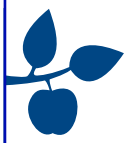


SYDDANSK UNIVERSITET
UNIVERSITY OF SOUTHERN DENMARK

Efficient Complex Event Processing Over Data Streams

Master's Thesis by
Jacob Aae Mikkelsen
kok04@imada.sdu.dk

Advisors: Yongluan Zhou
& Kim Skak Larsen

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

Abstract

A general purpose data stream management system (DSMS) is used for event processing over data streams. The events arrive in an on-line manner and must be treated efficiently. In this project, the essential elements of queries for a DSMS have been studied, leading to a query language that can handle a wide range of different queries. Most important is the ability to easily express queries that have a sequential structure, similar to regular expressions.

The focus of this thesis has been on processing queries that have such a sequential structure. In recent articles, a non-deterministic finite automaton (NFA) based approach is used. A tree based model is also possible, which is described and analyzed.

The algorithms used for processing data must be optimized to be able to handle the large amount of events. Optimizations using a lazy evaluation approach not generating partial results, until they are expected to be used is described. An optimization which, under the right circumstances, passes information between operators can, in those cases, improve the throughput of the system.

The two models are compared theoretically, demonstrating that the tree based model is at least as efficient as the NFA based model, along with an argument why the NFA model is not efficient for queries without sequence.

A prototype DSMS has been implemented including both models and used to experimentally test the difference between them. The results of the experiments substantiate the theoretical results, and also show that the tree based model clearly outperforms the NFA based model, when the time window of the query and thereby the amount of valid data is large.

Preface

> *A learning experience is one of those things that say, "You know that thing you just did? Don't do that."*
>
> Douglas Adams

During a whole year with a vaguely described project in an area of computer science that is brand new (at least at the Department of Mathematics and Computer Science in Odense) a lot of corners have been searched and a lot of blind alleys visited. There have been detours towards geometric data structures, Bloom filters, space complexity issues, and countless other topics.

The process has been allowed to go its own way, looking at more details in some small subproblems, and the balance between relational and pattern queries sliding towards pattern queries. A large part of the early work in this thesis was published by Mei and Madden of M.I.T. in an article midway through, creating additional pressure to come up with something new.

It has been a year with many learning experiences, and the result is now summarized in this report. The reader is expected to know the basic terminology from computer science, but is not expected to be an expert in event processing over data streams. A basic knowledge from a course in databases would however be helpful for the understanding.

Acknowledgements

I could not have done this thesis without help, so I am pleased to thank:

- My advisors Yongluan and Kim for competently guiding me through this process and encouragement.
- The former and current inhabitants of the office "Balkonen" for wonderful discussions and making it all more fun.
- Ida Coordt Elle and Asger Christiansen, whose proofreading improved the report greatly.
- My family and friends for support during the course of this thesis.

Contents

Chapter 1

Introduction

This chapter contains some applications of data stream management systems; the motivation of this project. It also contains a short survey of previous work in the area, in particular the material upon which is the basis of this project. Finally, it includes a summary of this thesis' contribution and an overview of the chapters.

1.1 Applications of Streaming Data

When data rates are too high to store in an ordinary database management system (DBMS) in time to answer queries, or the queries are continuous, a data stream management system (DSMS) is a solution.

A DSMS is only allowed to process a data event while it is in higher memory, once the event has left higher memory it is not retrievable again. Some examples of usage of a DSMS are listed here

Radio-frequency identification - RFID

RFID technology is used in a wide range of applications like product tracking, libraries, passports, transport payment, contact-less payment etc.. A general purpose DSMS could possibly reduce the implementation cost of systems to handle these quite different applications.

Financial applications monitoring stock-ticker streams

It is not difficult to imagine queries that could be posed in the stock market trading business. Finding patterns where a stock has dropped in price over a time period, but is changing is a simple example. Continuous monitoring is a necessity in such a situation.

Network monitoring data

Queries detecting attempts of "denial of service" attacks in a network requires monitoring packages in busy networks continuously.

Auction bids

An internet auction has the same characteristics as the stock-ticker stream. Updating the highest bid in the correct order is a continuous process until the auction ends.

Soldier surveillance

If soldiers in combat missions were monitored, queries involving heart rate, respiration, sweat production etc. could indicate if a soldier is experiencing too much pressure to be able to act rationally. Patient surveillance is another example.

Publish/Subscribe systems

Publish/subscribe systems like RSS must filter a large stream of keywords to locate the events that are relevant for a subscription. A subscription could be all blog entries that contains the keywords: "Roy Williams", "Dallas Cowboys", and "NFL".

Environmental sensor readings

With the recent development in sensor networks, sensor notes could be used for environmental monitoring, for example detecting forest fires in large areas. With the large number of measuring points, and therefore an increased chance of error, a query should detect if several nodes in the same geographic area would report unusual activity.

1.2 Challenges

Continuous Queries

In an ordinary DBMS, a tree is built when a query is posed, and the root of the tree can recursively ask for answers until there is no more data in the answer. The continuous queries of the DSMS cause the time of data arrival to decide when an answer to a query can be computed. This flips the direction of computation to bottom up, as inputs arrive in the leaves and propagate results to the top gradually. To achieve a better performance, it is important to minimize the computation of intermediate results.

High Data Rates (and/or bursty)

In many of the applications where a DSMS is used, the rate of input is simply too high for updating a DBMS. Time critical queries would not be answered in an acceptable time period. A patient surveillance system would not be successful, if the time period before detection of a heart failure would be just a few minutes.

