = RBuf; \)
8. For \( Br = \text{beginning of CBuf} \); \( Br \neq Cr \); \( Br++ \)
9. If \( Br \) and \( Pr \) satisfy the value and time constraints
10. \( Br \) and \( Pr \) are combined and inserted into Buf
11. Set \( Lr \) and \( Br \) as the initial record of LBuf and RBuf respectively, for the next round.

### 3.5.3 Disjunction

Disjunction, as specified in Chapter 2, is simply a union of its operands. Hence its evaluation is very straightforward: either operand can be directly inserted into the operator’s buffer if it meets both time and value constraints. The only thing we should be careful of is that the results are also generated in the end time order. ZSstream actually does not materialize results from the disjunction operator because most of the time, they will simply be a copy of the operands. An example of the disjunction evaluation is shown in Figure 3-10. The results are generated by simply merging the left buffer and the right
buffer according to their end time. The algorithm to evaluate disjunction in each batch round is as follows:

Algorithm 3-3: Disjunction Evaluation for Each Batch Round

| INPUT: | right child buffer pointer RBuf, left child buffer pointer LBuf, EAT |
| OUTPUT: | Dis node's buffer pointer Buf |

1. Set Lr and Rr to point to the beginning of LBuf and RBuf respectively
2. While Lr != end of LBuf or Rr != end of RBuf
3. If Lr.start_time < EAT, remove Lr from LBuf; Lr++; continue;
4. If Rr.start_time < EAT, remove Rr from RBuf, Rr++; continue;
5. If Lr.end_time > Rr.end_time, Pr = Rr; Rr++; Else Pr = Lr; Lr++
6. If Pr satisfies the value constraints, insert Pr to Buf
7. clear LBuf and RBuf

Figure 3-10: An example of the Disjunction operator evaluation.

3.5.4 Kleene Closure

Kleene Closure is mapped to a physical operator $K-Seq$ as shown in Section 3.2.2
internally in ZStream. Unlike the previous three operators, K-Seq is a trinary operator that processes three buffers: a start buffer, a closure (middle) buffer and an end buffer. After choosing a start point and an end point from the start buffer and the end buffer, it looks for matches in the middle buffer. Remember that in Kleene Closure, we can set a closure count on the middle buffer to specify the expected number of the closure events per group. 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. Then in each time step, the window is moved forward one step, generating several results for each pair. Again, the intermediate results need to be arranged in the end-time order; hence the end buffer is put in the outer loop of the evaluation algorithm.

Figure 3-11 illustrates an example of the evaluation of K-Seq. The buffer on the upper left corner shows the results when the closure count is not specified; the one on the upper right corner shows the results when the closure count is 2.

Figure 3-11: An example of the K-Seq operator evaluation for the pattern \((A; B^*; C)\). The buffer on the upper left corner shows the results when the closure count is not specified; the one on the upper right corner shows the results when the closure count is 2.
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. When the closure number is not specified, \(c6\) 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. After that, 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 K-Seq in each batch round is shown in Algorithm 3-4.

\[
\text{Next}(0)
\]

Figure 3-12: An example of the K-Seq operator evaluation for the pattern of \(A^4\) with the closure number equal to 4.

With the K-Seq operator, we also allow the start operand or the end operand or both to be omitted. The reason to have the start and the end operand is to fix the start and the end points for the Kleene Closure detection. However, if only one of the points is specified, the other one can be found by using time window constraints or the closure count. If neither start or end event is specified, every event in the closure buffer can potentially be a start point. Then, the end point can be derived using the time window constraints or the closure count. Figure 3-12 illustrates an example of evaluating the pattern \(A^4\).
Algorithm 3-4: K-Seq Evaluation for Each Batch Round

INPUT: start buffer pointer SBuf, middle buffer pointer MBuf, end buffer pointer EBuf, closure count cc, EAT
OUTPUT: result buffer pointer Buf

1. For each record Er in EBuf
2. If Er.start_time < EAT, remove Er from EBuf; Er++; continue;
3. Loop through the records Sr in SBuf from the beginning
4. If Sr.start_time < EAT, remove Sr from SBuf; Sr++; continue;
5. For each record Cr in MBuf from the beginning
6. If Cr.start_time < EAT or Cr.start_time <= Sr.end_time, remove Cr from MBuf
7. If cc is not specified
8. Group all the Crs where Cr.start_time > Sr.end_time
   and Cr.end_time < Er.start_time
   and Cr satisfies value constraints
9. Insert the composite event into Buf
10. Elseif cc is specified
11. Loop through Cr beginning with records where Cr.start_time > Sr.end_time
12. Group beginning from Cr where the record Gr in the group having
    Gr.end_time < Er.start_time
    and Gr follows corresponding value constraints
13. Until the number of instances in the group reaches the closure count cc
    or Gr.end_time >= Er.start_time
14. Insert the composite event into Buf if any
15. Cr++
16. Until Gr.end_time >= Er.start_time
17. Until Sr.end_time >= Er.start_time

3.5.5 Negation

As we mentioned in Section 3.2.3, ZStream includes a way to incorporate negation into the sequence operator to form a new operator N-Seq. N-Seq is a binary operator having a left operand A and a right operand B. It is defined as follows:

- Only event instances a of A that occur before b of B can be combined with b.
Exactly one of the operands must contain a negation event class. If A is a negation event class, the operator is called a left N-Seq. It is called a right N-Seq otherwise.

For example, for a sequential negation pattern (A; !B; C), the N-Seq operator that takes A and B as its inputs (A; !B) is a right N-Seq, and the one that takes B and C as its inputs (!B; C) is a left N-Seq.

We distinguish between left N-Seq and right N-Seq because their evaluation algorithms and extra time constraints generation are different. A sequential pattern like "!B WITHIN tw" does not make any sense if tw is specified as a relative time window (e.g. within 2 hours). First, the semantic to output nothing is vague. Second, it is difficult to set start point to begin timing. For example, if a pattern "!B" is satisfied during the time period t0 to t0+tw, then "!B" is also satisfied in any time period t0+ At to t0+ At+tw, where At can be arbitrary small time as long as the next B event instance occurs after t0+ At+tw. Hence there is at least one event class occurs before or after the negation event class "!B; A" or "A; !B". Now, we can complete the definition of critical negation event. A critical negation event is supposed to have critical negation property\(^3\) for its corresponding non-negation event.

In each N-Seq operator where A is the left operand and B is the right one, and tw is the time window allowed,

- If it is a left N-Seq (!A; B) and A is a primitive event class, for each b of B, the critical negation event is defined as the latest event instance a of A that occurs before b. If b.end_time - a.timestamp > tw no instance of A is found for b, b's critical negation event is set to NULL.
- If it is a right N-Seq (A; !B) and B is a primitive event class, for each a of A, the

---

\(^3\) Defined in Section 3.2.3
critical negation event is defined as the earliest event instance $b$ of $B$ that occurs after $a$. If $b.timestamp - a.start_time > tw$ or no instance of $B$ is found for $a$, $a$'s critical negation event is set to NULL.

The definition above only allows negation event class to be a primitive one. Semantics of composite negation is very complicated and out of scope in this thesis. Given these definition, we can now discuss the evaluation of N-Seq. Let's begin with a simple case where no parameterized predicates (i.e. predicates involving more than one event classes) are involved with the negation event class.

3.5.5.1 Non Parameterized Predicates Involved with Negation (NPPN)

![Diagram](Figure 3-13: An example sequential negation evaluation for the pattern “A; !B; C WITHIN tw” using a left N-Seq. The batch size is 6.)