Results being out of date

In queries calculating an aggregate over a time window, a result can time out, when the event gets too old. This situation can be predicted. However, a result previously produced can become invalid, when a new event arrives. A simple example of this is with aggregate operators, the maximum of an attribute in a stream of events could potentially be invalidated immediately after the result has been output if another event comes with a higher value.

1.3 Assumptions

Data does not arrive out of order

If data arrives out of order, this could cause generated results to be invalidated: The pattern that the query is searching for and output may not be valid if an event suddenly arrives that is placed in the middle of the pattern time-wise. If the data experiences slack, but the slack is bounded, this could be solved by preprocessing the data (and increase latency). This is not treated in this project, so an assumption is that data arrive in the correct order according to the end timestamp. The sequence join operator described in section 3.4 is optimized based on this assumption.

If two events have the same timestamp, the one arriving first in the input queue would be considered as earlier than the second, creating a total ordering. This has an effect when considering queries for patterns that are not allowed to skip any events, see section 2.1.5.

CNF without OR

The predicates involved in the queries are expected to be written in conjunctive normal form (CNF), and each clause is a single predicate without the use of OR. OR can be implemented as a post processing technique, creating two results and filtering doublets away. This is not considered in this thesis.

Regarding pattern queries, the left operator in a predicate is expected to match the event in a pattern. If two events are involved, then the left operator is expected to match the event that occurs first in the pattern. This is a trivial task to check and in case it is not fulfilled, then reverse the predicate. In a non-prototype system type-checking and weeding of queries would be implemented, but this has not been implemented in the work described in this report.

Output format

Some systems perform only event detection and output only that a match has occured. In this work event finding is essential, outputting not only that a result is available, but what the result is. This includes the possibility of outputting several results on one input.

It is also assumed that answers are exact, and not just an approximation. This imply that no events can be left out of the processing, regardless of the load on the

system.

1.4 Previous Work

A survey of work done prior to 2003 in the area of relational queries for data stream management can be found in [17]. It describes nine academic projects including STREAM, Aurora, SASE+ and TelegraphCQ .

The STREAM system[3, 5] from Stanford University uses the CQL[4] query language for relational style queries. The STREAM system uses adaptive algorithms for ordering of pipelined operators [7], to minimize processing cost. The CQL language is an SQL based syntax that can express continuous queries over streams and relations on disc. A library of queries written in CQL are available on the web [26].

The Aurora[1] project from Brandeis University, Brown University and M.I.T. express relational queries by dragging around boxes, representing operators, in a GUI based query system.

At University of California, Berkeley [6] presents work on adaptive query processing for shared-nothing databases. This made the foundation for the TelegraphCQ system[9], which in part is based on PostgreSQL version 7.3. The TelegraphCQ system handles relational queries.

The NiagaraCQ system [10] from the University of Wisconsin-Madison focus on internet data and a large number of queries, grouping queries to share memory and CPU cost. Their goal was to create a distributed database system for XML data and XML based query language.

At Cornell University [14, 13, 12], they have worked on the Cayuga algebra used in an NFA based system for pattern queries in publish subscribe systems. An extension of their work have been carried out by researchers at University of Massachusetts Amherst[2] extending the NFA with an event buffer at each state, and using this model in the SASE+ system[29, 15]. Further more, they are especially considering performance and runtime complexity along with several different contiguity requirements. A description of this work is found in section 2.3

Mai and Madden of MIT have with their ZStream [23] used a tree based evaluation model for pattern queries. They present an algorithm for calculating an optimal tree structure based on cost model analysis. This work was published halfway through this thesis, at a time where a tree based evaluation for pattern queries had been implemented and was under study. Parts of Mai and Maddens work is presented in chapter 3. The work presented in this report differs by considering some optimizations trade-offs, makes a more thorough comparison between the NFA based and tree based models, and considers different contiquity constraints in the tree based model.

Later in this thesis, more related work is referenced where it is relevant.

1.5 Contribution

The main contribution of this thesis is a thorough comparison of the NFA based and tree based approaches for finding patterns in streaming data. Using a tree for pattern

matching was developed and under study before [23] was published.

It is shown theoretically that the NFA based model is not suited for non-pattern queries, but more importantly a tree based approach could at least match the performance of the NFA and in many cases perform substantially better.

A prototype implementation has been done, and a series of experiments have been performed that substantiates the theoretical results. It is shown that when queries have large time windows, which is not an uncommon setting for processing data streams, then the tree based approach is notably better.

Several optimizations for general and specific cases have been suggested. These have been implemented and tested, showing promising results.

1.6 Thesis Outline

In Chapter 2, the components of queries for streaming data are outlined along with the design choice for the query language used in this thesis. Chapter 2 ends with a description of the NFA evaluation model for pattern searching, and some questions that should be answered in relation to this model.

Chapter 3 contains operators for the tree model for query evaluation of two different query types and a way to construct trees for both of them. Optimizations for the tree based evaluation model are described in chapter 4.

In Chapter 5 theoretical considerations on which model to use and comparison of the NFA and the tree model is performed, along with cost model analysis.

Experiments comparing the two models and showing the effect of the optimizations are placed in Chapter 6, and Chapter 7 contains the conclusion and suggestions for further work.

The appendix contains an analysis of an optimization for a left-deep tree based model.

Chapter 2

Data Model and Query Model

This chapter will start out with a description of the different elements that should be included in the query language. Some design choises for the language developped are stated, and the language is described along with some example queries.