$NPPN$ is a sequential negation pattern where the negation event class does not involve
any parameterized predicates between non-negation event classes. A typical example of an NPPN pattern is "A; !B; C WITHIN tw". Figure 3-13 illustrates a query plan for evaluating this pattern using a left N-Seq. The node N-Seq is a left N-Seq operator because its left operand B is a negation event class. The critical negation event for c5 is b3, which is recorded as "b3, c5" in the buffer of N-Seq. Extra time predicates "A.end_time < C.start_time" and "A.end_time > B.timestamp" are pushed into the Seq operator. These indicate that, because of the negation, only event instances of A in time range (3, 5) should be considered (e.g. a4). Finally, the composite result "a4, c5" is returned.

Figure 3-14 illustrates how time constraints imposed by the left N-Seq are generated for a sequential negation pattern of the form "A; !B; C". Suppose c is an instance of C and b is c’s critical negation event. The time range in which an event instance a of A will be combined with c is after b.timestamp and before c.start_time, as marked on the time line.

Figure 3-14: An illustration for how to decide the extra time constraints. For the negation sequential pattern "A; !B; C", c is an instance of C and b is c’s critical negation event. The double arrow marked on the time line is the range that event instances a of A allowed to be combined with c.

Figure 3-15 illustrates another query plan for evaluating the pattern "A; !B; C WITHIN tw" with the same input but uses a right N-Seq. The evaluation process is similar to that using a left N-Seq. An important difference in this case is the time constraints "C.start_time > A.end_time" and "C.end_time < B.end_time" created by the right N-Seq.
Figure 3-15: An example sequential negation evaluation for the pattern “A; !B; C WITHIN tw” using a right N-Seq. The batch size is 6.

Algorithm 3-5: Left N-Seq Evaluation for Each Batch Round

INPUT: left buffer pointer LBuf, right buffer pointer RBuf, and EAT
OUTPUT: result buffer pointer Buf

1. For each record Rr in RBuf
2. If Rr.start_time < EAT, Rr is removed from RBuf; continue;
3. Loop through the records Lr in LBuf from the end.
4. If Lr.end_time < Rr.start_time
5. If Lr.start_time < EAT, insert (NULL, Rr) to Buf, Lr is removed from LBuf
6. Else insert (Lr, Rr) to Buf
7. Break;
8. Elself Lr.start_time < EAT,
9. insert (NULL, Rr) to Buf, Lr is removed from LBuf, break
10. Until to the beginning of LBuf
11. If no Lr is found, insert (NULL, Rr) to Buf
12. Clear RBuf
The algorithm to evaluate the left N-Seq operator for NPPN patterns is shown above. It does not include the extra time predicates generated by the N-Seq operator because these predicates are determined when the query plan is generated. The algorithm to evaluate the right N-Seq can be derived similarly.

3.5.5.2 Parameterized Predicates Involved with Negation (PPN)

*PPN* is a sequential negation patterns where the negation event class involves parameterized predicates between non-negation event classes. If the negation event class’s parameterized predicates are all applied with one of the non-negation event classes, the N-Seq model designed for NPPN patterns is also suitable because the critical negation events still have the critical negation property for their corresponding non-negation events. For example, Figure 3-4 illustrates a left N-Seq tree plan for Query 2-5 to find all the drives from Boston to Framingham that did not use I-90 within two hours. The negative event class I-90 involves a parameterized predicate “I.carid = E.carid”. If the negation event class has parameterized predicates applied to more than one non-negation event classes, the critical negation property does not hold any more and needs amends. This problem will be addressed in our future work. The current version of ZStream evaluates such negation queries at the top of the plan as mentioned in Section 3.2.3.

3.6 Conclusions

In this chapter, we introduced the architecture of ZStream, including internal tree plan representations, buffer structures and the batch-iterator model. All the design decisions are made to process and evaluate the sequential patterns efficiently. We then proposed algorithms to evaluate each operator introduced in Chapter 2 in detail. In particular, we
described a novel way to evaluate sequential negation patterns, such that the negation operator can be efficiently incorporated into sequence evaluation.
CHAPTER 4: COST MODEL AND OPTIMIZATION

This chapter introduces a cost model to estimate the cost of each of the operators described in the previous chapters. Using these costs, the cost of a given query plan tree can be estimated by adding up the cost of all its composite operators. After describing the cost function for a query plan, we describe the mechanism used to search for the optimal plan from a variety of candidate plans. Our algorithm first searches for all pattern expressions that are semantically equivalent to the specified composite event pattern using a number of algebraic rewrites. For each of these semantically equivalent patterns, we use a dynamic programming algorithm to automatically choose the best evaluation order for its operators. Finally, we choose the plan with the minimal cost from amongst these various alternatives.

4.1 Cost Model

In traditional databases, the estimated cost of a query plan consists of two parts: the I/O cost and CPU cost [SAC+79]. I/O cost is significant in traditional databases because their tables are stored on disk. CPU cost is caused by computation performed in various operators — a typical measure is the number of function calls to access and modify tuples [SAC+79].

In real time CEP systems like ZStream, I/O cost is not considered because all external primitive events are memory resident. Once they are discarded from the memory, the outdated events will not be reaccessed. Hence the cost of CEP systems is mainly determined by the CPU cost of combining and processing events. ZStream measures the
CPU cost from two aspects: the cost to access the input data and the cost to generate the output data. These two parts of costs are estimated according to the number of events accessed and combined in the time constraint window respectively. Hence, the cost model is designed as follows:

\[
C = C_i + pC_o
\]

The cost of an operator consists in two parts: the cost of accessing the input data \( C_i \), the cost to generate the output data \( C_o \). \( p \) is a predefined parameter, which determines the weight between \( C_i \) and \( C_o \). \( p \)'s default value is 1.

In Formula 4-1, \( C_i \) 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 being touched, the weight \( p \) can be set to 1 by default. Also, according to our experiments run on ZStream, the cost model can reflect the real system performance accurately when \( p = 1 \). Hence in this thesis, we will simply use \( C = C_i + C_o \) to estimate the total cost. To simplify the presentation of this cost formula in detail, we make the following definitions:

- \( R_E \): rate of primitive events coming from the event class \( E \). This is the cardinality of \( E \) per unit time.
- \( TWp \): time window specified in a given query pattern \( p \).
- \( P_E \): Selectivity of all SARGable predicates for event class \( E \). It is the product of selectivity of each SARGable predicate of \( E \).
- \( CARDE \): the average cardinality of the relevant events\(^4\) of event class \( E \) which are active within the time constraint \( TWp \). This can be estimated as \( R_E \cdot TWp \cdot P_E \).

\(^4\) As defined in Section 2.2.1.1, relevant events are events that belong to any event class in the pattern and also pass all the SARGable predicates associated with that event class.
- $Pt_{E_1, E_2}$: selectivity of the implicit time predicate between event class $E_1$ and $E_2$, with $E_1.end\_time < E_2.start\_time$. In the absence of more accurate statistics, it is assumed to have value 1/2.

- $P_{E_1, E_2}$: Selectivity of all parameterized predicates between event class $E_1$ and $E_2$. It is the product of selectivity of each parameterized predicate between $E_1$ and $E_2$. If $E_1$ and $E_2$ do not have predicates, it is set to 1.

- $C_{io}$: the cost of accessing input data of the operator $O$.

- $C_{oo}$: the cost of accessing input data of the operator $O$.