The NFA evaluation model are described, as this is a common model used for pattern searching. This model will be used for reference and comparison in the later chapters. The chapter is concluded with a description of some limitations to the NFA evaluation model.

2.1 Components of Queries

In this project, queries are divided into two groups: Pattern queries and relational queries. Pattern queries specify an ordering or sequence of the events that are involved. Relational queries are those without a specific sequence of events.

The reason for creating a new language is that none of the existing languages studied supports both pattern queries and relational queries. Also not all languages supports different contiguity constraints. First a description of the elements that must be taken into consideration when designing a syntax.

2.1.1 Timestamp and Time Windows

When queries are continuous, time windows must be present in order to prevent overflow in memory usage. If no time window is present, a join must store all relevant events generating an indefinite data size.

Different strategies could be used when considering time. Should the user supply time or should it be the arrival time in the queue? Here, In order for the user to have full control, the events must be time stamped upon entering the DSMS. This have another positive effect, suppose a user would rather consider the last x events instead of the events occurring within the last y seconds. The user could then "timestamp" the events with one second between each, and let the time window of the query be x seconds, then exactly x events would be legal in the time window.

Several articles are treating time windows. [1] treats it in the Aurora system, [18] studies join algorithms for time windows, and time windows in general are handled

Id	Stream	Time
a	S3	20
b	S2	40
c	S1	60
d	S3	80

Table 2.1: Inputs to demonstrate the need for two timestamps.

in [22].

It is necessary to have a starting and ending time for events, in order to be able to use output from one query as input for another. With this possibility, nested queries can be expressed using several queries. The next example also show why double timestamps are preferable.

Why a single timestamp is not enough

This is most easily described using an example. Consider a query which must join events coming from three input streams: S1, S2, and S3, and the inputs from table 2.1.

If input from S1 and S2 are joined first, this will produce an event containing b and c. If the timestamp for this event is selected to be in the interval $[40-50]$, it will erroneously be able to join with a, and if the timestamp is in the interval $[50-60]$ it will erroneously be able to join with d. Two timestamps denoting the beginning and the end of an event are therefore necessary to compute the correct results.

2.1.2 Aggregate Functions

The well known aggregate functions Min, Max, Count, Sum, Avg should be considered when making a general purpose DSMS. Interpretation of these depends on the query type.

In pattern queries, the aggregate function would be calculated over the events that are included in a Kleene plus operation. One result would be generated for each time a series of events match the pattern in the query.

In relational queries, the aggregate function would be evaluated over all the events that satisfy the predicates within a time window. If this is made completely continuously, problems with invalidation of already outputted results arise. A result can be equipped with an expiration time that invalidates the result, i.e. when the event that generated the result times out from the time window of the query. However, often it is not possible to predict the exact expiration time. As an example, the average of an attribute will change as an event gets too old and is invalidated in the time window, but also as soon as a new event arrives making the previous outputted result invalid. Golab and Özsu presents different solutions to the problem in [20].

One solution for the problem of output becoming invalid is to use sliding windows, i.e. calculate the aggregate functions over a fixed time interval, sliding with a (pos-

sibly) different time window than the entire query. An example could be calculating the average hourly sale for a series of cash registers, but outputing the results once every 15 minutes. Each incoming event would then affect four time windows, making it possible to update those four time windows upon arrival of events, and when time is up, output the result and advance the window.

Just as with normal time windows, the slide can be made with number of events, if only the input it timestamped with one second between each. With just a small preprocessing making a conversion to timestamps, is it possible to use any attribute with a total ordering.

Should the input of data be irregular, then the aggregate functions would not be able to know the current time, making punctuations or heartbeat events a necessity in order to output results at the appropriate times.

Heartbeat Events

Heartbeat events act as clock ticks. There are a number of obvious sources for the heartbeats to come from, some are mentioned here, and more can be found in [27]. [27] also describes how punctuations can be used as assertions on input, relevant for when slides can be made. Heartbeats used in a special join type are treated in [16].

Source or sensor intelligence The source producing the data may have enough knowledge to supply the heartbeats, especially if the events are produced directly, since this would require time-stamping them.

Knowledge of access order If an event stream is produced by a scan or a fetch from data stored on disc or the like, the information needed to produce heartbeats may be present here. This could come from an index.

Auxiliary information If n sensors should report data, a list could be updated, and the heartbeat sent when all sensors have responded.

Internal information Even if an event is not matched by any operator, it still contains the current timestamp, and could be a source of a heartbeat.

2.1.3 What is Included From Where

A DSMS could potentially handle inputs from many different sources, just like a DBMS can handle inputs from many different tables. The inputs should therefore have a label to identify the stream it is coming from, and in order for queries only to consider relevant input streams. If data for a query is coming from more than one stream, it is necessary to join the input streams.

When considering pattern queries, the structure of the pattern must be specified: How many events are included in the pattern, and are any of the events allowed to be repeated (Kleene plus). When using pattern queries, it must therefore be specified which events in a pattern comes from which input stream.

As in DBMS, not necessarily all attributes of a join of relations are needed in the output. This calls for a projection-like operator that can filter out only the needed

attributes of the output result. If the `all` operator `*` is used in a pattern query, then the output will be an event with a list containing the events included in the pattern.

2.1.4 Selection Predicates

We must be able to consider different types of predicates. The notation for a predicate is as follows: $(left, relop, right)$ where $left$ is a stream with an attribute or a pattern event with an attribute, $relop \in \{=, != , <, >, \leq, \geq\}$ and right can be a value, a stream with an attribute or a pattern event with an attribute.