- $CARD_O$: average cardinality of the composite events output by the 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 $CARD_O (C_{oo})$.

$R_E$ determines the number of events per unit time, and $R_E * P_E$ estimates the number of relevant events per unit time. Hence we can use $R_E * TWp * P_E$ to estimate the average cardinality $CARD_E$ of all relevant event instances of $E$ that are active within the time constraint $TWp$. Given the left-deep plan in Figure 4-1 for example, $CARD$ of $IBM$ events is:

$$CARD_{IBM1} = CARD_{IBM2} = R_{STOCK} * P_{IBM} * (10 \, \text{sec})$$

Operators that involve sequentiality such as sequence, Kleene Closure and sequential negation have implicit time predicates. $Pt_{E_1, E_2}$ is used to measure the selectivity of such predicates. For example, the pattern "$E_1; E_2$" implicitly includes a time predicate "$E_1.end\_time < E_2.start\_time$", which indicates that only event instances $e1$ of $E1$ that occur before event instances $e2$ of $E2$ can be combined with $e2$. However, $Pt_{E_1, E_2}$ does not apply to the cost formula for conjunction and disjunction as they are independent of
sequentiality. $P_{EI, E2}$ is similar to $Pt_{EI, E2}$ except that it includes the selectivity of all parameterized predicates between $E1$ and $E2$.

As in traditional databases, we assume that all these statistics ($R_E, P_E, Pt_{EI, E2}$ and $P_{EI, E2}$) can be estimated from the historical data. Table 4-1 lists the values of these parameters that are used when historical statistics are not available. When $R_E$ is not available, its default value should be chosen such that “$CARDE > 1$”. This constraint is made to avoid the case where “$CARDE < 1$” — implying that the more buffers are combined, the less combination work need to be done. Once we have this cardinality constraint, it is
sufficient to estimate and compare costs of alternative query plans by using relative event rates between event classes. Table 4-1 sets the selectivity of equality predicates (for both $P_E$ and $P_{E1,E2}$) as 1/10 when historical statistics are not available. In fact, this selectivity can be set as arbitrary number but should be smaller than that of a range predicate (because equality predicates are less selective than range predicates) and must be larger than zero.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{E1} : R_{E2}$</td>
<td>Relative event rate between $E1$ and $E2$</td>
<td>1:1</td>
</tr>
<tr>
<td>$P_{E}$</td>
<td>Equality predicate $(E.\text{attri} = \text{val})$ The number of distinct attri values is $n$</td>
<td>$1/n$</td>
</tr>
<tr>
<td></td>
<td>$n$ is unknown</td>
<td></td>
</tr>
<tr>
<td></td>
<td>range predicate $(E.\text{attri} &gt; \text{val})$ The high value is $v_{\text{max}}$, and the low value is $v_{\text{min}}$</td>
<td>$(v_{\text{max}}-\text{val}) / (v_{\text{max}}-v_{\text{min}})$</td>
</tr>
<tr>
<td></td>
<td>$v_{\text{max}}$ and $v_{\text{min}}$ are unknown</td>
<td>$1/2$</td>
</tr>
<tr>
<td>$P_{E_{E1,E2}}$</td>
<td>Time predicate between $E1$ and $E2$ for Seq and K-Seq operator. It is not applicable for conjunction and disjunction</td>
<td>$1/2$</td>
</tr>
<tr>
<td>$P_{E1:E2}$</td>
<td>Equality predicate $(E1.\text{attri1} = E2.\text{attri2})$ Max(The number of distinct attri1 values, the number of distinct attri2 values) = $n$</td>
<td>$1/n$</td>
</tr>
<tr>
<td></td>
<td>$n$ is unknown</td>
<td>$1/10$</td>
</tr>
<tr>
<td>$P_{E1:E2}$</td>
<td>Range predicate $(E1.\text{attri1} &gt; E2.\text{attri2})$ NONE</td>
<td>$1/2$</td>
</tr>
</tbody>
</table>

### 4.1.1 Operator Cost

Table 4-2 summarizes the input cost formulas ($C_{io}$) and output cost formula ($C_{oO}$) for each individual operator. The input cost $C_{io}$ is expressed in terms of the number of operand combinations performed; and the output cost $C_{oO}$ is measured as the number of composite events generated from the operator, which is exactly the $\text{CARD}_O$. The total cost is of an individual operator is the sum of its input cost and output cost as indicated in Formula 4-1.
Table 4-2: Input Cost Formula and Output Cost Formula for Individual Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Input Cost $C_i$ (CARD)</th>
<th>Output Cost $C_o$ (CARD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Scan A</td>
<td>It is the cost to scan the leaf buffer for each primitive event class A.</td>
<td>$R_A * TWP$</td>
<td>$R_A * TWP * P_A$</td>
</tr>
<tr>
<td>Sequence (A ; B)</td>
<td>A and B are two input event classes. The cost is expressed as the number of operand combinations being tried. $P_{t_A, B}$ captures the fact that the Sequence operator does not try to assemble any event a of A with event b of B where b occurs before a.</td>
<td>$CARD_A$ * $CARD_B$ * $Pt_{A, B}$</td>
<td>$CARD_A$ * $CARD_B$ * $P_{A, B}$</td>
</tr>
<tr>
<td>Conjunction (A &amp; B)</td>
<td>A and B are two input event classes. Unlike Sequence, Conjunction can combine event a of A with any b of B within the time window constraint.</td>
<td>$CARD_A$ * $CARD_B$</td>
<td>$CARD_A$ * $CARD_B$ * $P_{A, B}$</td>
</tr>
<tr>
<td>Disjunction (A</td>
<td>B)</td>
<td>A and B are two input event classes. Since Disjunction is just a merge of its two operands, its cost is just the cost of fetching each event from its inputs.</td>
<td>$CARD_A$ + $CARD_B$</td>
</tr>
<tr>
<td>Kleene Closure (A; B'; C)</td>
<td>A and C are the start and the end operand of Kleene Closure respectively. B is the closure operand. 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 not specified N = 1 if operand B is missing.</td>
<td>$CARD_A$ * $CARD_C$ * $Pt_{A, C}$ * N</td>
<td>$CARD_A$ * $CARD_C$ * $P_{A, B}$ * $P_{B, C}$</td>
</tr>
<tr>
<td>Negation (A; !B; C)</td>
<td>Negation pushed down, using left N-Seq, expressed as: Seq(A, N-Seq(B, C))</td>
<td>$CARD_A$ + $CARD_A$ * $CARD_C$ * $Pt_{A, C}$</td>
<td>$CARD_A$ + $CARD_A$ * $CARD_C$ * $Pt_{A, C}$ * $Pt_{B, C}$</td>
</tr>
<tr>
<td></td>
<td>Negation pushed down, using right N-Seq, expressed as: Seq(N-Seq(A, B), C)</td>
<td>$CARD_A$ + $CARD_A$</td>
<td>$CARD_A$ + $CARD_A$ * $CARD_C$ * $Pt_{A, C}$ * $Pt_{B, C}$</td>
</tr>
</tbody>
</table>

The cost of buffer scanning includes the costs of applying all SARGable predicates on an event class to filter out all irrelevant events. Although it is not defined as an operator in this thesis, to unify the cost estimation, we also include its cost formula in the formula
The implicit time selectivity $P_{tA,B}$ is attached to sequence’s cost formula because the buffer structure designed in ZStream automatically filters out all event instances $a$ of $A$ with $a.end_time >= b.start_time$ for each event instance $b$ of $B$. The evaluation of conjunction and disjunction is independent of the order of their operands; hence time predicates do not need to be included in their cost formulas. Another important issue for disjunction is that it is not valid to apply parameterized predicates over its operands because either of the two operands may occur without the other.

The cost for Kleene Closure is more complicated. In the simplest case, where the closure operand is missing, Kleene Closure is simply a special case of the sequence operator; hence, its cost is exactly the same as that of sequence. In the case where the closure operand is not missing, 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 visited event instances $N$ from the middle operand $B$ can be estimated as the number of events from $B$ that can match with each start-end pair. So $N = CARD_B * P_{tA,B} * P_{tB,C}$, where $A$ and $C$ stand for the start and end event class respectively. If the closure count $cnt$ is specified, then for any start-end pair, each event instance from $B$ occurs in between this pair will be output $cnt$ times. Hence $N = R_B * P_{tA,B} * P_{tB,C} * cnt$.

As mentioned in Chapter 3.1.3, we have two ways to evaluate negation. One is to put an NEG operator on top to filter out results having composed negation events occurred. The other is to use an N-Seq operator to push the negation into the tree. 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 normally. 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. \( Z \)Stream can find such a \( b \) by refer to the corresponding critical negation event, hence does not need to scan the entire buffer \( B \). The cost of the second way \((\text{Seq}(A, \text{N-Seq}(B,C)))\) also contains two parts: the cost to evaluate the left \( \text{N-Seq} \) and that to evaluate the \( \text{Seq} \) operator. The input cost of the left \( \text{N-Seq} \) is \( \text{CARD}_c \) and not related to \( \text{CARD}_b \) for the same reason that \( Z \)Stream can find each \( c \)'s critical negation event (which is just the latest event in \( B \) before \( c \)) directly, without searching for the entire \( B \) buffer. Then the cost of the \( \text{Seq} \) operator can be estimated as normal.

The cost formulas shown in Table 4-2 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. For example, the cost of the sequence operator where its first operand is an operator \( O \) can be estimated as \( \text{CARD}_O \ast \text{CARD}_B \ast P_{to, B} \), where \( \text{CARD}_O \) stands for the cardinality of the composite events generated by \( O \).

### 4.1.2 Tree Plan Cost

The previous discussion showed how to estimate the cost of individual operators. In this section, we will focus on cost estimation for an entire query plan. Intuitively, the cost of an entire tree plan can be estimated by adding up all the costs of its composite operators, as shown below.

**Formula 4-2 Tree Plan Cost Formula**: For any plan that composed of operators \( O_1, O_2, \ldots, O_n \), the cost of the plan is:

\[
\text{TotalCost} = \sum_{i=1}^{n} C_{oi}
\]
Figure 4-1 shows two equivalent query plans for Query 2-7: a left-deep plan and a right-deep plan; we can estimate their costs for the entire trees as follows:

**Cost Estimation for the left-deep plan of Query 2-7**

<table>
<thead>
<tr>
<th>Cost Component</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{STOCK}$</td>
<td>$R_{STOCK} \times TWP$</td>
</tr>
<tr>
<td>$CARD_{IBM1}$</td>
<td>$CARD_{IBM2} = R_{STOCK} \times TWP \times P_{IBM}$</td>
</tr>
<tr>
<td>$CARD_{Google}$</td>
<td>$R_{STOCK} \times TWP \times P_{Google}$</td>
</tr>
<tr>
<td>$C_{Seq1}$</td>
<td>$CARD_{IBM1} \times CARD_{Google} \times P_{IBM1, Google}$</td>
</tr>
<tr>
<td>$CARD_{Seq1}$</td>
<td>$CARD_{IBM1} \times CARD_{Google} \times P_{IBM1, Google}$</td>
</tr>
<tr>
<td>$C_{Seq2}$</td>
<td>$CARD_{Seq1} \times CARD_{IBM2} \times P_{Google, IBM2}$</td>
</tr>
<tr>
<td>$CARD_{Seq2}$</td>
<td>$CARD_{Seq1} \times CARD_{IBM2} \times P_{Google, IBM2} \times P_{Google, IBM2}$</td>
</tr>
<tr>
<td>$C_{Tree}$</td>
<td>$C_{Seq1} + CARD_{Seq1} + C_{Seq2} + CARD_{Seq2} + CARD_{IBM1} + CARD_{IBM2} + CARD_{Google} + C_{STOCK}$</td>
</tr>
</tbody>
</table>

**Cost Estimation for the right-deep plan of Query 2-7**

<table>
<thead>
<tr>
<th>Cost Component</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{STOCK}$</td>
<td>$R_{STOCK} \times TWP$</td>
</tr>
<tr>
<td>$CARD_{IBM1}$</td>
<td>$CARD_{IBM2} = R_{STOCK} \times TWP \times P_{IBM}$</td>
</tr>
<tr>
<td>$CARD_{Google}$</td>
<td>$R_{STOCK} \times TWP \times P_{Google}$</td>
</tr>
<tr>
<td>$C_{Seq1}$</td>
<td>$CARD_{Google} \times CARD_{IBM2} \times P_{Google, IBM2}$</td>
</tr>
<tr>
<td>$CARD_{Seq1}$</td>
<td>$CARD_{IBM1} \times CARD_{Seq2} \times P_{IBM1, Google}$</td>
</tr>
<tr>
<td>$C_{Seq2}$</td>
<td>$CARD_{IBM1} \times CARD_{Seq2} \times P_{IBM1, Google}$</td>
</tr>
<tr>
<td>$CARD_{Seq2}$</td>
<td>$CARD_{Seq2} \times CARD_{IBM2} \times P_{Google, IBM2} \times P_{Google, IBM2}$</td>
</tr>
<tr>
<td>$C_{Tree}$</td>
<td>$C_{Seq1} + CARD_{Seq1} + C_{Seq2} + CARD_{Seq2} + CARD_{IBM1} + CARD_{IBM2} + CARD_{Google} + C_{STOCK}$</td>
</tr>
</tbody>
</table>

Appendix A illustrates such cost estimation with real statistics numbers.

### 4.2 Equivalent Query Plans

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 two types of equivalence: rule-based transformations and operator reorderings.

### 4.2.1 Rule-Based Transformations

It is possible to express a given pattern in many different ways. For example, the following two expressions are semantically identical:

- **Expression 1:** “\( A; (\neg B \land \neg C); D \)”
- **Expression 2:** “\( A; (B \mid C); D \)”