Naturally the predicates can be divided into categories as described below.

Static predicates A predicate is denoted static if it only involves one event. An example could be $(a.price, =, 42)$.

Parameterized predicates A predicate is denoted parameterized if it involves events from more than one stream or two events from the same stream. An example could be $(a.price < c.price)$

Relative to Last predicate When using the Kleene plus operator, this type of predicate can be used to compare an event to the previous event in the pattern. Example: $(a.price[i-1] < a.price[i])$ for a strictly increasing sequence of prices.

Other predicate types are possible, having events relate to other than the last event in the kleene plus sequence, using aggregate functions in predicates, a partition operator etc. but these have not been considered in this work.

2.1.5 Selection Strategy

The selection strategy is only applicable to pattern queries. It decides the condition for contiguity, i.e. is it allowed to skip any events in the input sequence. Three strategies are described here, including a few remarks on their complexity.

Strict No events can be skipped from the streams involved in the query. If the query does not contain any Kleene plus', then each intermediate result will either match the next event and grow longer or not match and could be removed.

In this situation, there is only a linear number of intermediate results possible, independent of the number of active events in a time window. However, if the query contains Kleene plus, there could be a polynomial number of intermediate results bounded by the number of active events in the time window.

Skip till any Any event can be skipped in the sequence. This also means, that any partly matched pattern from the start of the pattern could be continued.

If the length of the pattern is n and the number of events arriving in the time window is k, an arriving event could: 1) Start a new pattern by matching a_1. 2) For each existing partial match create a new partial pattern. Assuming every event matches as the next event in all partially matched patterns, the number of partially matched patterns would be $\boldsymbol{O}\left(n^k\right)$. This also holds if the pattern involves Kleene plus.

Skip till next Only skip irrelevant events. Relevant in the sense that a partition operator must be present, and irrelevant are all events that do not belong to the partition. Considering a stock exchange example, wanting the name of any stock with strictly increasing prices over a time window, partitions are determined by the name, and an event is only relevant for one partition, being irrelevant for all others. As no partition operator is considered in this project, neither is this strategy.

The complexity of the selection strategies is comprehensively described in [2]. From the complexity of the strategies, the exponential complexity of the skip till any strategy is most difficult to deal with, and the focus of the present work has therefore been to optimize this strategy.

2.1.6 The Query Language

Query languages for both relational and pattern based queries have a lot of similarities, as described above.

It would be advantageous if the output stream can be used as an input for other queries, and for this to be possible, the outputted events must have a stream label, which should be set from the query. The requirements for a query language can therefore be summarized as:

- Output label, if the output should be used as an input.
- Which input streams should be considered.
- Size of the time window in the query, and if aggregate operators are used, possibly a length of the slide.
- Predicates for event selection.
- What attributes of the selected event that should be outputted.
- A pattern structure, if it is a pattern query.

Given the above constraints, the query language is defined to be

```
<Output stream label> :
SELECT <Projection criteria>
PATTERN <Pattern structure>
FROM [ RANGE <time window> SEC SLIDE <slide time> ] : <Streams>
WHERE <selection strategy> [predicates]
```

where the optional parts are the first line defining the output stream name, the third line if it is not a pattern query, the `SLIDE <slide time>` from the fourth line and the selection strategy from the last line. If no selection strategy is marked, the strict is set as default.

The `<Streams>` part must be a list of streams separated by `;` if it is a relational query, or a list of streams with the events from the pattern that belongs to each stream.

An example of a relational query with aggregates and sliding window is

Query 1

```
S1: SELECT SUM(price) , MAX(price)
FROM [RANGE 10 SEC SLIDE 5 ] S1
WHERE S1.name = "IBM"
```

and an example of a pattern query with Kleene plus is

Query 2

```
SELECT *
PATTERN ab+c
FROM [RANGE 2 SEC] S: a,b,c
WHERE SKIP_TIL_ANY(a,b,c) {
      a.name = "IBM" AND b.name = "Sun" AND c.name = "Oracle"
}
```

2.2 Definitions and notation

Definition 1 (**Event**)**.** An event e $\langle s, \tau_{start}, \tau_{end} \rangle$, where s is a mapping from a schema to corresponding values and τ_{start}, τ_{end} is the beginning and ending timestamps of the event.

Definition 2 (**Tuple**)**.** A tuple t is a collection of non-overlapping events each having a type from the pattern it belongs to. The tuple also has beginning and ending timestamps.

Definition 3 (**Stream**)**.** A stream S is a (possibly infinite) bag (multi-set) of events, ordered so the τ_{end}'s of the events are non-decreasing.

In this project, a stream is expressed as $e_1 e_2 e_3 \ldots$ This definition differs slightly form the definition in [4], since a single timestamp is not enough, if the output of queries is to be used for input to other queries. See section 2.1.1 for an elaborate explanation.

Definition 4 (**Pattern**)**.** A pattern is a sequence of non-overlapping events from one or more streams. An event in the pattern can have requirements to single attributes, and the events can have requirements to attributes between them.

A pattern can include a Kleene plus, which allows several events fulfilling the requirements to be included in the solution of pattern.

A pattern is written as $\mathtt{a}_1\mathtt{a}_2\mathtt{a}_3 \ldots \mathtt{a}_n$ or $\mathtt{a}_1\mathtt{a}_2\mathtt{+a}_3 \ldots \mathtt{a}_n$ if events matching the second event in the pattern are allowed to be repeated.