Their expression complexity and evaluation cost, however, are substantially different from each other. Table 4-3 lists all the equivalent transformation supported by ZStream:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator Applied on</th>
<th>Transition Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence (;)</td>
<td>Sequence (;)</td>
<td>( A; (B; C) = (A; B); C )</td>
</tr>
<tr>
<td></td>
<td>Conjunction (&amp;)</td>
<td>( A; (B \land C) = (A; B) \land (A; C) )</td>
</tr>
<tr>
<td></td>
<td>Disjunction (!)</td>
<td>( A; (B \mid C) = (A; B) \mid (A; C) )</td>
</tr>
<tr>
<td>Conjunction (&amp;)</td>
<td>Sequence (;)</td>
<td>( A \land (B; C) = (A \land B); (A \land C) )</td>
</tr>
<tr>
<td></td>
<td>Conjunction (&amp;)</td>
<td>( A \land (B \land C) = (A \land B) \land C )</td>
</tr>
<tr>
<td></td>
<td>Disjunction (!)</td>
<td>( A \land (B \mid C) = (A \land B) \mid (A \land C) )</td>
</tr>
<tr>
<td></td>
<td>Sequence (;), Disjunction (!)</td>
<td>( A \land B = (A; B) \mid (B; A) )</td>
</tr>
<tr>
<td>Disjunction (!)</td>
<td>Disjunction (!)</td>
<td>( A \land (B; C) = (A \land B); (A \land C) )</td>
</tr>
<tr>
<td>Kleene Closure (*)</td>
<td>Kleene Closure (*)</td>
<td>( (A^*) = A )</td>
</tr>
<tr>
<td>Negation (!)</td>
<td>Sequence (;)</td>
<td>( \neg (A; B) = (\neg A; B) \land (A; \neg B) )</td>
</tr>
<tr>
<td></td>
<td>Conjunction (&amp;)</td>
<td>( \neg (A \land B) = (\neg A) \land (\neg B) )</td>
</tr>
<tr>
<td></td>
<td>Disjunction (!)</td>
<td>( \neg (A \land B) = (\neg A) \land (\neg B) )</td>
</tr>
<tr>
<td></td>
<td>Negation (!)</td>
<td>( \neg (\neg A) = A )</td>
</tr>
</tbody>
</table>
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, and then selecting the one with the minimal cost. Instead, we narrow down the equivalent transition space by always trying to simplify the pattern expression. That is, a transition is taken only when the target expression
1. has a smaller number of operators and, if the expression has the same number of operators, then
2. includes operators with lower cost.

Based on the transition rule given in Table 4-3, plans with smaller number of operators will also include smaller number of event classes to be combined, thus are more likely to result in fewer intermediate composite events being generated. If the alternative plan has the same number of operators, but includes lower cost operators, it would also be performed. Based on the cost formula given in Table 4-2, we observe that the cost of operators satisfies the inequality shown in Formula 4-3.

<table>
<thead>
<tr>
<th>Formula 4-3 Operator Cost Inequality Formula: If all the other conditions are the same, the operators of Sequence, Conjunction and Disjunction satisfy the following correlation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{\text{Dis}} &lt; C_{\text{Seq}} &lt; C_{\text{Con}}$</td>
</tr>
</tbody>
</table>

Returning to the two expressions given at the beginning of this section, the optimizer will choose to transition from \textit{Expression 1} to \textit{Expression 2} because \textit{Expression 2} has a smaller number of operators and the cost of the Disjunction operator is smaller than that of the Conjunction.
4.2.2 Operator Reordering

The other source of equivalent query plans is operator orders. As indicated in Table 4-3, when the pattern expression only includes sequence, conjunction and disjunction operators, the order of the operators does not affect the semantics of the expression. Although such alternative orderings are not simpler, because they have exactly the same operators and event classes, the evaluation costs of each alternative expression can vary. The two query plans shown in Figure 4-1 are an example of equivalent alternatives for Query 2-7. The only difference between these two tree plans is the order in which the two sequence operators are performed. As we showed in Section 4.1.2, the costs of these two plans are different. Deciding which is cheaper is a simple matter of substituting statistics for these symbols to estimate the cost. However, we still need an algorithm to search through the possible orderings to find the optimal one. This is especially important for long sequential patterns where the optimal order is not that obvious.

4.2.2.1 Optimal Operator Order Algorithm

In this section, we describe an algorithm to search for the optimal physical plan for a sequential pattern. We first observe that the problem of finding the optimal order has an optimal substructure, suggesting it is amenable to dynamic programming. To see this, consider Figure 4-2. For a given query pattern, 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. This is true because otherwise it would be possible to find a sub tree plan $T_i'$ with lower cost than $T_i$. 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. In Figure 4-2, the root $Seq1$ divides the sequential pattern “$A_1; A_2; ...; A_n$” into two sub patterns “$A_1; A_2; ...; A_i$” and “$A_{i+1}; A_{i+2}; ...; A_n$”. If $T$ is optimal, the two sub trees $T1$ and $T2$ will also
be optimal for the two sub patterns.

Figure 4-2: An example tree plan $T$ for the sequential pattern "$A_1; A_2; \ldots; A_n$". The root $Seq_1$ divides the sequential pattern to two sub patterns "$A_1; A_2; \ldots; A_i$" and "$A_{i+1}; A_{i+2}; \ldots; A_n$". If $T$ is optimal, the two sub trees $T_1$ and $T_2$ will also be optimal for the two sub patterns.

Algorithm 4-1: Searching for the Optimal Operator Order

INPUT: the number of event classes in the pattern $N$

OUTPUT: root buffer $ROOT$ that records the roots of all of the optimal sub trees

1. Initialize $Min[s][i]$, $ROOT[s][i]$ and $CARD[s][i]$, where $s$ is the current list size and $i$ is used to identify different lists for the same list size. $Min$ records the minimal costs, $ROOT$ records the roots of the optimal sub trees, and $CARD$ records the resulting composite event cardinality for the corresponding root operator. $CARD[1][i]$ records the cardinality of $i$-th event class.
2. For list size $s = 2$; $s <= n$; $s++$
3. For list index $i = 1$; $i <= n-s + 1$; $i++$
4. For root position in each set $r = i+1$; $r < i+s$; $r++$
5. $opcost = \text{calc\_inputcost}(CARD[r-i][i], CARD[s-r+i][r], r)$;
6. $Cost = Min[r-i][i] + Min[s-r+i][r] + opcost$;
7. If $Min[s][i] > Cost$
8. $Min[s][i] = Cost$; $ROOT[s][i] = r$; $CARD[s][i] = \text{calc\_CARD}(opcost, r)$
Based on the optimal substructure observation, we can search for an optimal tree plan by combining all 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 described as Algorithm 4-1.

Algorithm 4-1 begins by calculating the optimal sub plans from the event lists of size 2. In this case, there is only one operator connecting the event classes; hence its operator order is automatically optimal. In the outer most loop (step 2), the algorithm increases the event set size by 1 each time. The second loop (step 3) is goes over all possible event sets of the current event set size. The third loop (step 4) records the minimal cost plan found so far by searching for all possible optimal sub trees. Once an optimal plan is found for
the sub pattern, the root of the tree plan is recorded in the **ROOT** matrix. In the end, the optimal tree plan can be reconstructed by walking in reverse through each of the sub tree. Algorithm 4-1 has two function calls: `calc_inputcost()` and `calc_CARD()`. They are used to estimate the input cost and output cost of an operator as stated in Section 4.1.

Figure 4-3 illustrates an entire procedure to construct an optimal plan when the input event list size is equal to 4. During the last run where the only event list is the entire pattern (1, 2, 3, 4), the algorithm tries all possible combination of its sub lists: 1 and (1, 2, 3), (1, 2) and (3, 4); (1, 2, 3) and 4. Root information is recorded above each event sub list. The corresponding optimal tree plan is shown in Figure 4-4.

![Figure 4-4: An optimal tree plan resulted from the example given in Figure 4-3.](image)

The algorithm only considers tree plans whose operators are connecting event classes that appear successively in the pattern. In other words, for a given pattern “\(A_1; A_2; ...; A_n\)”, tree plans with operators connecting \(A_i\) and \(A_j\), where \(i \neq j+1\) are not considered. We do not consider such tree plans because the intermediate composite event results generated by the operators connecting non-successive event classes \(A_i\) and \(A_j\) have “holes” in between \(A_i\) and \(A_j\), as illustrated in Figure 4-5. That means those “hole” classes (e.g. \(A_i\)) in between \(A_i\) and \(A_j\) should later be combined with these composite events, which is not
efficient because locating time ranges and hole positions within composite events are not straightforward.