Definition 5 (**Initial event**)**.** An event matching a_1 in a pattern is called an initial event.

Definition 6 (**Triggering event**)**.** An event matching a_n in a pattern is called a triggering event.

Definition 7 (**Selectivity**)**.** If the percentage of events accepted by a predicate is small, it is called highly selective or having low selectivity

Definition 8 (**Throughput**)**.** Throughput of the system is calculated as the number of events processed each second, when the system is not idle and waiting for events in the process. The measure is $\frac{events}{second}$

Definition 9 ($\in_R$)**.** When a value is chosen uniformly at random from a set, $\in_R$ denote this.

2.3 Pattern Query Evaluation with NFA

This section describes the NFA model for pattern matching in event streams, as described in [13, 2].

2.3.1 NFA Model Description

The model consists of a NFA, where with each state a buffer for events are attached.

The structure of the NFA is determined from the query. The building of the structure is illustrated by example from the pattern `ab+c`, and the *skip till any* selection strategy.

The starting state is labeled with the first character from the input, `a` with a forward pointing begin edge. The edge points to the next state, which is labeled `b[1]` as the `b` from the input is followed by a Kleene star. This state has a forward pointing begin edge to a state labeled `b[i]`, which has a looping *take* edge. From this state, a forward pointing proceed edge, that is nondeterministic (ε), points to the next state labeled `c`. From this state a begin edge points to the accepting state (`F`). Since we are allowed to skip any events in the sequence, the states between the starting and the accepting state are equipped with a looping *ignore* edge. The structure can be seen in figure 2.1

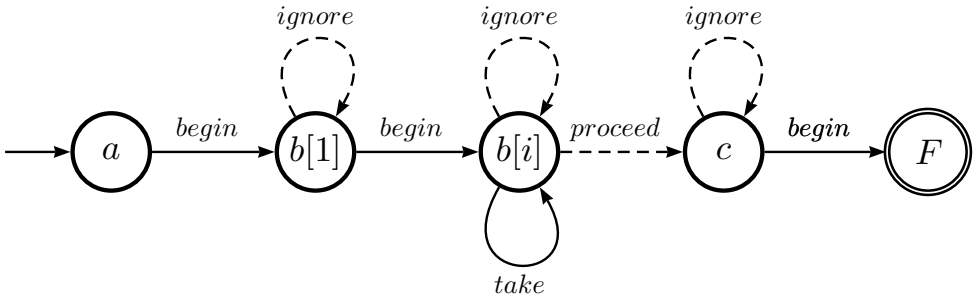


Figure 2.1: A sample NFA for the pattern structure `ab+c` with the *skip till any* selection strategy.

If the selection strategy is strict, no ignore-edges are necessary, but they could be included and just evaluated to false. It is clear that a mixed strategy is possible, i.e. part of the pattern is strict and part of the pattern are allowed to skip events, since this depends only on which ignore edges exist (respectively evaluates to true).

2.3.2 Allocating predicates to the edges

The predicates of the query must be assigned to the edges in the following way.

Static predicates

A static predicate must be assigned to the begin edge of the corresponding event, and if there exists a loop edge on this too.

Parameterized predicates

A parameterized predicate cannot be evaluated until the last of the two events occur in the pattern. The predicate should be assigned to begin and forward edges going into states corresponding to the last occuring event in the predicate. If the state has an take edge, then the predicate must be assigned this edge too.

Relative to Last predicate

Predicates of this type should be assigned only to the relevant take edge. On the take edge, the configuration is guaranteed to have at least one event of the same type included, from the previous begin edge (or if the take edge already have been followed).

2.3.3 Configurations and Using the NFA

When using the model, and an event arrives for processing, the forward edge from the starting state is examined, and if it matches, the event is added to the buffer in the first state, and a new configuration is created. A configuration consists of the current state, a version number, a pointer to the most recent event in the configuration, the starting time, and a summary of the values that are included in the partial match to speed up predicate evaluation.

For all existing configurations, the incoming event is checked on all outgoing edges from the configuration,s current state. If an ignore edge exists, the configuration is kept as is. If a take edge exists and evaluates positively, the event is added to the configuration, and if a begin edge exists and evaluates positively, the event is added to the configuration and the current state is updated. If more than one of the above edges evaluate positively, the configuration must be copied before the second update.

The summary in the configuration should contain the values needed to evaluate predicates later in the pattern, saving time especially if the predicate involves an aggregate.

Once all configurations have been examined, output can be generated from the configurations that have an accepting current state, and those can then be removed from the list of configurations.

2.3.4 Version Numbers and Buffers

When an event is added to a buffer in a state, it is assigned a pointer, with a version number, to the event previously seen in the pattern. The local identification number is used to check if the event is already added to the buffer, and to identify which events belong to a configuration.

The version numbers for the pointers and configurations are of the form $id_1(.id_i)^*$ where id_1 is the local identification number from the first state, and id_i is the local number from the i^{th} following buffer. Note that two identification numbers can be identical without referring to the same finished tuple, if the NFA have two take edges in different states, and the two configurations skip events in different states. This is not a problem, as the version numbers are implemented using the local number and a pointer to the prefix of the number. From this, they can easily be separated.

2.3.5 Generating output from a Version Number

Once a configuration has reached the accepting state, it indicates that a match for a pattern has occurred. The configuration contains a wrapper with the last event and a list of previous pointers for events preceding in some pattern. In order to create the right pattern, it must backtrack to the first event in the pattern and build the pattern on the way back. Following the previous pointers to earlier wrappers that corresponds to the version number, the first event is found when a wrapper's previous pointer list is empty indicating this is the first state.