![Diagram](image)

**Figure 4-5:** An example illustrating holes in a composite event.

### 4.3 Conclusions

In this chapter, we proposed a model to estimate the cost of a tree plan by adding up the cost of all its composite operators. Based on this cost model, we developed a dynamic programming algorithm to search for the optimal physical tree plan for a given logical query pattern.
CHAPTER 5: PERFORMANCE EVALUATION

This chapter analyzes the performance of ZStream in detail. We begin with an experiment to measure the optimal batch size in the batch-iterator model. Using this batch size, we run a series of experiments to show that selectivity of parameterized predicates and event rates dramatically affect the performance of different query plans. We also show that the designed cost model can capture these differences accurately. Finally, we demonstrate the correctness and efficiency of negation pushed down for sequential negation queries.

5.1 Experimental Setup

All experiments were run on a dual core CPU 3.2GHz Intel Pentium 4 with one core turned off and 2GB RAM size per processor. We ran ZStream on a pre-recorded data file; data was read into the system at the maximum rate the system could accept. System performance was measured by the rate at which input data was processed (input data size / the total elapsed time to process the data) and overall running time. The input data is stock trade data with the schema shown in Table 2-2. We implemented a script to generate synthetic stock events so that event rates and the selectivity of parameterized predicates could be controlled.

In addition, for any given query pattern, to avoid unnecessary complexity caused by irrelevant events\(^5\), we made the event generating script generate only relevant events as input. Since all SARGable predicates are pushed down to the leaf buffers, irrelevant events will be filtered out before they can arrive in any leaf buffers in any case, which

\(^5\) Defined in Section 2.2.1.1. Relevant events are defined as events that belong to any event class in the pattern and also pass all SARGable predicates associated with that event class. All other input events are defined as irrelevant events.
means they do not contribute to composite event assembly, which is the major source of costs. Query 5-1 shows the basic query used for most experiments in this section.

**Query 5-1:** Find a trade of IBM followed by Sun with lower price, followed by an Oracle trade within 200 time units.

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>Trade1; Trade2; Trade3</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>Trade1.name = 'IBM'</td>
</tr>
<tr>
<td>AND</td>
<td>Trade2.name = 'Sun'</td>
</tr>
<tr>
<td>AND</td>
<td>Trade3.name = 'Oracle'</td>
</tr>
<tr>
<td>AND</td>
<td>Trade1.price &gt; Trade2.price</td>
</tr>
<tr>
<td>WITHIN</td>
<td>200</td>
</tr>
<tr>
<td>RETURN</td>
<td>Trade1, Trade2, Trade3</td>
</tr>
</tbody>
</table>

### 5.2 Batch-Iterator Model Evaluation

As described in Section 3.4, operator evaluation algorithms take advantage of the earliest allowed time (EAT) calculated in each batch round to discard outdated events. This works well when batch sizes are relatively small. Remember that the EAT is calculated by subtracting the time window constraint from the *earliest* end-timestamp of the final leaf buffer; while the time range considered in each assembly round is from EAT to the *latest* end-timestamp of the final leaf buffer. In the extreme case where the batch size is set to 1, there is only one event instance \( f \) in the final leaf buffer in each assembly round. Then, the EAT can represent the exact earliest allowed time for all composite events having their last composed events as \( f \). Hence, the EAT constraint itself would guarantee that all composite events generated from the root and internal nodes are within the time window constraint. However, if batch sizes are big, the time range considered in each assembly round can be relatively large, resulting in some unnecessary intermediate results being generated as illustrated in Figures 3-6 and 3-7. Since the generated composite events may
have their occurrence time period exceeds the entire time window allowed by just applying EAT, the time window constraint must also be applied. We use three alternative ways to solve this problem:

1. *tw down*: The time window constraint is pushed down to the internal tree nodes to make sure that each intermediate composite event satisfies the time constraint. EAT constraints are applied during each batch assembly round.

2. *tw on top*: A time window filter is put on top of the root operator such that only those composite events within the time window constraint are output. EAT constraints are applied during each batch assembly round.

3. *no tw*: Only EAT constraints are applied. This is only applicable for batch size equal to 1.

![Graph](image)

**Figure 5-1: Throughputs of the left-deep plan of Query 5-1 with increasing batch size.**

Figure 5-1 illustrates the experimental results of these three strategies with increasing batch sizes. The experiments are run against the left-deep query plan “((Trade1; Trade2); Trade3)” of Query 5-1. The time window constraints applied on internal nodes or the root
is “WITHIN 200 units”. The input data size is 6000. Other parameters of the input data set are shown as in Table 5-1. The relative event rates amongst IBM, Sun and Oracle is proportional to 1:1:1, and the selectivity of the parameterized predicate “Trade1.price > Trade2.price” is set to 1/2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Rates (IBM : Sun : Oracle)</td>
<td>1:1:1</td>
</tr>
<tr>
<td>Selectivity of “Trade1.price &gt; Trade2.price”</td>
<td>1/2</td>
</tr>
</tbody>
</table>

The experiments show that the best performance is achieved by the no tw strategy with batch size 1. This verifies our claim that when batch sizes are small, EAT constraints can tightly bound the number of unnecessary results. Our concern to put a time window on top is that when the EAT constraint is relatively tight, most composite events generated from the root operator will have already satisfied the time constraint such that only a few results will be ruled out. Our concern to push time windows down to internal nodes is that the results can be filtered earlier, minimizing the number of intermediate results generated. As shown in Figure 5-1, however, tw down outperforms tw on top for all batch sizes, indicating minimizing intermediate results in fact significant.

As the batch size increases, more unnecessary intermediate results are assembled as EAT constraints have relatively smaller impacts on keeping the time range considered in each assembly round small. Hence, system performance degrades. Keeping time window on top degrades faster than when time windows are pushed down because the selectivity of the EAT degenerates quickly as the batch size increases. Similar conclusions can be made by varying the input data sizes and time windows.
5.3 System Scalability

Figure 5-2 illustrates system performances with increasing input size when running two alternative plans of Query 5-1; one is a left-deep plan “((Trade1; Trade2); Trade3)” where Trade1 and Trade2 are combined first, and the other is a right-deep plan “(Trade1; (Trade2; Trade3))” with Trade2 and Trade3 combined first. We use the no tw strategy for this experiment. Other parameter settings for input data are shown in Table 5-1.

![Figure 5-2: Throughputs of the left-deep plan and the right-deep plan of Query 5-1 with increasing input size.](image)

Figure 5-2 shows that system throughputs are stable for both the left-deep and right-deep plans for a fairly large range of input sizes. In addition, since the left-deep plan evaluates the sequence operator which has a parameterized predicate “Trade1.price > Trade2.price” first, it outperforms the right-deep plan as expected. Also, the percentage by which the left-deep plan outperforms the right-deep plan stays the same, indicating
that performance gaps between different query plans are not affected much by the input
data size or the running time of plans.

5.4 Parameters Affecting Costs

In this section, we experiment on various factors that affect the costs of query plans. Based on these experiments, we demonstrate that the cost model proposed in Chapter 4 accurately reflects the system performance.

5.4.1 Parameterized Predicate Selectivity

The experiments are conducted on two sequential queries, Query 5-1 and Query 5-2 by using both left-deep plans and right-deep plans. Query 5-2 is similar to Query 5-1 but uses a longer pattern. In addition, its parameterized predicates are applied on the last two event classes instead of the first two as in Query 5-1. We assume that incoming trades have a uniform distribution over stock names, which indicates relative event rates are 1:1.

Query 5-2: Find a sequential pattern of the form “IBM trade; Sun trade; Oracle trade;
Google trade; Yahoo trade” where the price of Google is higher than that of Yahoo within 100 time units.

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>Trade1; Trade2; Trade3; Trade4; Trade5</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>Trade1.name = 'IBM'</td>
</tr>
<tr>
<td>AND</td>
<td>Trade2.name = 'Sun'</td>
</tr>
<tr>
<td>AND</td>
<td>Trade3.name = 'Oracle'</td>
</tr>
<tr>
<td>AND</td>
<td>Trade4.name = 'Google'</td>
</tr>
<tr>
<td>AND</td>
<td>Trade5.name = 'Yahoo'</td>
</tr>
<tr>
<td>AND</td>
<td>Trade4.price &gt; Trade5.price</td>
</tr>
<tr>
<td>WITHIN</td>
<td>100</td>
</tr>
<tr>
<td>RETURN</td>
<td>Trade1, Trade2, Trade3, Trade4, Trade5</td>
</tr>
</tbody>
</table>
Figure 5-3: Throughputs of the left-deep plan and the right-deep plan of Query 5-1 with various selectivities for the predicate Trade1.price > Trade2.price.

Figure 5-4: Throughputs of the left-deep plan and the right-deep plan of Query 5-2 with various selectivities for the predicate Trade4.price > Trade5.price.
Figure 5-3 illustrates the throughputs of the two alternative plans (the left-deep plan and the right-deep plan) for Query 5-1. The left-deep plan outperforms the right-deep one because it evaluates the operator with the parameterized predicate first, such that it generates fewer intermediate results. The smaller the selectivity, the fewer intermediate results are produced. Hence, the gap between the performance of the two plans increases with decreasing selectivity. Throughputs for each plan increase with decreasing selectivity because lower selectivity generates fewer (intermediate) results. Similar results are shown in Figure 5-4 for Query 5-2 which has a longer pattern. But in this case, the right-deep plan evaluates the operator with the parameterized predicate first; hence the right-deep plan outperforms the left-deep one.

An interesting observation is that even in the case where selectivity is 1, which means the intermediate result size is the same for the left-deep and the right-deep plans, the plan with the predicate pushed down still slightly outperforms the other one. This small gap is caused by additional predicate evaluation. In Query 5-1 for example, its left-deep plan evaluates the parameterized predicate when combining *Trade1* and *Trade2* together, while its right-deep plan evaluates the predicate when combing *Trade1* and the results of (*Trade2, Trade3*) together. Obviously, the number of predicates evaluated is different in these two cases. Hence we can amend the cost model described in Chapter 4 as follows:

---

**Formula 5-1 Revised Cost Formula:**

\[ C = (1+nk) \, C_i + pC_o \]

The cost \( C \) of an operator consists of three parts: the cost of accessing the input data \( C_i \), the cost to generate the output data \( C_o \) and the cost of predicate evaluation \( (nk) \, C_o \) where \( n \) is the number of parameterized predicates the operator has, and \( k \) and \( p \) is a predefined parameter.

---
As described in Chapter 4, $C_i$ is measured by the number of records passed into an operator, and the $C_o$ is measured by the number of real events composed. The parameter $p$ determines the weight between $C_i$ and $C_o$, and its value is still set as 1. Since the predicate evaluation is performed during the process of accessing the input data, its cost is proportional to $C_i$. The parameter $k$ can be estimated based on the experimental throughput gap when the selectivity is equal to 1. Its estimated value is around 0.2 in Zstream.

![Running Time - Selectivity, Estimated Cost - Selectivity, Revised Estimated Cost - Selectivity](image)

**Figure 5-5:** Running time, estimated costs and the revised costs of the left-deep plan and the right-deep plan of Query 5-1 with various selectivities for the predicate Trade1.price > Trade2.price.

Figure 5-5 shows the real running time, estimated costs and the revised costs of the left-deep plan and the right-deep plan of Query 5-1 with various selectivities. As shown, the cost model describes the system behavior well except in the case where the selectivity is equal to 1, while the revised cost model also captures this. In most cases, however, if

---

6 The procedure of (revised) cost estimation for queries in this Chapter is shown in Appendix A.
we just want to estimate the major trend or relative performance of query plans, the original (coarse) cost model is sufficient. We get the similar results when experimenting on Query 5-2.

5.4.2 Event Rates

The event rate is another factor that affects the cost of running a query. Obviously, if the number of events arriving at the system doubles, more time is needed to process the additional data. A more interesting question is to see how variances of relative event rates of different event classes affect the cost of query plans for each query. 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 parameterized predicates, we experiment on two query plans (left-deep and right-deep) of a simple query as Query 5-3 which has no parameterized predicates involved. We keep the input data rate as one trade event per tick and vary the relative event rate of the input. The input data size is 6000 and 200000 for the experiments shown in Figure 5-6 and Figure 5-7 respectively.

Query 5-3: Find a trade of IBM followed by Sun, followed by an Oracle trade within 200 time units.

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>Trade1; Trade2; Trade3</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>Trade1.name = 'IBM'</td>
</tr>
<tr>
<td>AND</td>
<td>Trade2.name = 'Sun'</td>
</tr>
<tr>
<td>AND</td>
<td>Trade3.name = 'Oracle'</td>
</tr>
<tr>
<td>WITHIN</td>
<td>200</td>
</tr>
<tr>
<td>RETURN</td>
<td>Trade1, Trade2, Trade3</td>
</tr>
</tbody>
</table>
Figure 5-6: Running time and estimated costs of the left-deep plan and the right-deep plan of Query 5-3 with various event rates between relevant events (IBM: Sun: Oracle).

Figure 5-7: Running time and estimated costs of the left-deep plan and the right-deep plan of Query 5-3 with various event rates between relevant events (IBM: Sun: Oracle). In this case, rate differences between events are huge.
Figure 5-6 shows that the system has the largest running time and estimated cost when input stocks arrive at the same rate (IBM: Sun: Oracle = 1:1:1). When the distribution is biased, the running time and estimated cost decrease quickly because many fewer final results are generated. In addition, Figure 5-6 shows that when Oracle stocks arrive faster than IBM and Sun, the left-deep plan that combines IBM and Sun first outperforms the right-deep one a little, but not much. Similarly, when IBM has a higher rate, the right-deep plan outperforms the left-deep plan a little. However, this does not mean relative event rates between event classes do not affect system performances significantly. It only show up when arrival rates vary significantly. As shown in Figure 5-7, when the event rate of Oracle is much higher than the other two (30 – 50 times higher), the left-deep plan outperforms the right-deep one significantly. In both Figure 5-6 and Figure 5-7, we show the corresponding estimated costs for the two plans. Here, our cost model is able to capture the system cost for various event rates quite accurately.

5.5 Negation Pushed Down

As discussed in Section 3.5.5, we have two ways to evaluate sequential negation patterns. One is to put the negation on top of the entire query plan, and rule out the negated composite events after all the sequence operators are applied. The other is to use the N-Seq operator that directly incorporates negation into the sequence operator. In this section, we compare the performance of these two methods. The experiment is run on two query plans for Query 5-4:

**Plan 1:** Use a left N-Seq operator to combine Trade2 (Sun) and Trade3 (Oracle) first. Then a Seq operator is applied to combine Trade1 (IBM) with the results from the N-Seq. A similar query plan is shown in Figure 3-13.