2.3.6 Limitations of the NFA Evaluation Model

The NFA evaluation model for pattern queries assumes that all the events are from one stream or a union of several streams, which means joins of multiple streams are not considered. If data arrives from several streams this partition should be exploited.

The NFA evaluation model is radically different from the one used in traditional DBMS and DSMS systems. A natural question to ask is: Is this radical change necessary? Can non-pattern queries be evaluated with this model or can we use the traditional evaluation model, i.e. a query plan organized as an operator tree, to efficiently evaluate pattern queries? How is such an approach compared to the NFA approach? Which one is more efficient?

Another potential benefit of using the traditional evaluation model is the ability to evaluate the queries with both relational joins and sequence pattern matching in a uniform manner. In other words, no need to handle the awkward combination of NFA and trees of join operators. Taking this into consideration, can the performance benefit of the NFA model justify such additional complexity into a DSMS? These questions are considered in the following chapters.

Chapter 3

Evaluating Pattern Queries with Operator Trees

The idea for using a tree structure for the DSMS comes naturally from how SQL queries are executed in a DBMS. Given an SQL query, the query engine of a DBMS will generate a tree-shaped query plan composed by various operators. The query results can be requested from the root node of the operator tree and the request will be propagated to the other operators in a top-down manner. In a DSMS, data is only accessible for a short period while in memory, so immediately after the tree is built, it is not possible to generate results, as the data has not arrived yet. As queries are continuous, the trees are used over a longer period, and the generation of results must start from the leaves, since they "know" when data is available. Since the root is unable to tell when results are present, it is the leaves that must initiate the production of resultst when data arrives.

This chapter will outline two different plans for generating results, the operators used in the trees and how trees are built.

3.1 Propagation Plan

The NFA model described in section 2.3 produces the first part of a pattern as soon as it is possible. Such eager strategy is applicable with evaluation trees too: as soon as a sub-result can be produced, it is propagated to the parent. This will be called the push-based propagation plan.

The pull-based evaluation plan adopts the typical lazy evaluation strategy. Figure 3.1 shows a sketch of an evaluation tree. Assume that subtree T_3 cannot generate any results. If this is the case, it is clear that no results can be produced from the entire tree. Since no results can be produced, there is no reason to evaluate the results in the subtrees T_1, T_2, and T_4 as this would be a waste of energy, if the intermediate results would timeout before being used. A lazy strategy where the root of a subtree can pull the results when needed seems to be possible. This technique for propagation is called the pull-based propagation plan.

Section 4.1 describes advantages and disadvantages of the two propagation plans

in more detail, but this introduction was needed to understand the operators used in the tree.

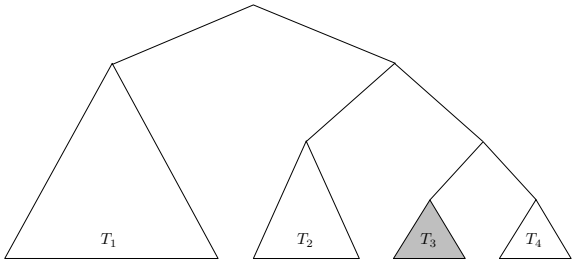


Figure 3.1: A tree where T_3 cannot produce results, to demonstrate why the pull-based plan could be beneficial.

3.2 Common Operators

The operators used in both the relational model and in pattern queries are described here.

Selection

The selection operator handles only the static predicates. If the incoming event satisfies the predicate, it is passed on to the selection operator's parent, if not it is discarded.

Aggregate Operator

The aggregate operator must initialize the memory for the sliding windows, depending on the number of slices the total time window is divided into. It is responsible for propagating an event once time for a slice has expired, and thus handling the heartbeat events that carry only timestamp information.

If no events or heartbeats have arrived in more time than two slices, it must output events for both slides in the window and advance the window accordingly.

Input

Besides being an entry point for data, the input operator checks if the event input matches the Stream it is supposed to come from. If the event does not match, there is no need to pass it on to the parent.

Output

The output operator is a placeholder for the output stream. When a dynamic tree structure is used, this will be the root, that has the dynamic part of thetree a child. It has it has no influence on the output of the query, except it assures that all output events have the correct label.

3.3 Operator for Relational Queries

In order to process input from more than one stream, a join operator is necessary. Many of the join algorithms known from from relational data queries are inapplicable in the data stream setting. Examples are joins that use some kind of partitioning or sorting, as the data involved is dynamic. The join algorithm presented below is a very simple algorithm, and optimizations are possible. It is included here only as a proof of concept example, to show that algebra trees in fact can be applied to relational queries on streaming data. Several papers describes solutions to this, among those [3, 9, 18, 19, 28].

Join

The join operator creates a new event from two input events from different streams. It handles the parameterized predicates that relate different streams.

Due to the on-line nature of the problem when two streams must be joined, it is necessary to store the events that are not outdated in the current time window from both input streams. If not it would be impossible to decide if an input from one of the streams would match a previous input from the other stream. Algorithm 1 shows the join.