**Plan 2:** Use a Seq operator to combine Trade1 (IBM) and Trade3 (Oracle) first. Then a
Neg operator is applied to rule out the \((IBM, Oracle)\) pairs where \(Sun\) events occurred in between.

**Query 5-4:** Find a trade of IBM followed an Oracle trade within 200 time units where no Sun trade occurs in between.

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>Trade1; ! Trade2; Trade3</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>Trade1.name = 'IBM'</td>
</tr>
<tr>
<td>AND</td>
<td>Trade2.name = 'Sun'</td>
</tr>
<tr>
<td>AND</td>
<td>Trade3.name = 'Oracle'</td>
</tr>
<tr>
<td>WITHIN</td>
<td>200</td>
</tr>
<tr>
<td>RETURN</td>
<td>Trade1, Trade3</td>
</tr>
</tbody>
</table>

Figure 5-8 illustrates the results for these two plans with various relative event rates. The input data size is 200000, with one trade event per tick. From Figure 5-8, we can see that the Plan 1 always outperforms Plan 2. Again, the longest running time is achieved when input stocks arrive at a same rate. When the relative event rates are skewed, the performance increases fast for Plan 2 because it generates many fewer (intermediate) results. As shown in the first graph in Figure 5-8, however, when Oracle event rates are increasing, the cost of Plan 1 actually increases slightly. This is due to how N-Seq works. As shown in Algorithm 3-5, N-Seq matches each Oracle event \(o\) with the latest Sun event before \(o\). Hence increasing the rate of Oracle will significantly affect the amount of computation done by N-Seq, which counteracts the benefits introduced by the biased distribution. Another observation is that the cost of the Plan 2 decreases much more quickly when event distribution is biased to Sun events than to the other two. This is because the Plan 2 combines IBM and Oracle first. The distribution biased on Sun will cause many fewer combined \((IBM, Oracle)\) pairs.
Figure 5-8: Running time of two query plans of Query 5-4 with various relative event rates. One pushes the negation down using an N-Seq operator, and the other performs negation on the top of the entire plan.
5.6 Conclusions

In this chapter, we described several experiments conducted on ZStream. We showed that smaller batch sizes can make EAT constraints more efficiently by bounding the time period in which a composite event is allowed to occur. We illustrated how selectivities of parameterized predicates and various relative event rates affect the cost of query plans for a given query, showing that the optimal plan depends on these rates and selectivities. In addition, we revised the cost model introduced in Chapter 4 to also include the cost of parameterized predicate evaluation. Finally we compared the performance of two ways to implement negation. According to the experiments, pushing negation down, and incorporating it into the sequence operator almost always outperforms putting a negation filter on top of the plan.
CHAPTER 6: CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

This thesis presented ZStream, a high performance CEP system designed and implemented to efficiently process sequential patterns. As described in Chapter 2, sequential patterns are widely used to capture a series of events occurring in temporal order in many real world applications. Besides simple sequentiality, ZStream is also able to support other relations such as conjunction, disjunction, negation and Kleene Closure. For Kleene Closure, it can either group the maximal number of events on the closure event class (e.g., \( A^* \) or \( A^+ \)), or accept a closure count \( n \) of the form \( A^n \) such that the number of events to be grouped in a closure can be controlled.

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 a cost model was proposed to estimate the computation cost of a plan. Experiments described in Chapter 5 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. Finally, we proposed a dynamic programming algorithm and equivalent transition rules to search for an optimal query plan. In addition, 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.
6.2 Future Work

In the future, we plan to extend ZStream in four directions:

1. Adding support for pushing negation into the query plans of PPN (parameterized predicates involved with negation) patterns. As shown in Chapter 5.5, incorporating negation into sequence evaluation can significantly increase performance for NPPN (non parameterized predicates involved with negation) pattern. Hence it may also be useful to extend this idea for PPN patterns.

2. ZStream will be extended to efficiently support query plans that combine non-successive event classes together. In the case where two non-successive event classes have very selective parameterized predicates between them, combining them first may be a good idea.

3. ZStream will be extended to adaptively choose and evaluate an optimal plan. For real time applications, the statistics such as relative event rates and predicate selectivity may change with time.

4. ZStream will be extended to use some common optimization technologies such as partitioning, hashing, etc.

In addition, we plan to evaluate ZStream and the dynamic programming algorithm used to search for the optimal plan on some real data set, such as real stock data and CarTel data. Based on this evaluation, we can see how different physical plans affect performance in real applications. We will also compare the performance of ZStream with other CEP systems and data streams.
REFERENCES


APPENDIX A

Original and Revised Estimated Cost for the Left-deep and Right-deep Plan of Query 5-1.

The input data is one stock event per tick, and the time window is 200. Since the relative event rate between IBM: Sun: Oracle is proportional to $1:1:1$, the event rates of IBM, Sun and Google are all $200/3$. Then the cost for the left-deep plan and the right-deep plan can be estimated according to the cost formula shown in Table 4-2. The input cost and the output cost for each operator is listed as below. After the input cost and the output cost for each plan are estimated, we can easily calculate the revised estimated cost by using Formula 5-1. We use the parameter $k = 0.2$ and $p = 1$.

<table>
<thead>
<tr>
<th>selectivity</th>
<th>seq1</th>
<th>seq2</th>
<th>total</th>
<th>seq1</th>
<th>seq2</th>
<th>total</th>
<th>raw data</th>
<th>Total Cost = C1+C2+Buffer</th>
<th>Revised Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>left-deep plan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.00</td>
<td>2222.22</td>
<td>74074.07</td>
<td>76296.30</td>
<td>2222.22</td>
<td>74074.07</td>
<td>76296.30</td>
<td>400.00</td>
<td>152992.59</td>
<td>153488.1</td>
</tr>
<tr>
<td>0.50</td>
<td>2222.22</td>
<td>37037.04</td>
<td>39259.26</td>
<td>1111.11</td>
<td>37037.04</td>
<td>39259.26</td>
<td>400.00</td>
<td>77807.41</td>
<td>78055.19</td>
</tr>
<tr>
<td>0.25</td>
<td>2222.22</td>
<td>18518.52</td>
<td>20740.74</td>
<td>555.56</td>
<td>18518.52</td>
<td>20740.74</td>
<td>400.00</td>
<td>40214.81</td>
<td>40338.7</td>
</tr>
</tbody>
</table>

<p>| right-deep plan | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th>selectivity</th>
<th>seq1</th>
<th>seq2</th>
<th>total</th>
<th>seq1</th>
<th>seq2</th>
<th>total</th>
<th>raw data</th>
<th>Total Cost = C1+C2+Buffer</th>
<th>Revised Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>2222.22</td>
<td>74074.07</td>
<td>76296.30</td>
<td>2222.22</td>
<td>74074.07</td>
<td>76296.30</td>
<td>400.00</td>
<td>152992.59</td>
<td>169511.1</td>
</tr>
<tr>
<td>0.50</td>
<td>2222.22</td>
<td>37037.04</td>
<td>39259.26</td>
<td>2222.22</td>
<td>37037.04</td>
<td>39259.26</td>
<td>400.00</td>
<td>115955.56</td>
<td>124214.8</td>
</tr>
<tr>
<td>0.25</td>
<td>2222.22</td>
<td>74074.07</td>
<td>76296.30</td>
<td>2222.22</td>
<td>18518.52</td>
<td>20740.74</td>
<td>400.00</td>
<td>97437.04</td>
<td>101566.7</td>
</tr>
</tbody>
</table>