Algorithm 1: Join algorithm for relational queries

```
   Input: Event e, child location t
 1 removeOutdatedEvents (e.end)
 2 if t == left then
 3     foreach Event er in right input list do
 4         if Time constraint and join condition are satisfied then
 5             newEvent ← Event created by merging of e and er
 6             parent.inputEvent (newEvent, parentType)
 7     place event in left input list
 8 else
 9     foreach Event el in left input list do
10         if Time constraint and join condition are satisfied then
11             newEvent ← Event created by merging of el and e
12             parent.inputEvent (newEvent, parentType)
13     place event in right input list
```

Projection

The projection operator takes as input an event. It then creates a new event and adds only the selected attributes from the incoming event. The new event is passed on to the parent.

3.4 Operators for Pattern Queries

Sequence Join

Due to the fact that the operator tree can have different shapes, the sequence join operator must be able to handle inputs both as simple events and as tuples. Determined by the selection strategy and the propagation strategy the operator must act accordingly.

Inspired by the version number concept described in section 2.3.4, when a tuple is constructed from another tuple, it does not copy the list of ids and events, but just a reference to the old tuple. A new summary table is however created from the old.

Algorithm 2 shows how event input is treated. The `child location` parameter indicates if the input comes from the left child or the right child. If the input comes from the left child, no tuples can be generated, since there are no events that have arrived after the end timestamp of the event. If it comes from the right child, tuples are pulled from the left child if needed through the getTuples method (algorithm 3). The getTuples method uses the return list as a parameter, so no copying of tuples are necessary. A very similar algorithm for tuple input exists, but is omitted from this report.

The removeOutdatedEvents removes events and tuples from the input queues that are too old, i.e if $now - beginTimestamp > TimeWindow$.

The getTuples algorithm (algorithm 3) is not as comprehensive as a first glance may indicate. At most one of the outer loops (line 8 and 15) will have any elements, and at most one of the inner loops (line 9,12,16 and 19) will have any elements (unless the pattern includes a Kleene plus, see below). This stems from the four combinations of event and tuple input, where the sequence join is in one of these configurations. It should also be noted that if all events match, the number of outputs is $\boldsymbol{O}\left(n^2\right)$, and the algorithm is asympthotically optimal in the worst case.

Kleene Plus

Handling Kleene plus is actually a part of the sequence join operator, but for clarity it is described separately. Only the sequence joins with inputs as events must handle the Kleene plus. If the language had been extended to include paranthesis with Kleene plus, thus allowing repetition of more than one event, then the internal sequence joins should have handled it. This has not been studied, but is listed as a topic in future studies.

How Kleene star is treated depends on it belonging to a left or a right input.

Kleene Plus is left input Must generate the Kleene plus tuples before matching with the right input.

Kleene Plus is right input Store a copy of the event passed on to the parent in the left input (Tuple) list. This enables the operator to add another event to the end of the tuple.

Algorithm 2: Sequence join algorithm for event inputs.

```
   Input: Event e, child location t
 1 removeOutdated(e.end)
 2 if t == left then
 3 |   place event in left input list
 4 |   return
 5 else
 6 |   if propagate then
 7 |   |   if left child does not propagate then
 8 |   |   |   getTuples(leftTupleInputList)
 9 |   |   foreach Event el in leftEventInputList until
   |   |   el.EndTime ≥ e.startTime do
10 |   |   |   if Time constraint and join condition are satisfied then
11 |   |   |   |   newTuple ← Tuple created from el and e
12 |   |   |   |   parent.inputTuple(newTuple, parentType)
13 |   |   foreach Tuple tl in leftTupleInputList until
   |   |   tl.EndTime ≥ e.startTime do
14 |   |   |   if Time constraint and join condition are satisfied then
15 |   |   |   |   newTuple ← Tuple created from tl and e
16 |   |   |   |   parent.inputTuple(newTuple, parentType)
17 |   else
18 |   |   place event in right input list
```

Kleene Plus is left and right input Kleene stars right after each other should be handled differently, so the sequence of the input is not mixed. Thus, a check must be performed before inserting from the left, that a right input is not the last in the pattern. Alternatively use a separate list to the events that have only events from the left input, and tuples that already have events included from the right input.

Buffer

In order to be able to rebuild a tree, where the intermediate results are discarded, the inputs must be buffered for the current time window. This buffering also allows the estimation of the optimal tree to use the size of the buffer as a measurement of the size of the input.

Tuple to Event Converter

Internally, the query evaluation tree for patterns handles tuples but must output events. This conversion can be considered as an operator in itself or placed in the output operator as an added functionality. It is quite similar to the projection opera-

Algorithm 3: GetTuples algorithm used with the pull evaluation strategy.

```
   Input: ResultList l , Timestamp now
 1 removeOutdatedEvents(now)
 2 if ¬ left child propagate then
 3     leftChild.getTuples(leftTupleList , now)
 4 if left tuple and event input list are empty then
 5     return
 6 if ¬ right child propagate then
 7     rightChild.getTuples(rightTupleList , now)
 8 foreach Event er in rightEventList do
 9     foreach Event el in leftEventList until el.end ≥ er.start do
10         if Time constraint and join condition are satisfied then
11             list.append(Tuple created from el and er)
12     foreach Tuple tl in leftTupleList until tl.end ≥ er.start do
13         if Time constraint and join condition are satisfied then
14             list.append(Tuple created from tl and er)
15 foreach Tuple tr in rightTupleList do
16     foreach Event el in leftEventList until el.end ≥ tr.start do
17         if Time constraint and join condition are satisfied then
18             list.append(Tuple created from el and tr)
19     foreach Tuple tl in leftTupleList until tl.end ≥ tr.start do
20         if Time constraint and join condition are satisfied then
21             list.append(Tuple created from tl and tr)
22 Empty rightTupleList and rightEventList
```

tor, creating a new event with the specified attributes. The difference is that it must locate the correct event in the pattern first, i.e. if *b.type* is what is needed, the event in the tuple that corresponds to *b* should be found first.

If the pattern contains a Kleene plus, then *sum*(*b.price*) would be a meaningful projection criteria, and here all events matching *b* must be considered, making the functionality similar to an aggregate operator.

3.5 Building the Tree

The evaluation tree is built differently depending on the type of query type (relational or pattern based) and if a dynamic or static structure is desired.

3.5.1 Relational Tree

For relational queries, a simple (unoptimized) way of building an evaluation tree is to first make a left-deep tree with joins of the different input streams, specified in the `FROM` part of the query, and then add selection operators above the input operators for the static predicates, and place the parameterized predicates in the joins, at the lowest place possible in the tree. The Stanford STREAM system [3, 4] has excellent solutions to this problem with well documented optimizations, as does the Aurora system[1] and others.

Figure 3.2 shows the tree corresponding to query 3. The reason for the last stream in the query being the lowest is due to the parser building lists in backwards order. An optimized version would try to find an optimal join order, but this has not been pursued in this project.

Query 3

```
SELECT S1.price
FROM [RANGE 5 SEC] S1 ; S2 ; S3
WHERE S1.name = "IBM" AND S2.name = "Sun" AND S3.name = "Oracle"
      AND S1.price < S2.price AND S2.price < S3.price
```

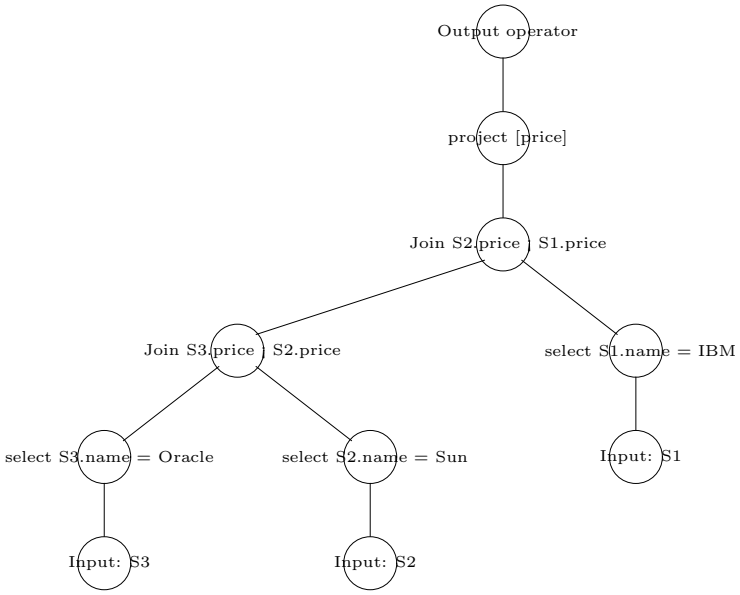


Figure 3.2: Sample tree for query 3

3.5.2 Static Structure for Pattern Query

A static structure has little overhead, i.e. no checking if the structure is optimal. There are however inputs that cause a static structure to perform poorly.

The first tree built in this project to handle pattern queries was the static leftmost tree. The leftmost tree resembles the NFA as will be described in section 5.2.1, and while working with the leftmost tree, the push and pull plans were developed along with other optimizations. The leftmost tree can be built in the following way (quite similar to the relational tree).

Start by making an input node from the last event in the pattern. Going backwards in the pattern, creating input nodes for each and keeping track of the two next, make a sequence join node with an input as a right child and a join as a left child, until there is only one event left in the pattern. This is then placed as left child in the lowest join. After the basic structure is created, the static predicates are inserted as selection operators just above the input operators, the parameterized predicates added to the sequence join nodes, and an output operator attached as root, which also handles turning the tuple into an event using the `SELECT` part of the query.

3.5.3 Dynamic Query Plan for Pattern Queries

The dynamic structure described next comes from [23], which was published during the course of this thesis. The contribution in this section is therefore "filling in the blanks".

The optimal shape of an evaluation tree will depend on the selectivity of the predicates and the rate of arrival, which is impossible to predict. An informed guess based on the data already seen can however be made. Algorithm 4 uses dynamic programming to make such a guess. Using a structure of the tree that can be rebuilt if needed, the inputs at each leaf must be stored, so intermediate results that are lost during rebuilding can be regenerated, which is where the buffer operator is relevant.

The initialization and purpose of the tree matrices are:

$Min[x][y]$ Minimum estimated price for evaluating a tree of size x starting from position y. The matrix must be initialized with ∞.

$ROOT[x][y]$ Calculated optimal location of the root in a subtree of size x starting from position y. Initialization is not important, as it only stores information needed for building the tree later.

$CARD[x][y]$ The expected output size of the output for a tree with size x starting from location y. The first row of the matrix must be initialized with the expected number of inputs in a time window for the event matching the location in the pattern. After the tree has been active for the first time window, this information can come from the buffer operators.

The algorithm runs in $\boldsymbol{O}\left(n^3\right)$, if the getSelectivity does constant work, in terms of the length of the pattern. It will of course depend on the number of parameterized predicates, as they determine the selectivity. See section 3.5.4 for a full description of getSelectivity.

Once algorithm 4 has calculated the optimal tree structure, the recursive algorithm 5 returns the root when called with arguments buildTree$(root, 1, n, root[n][1])$, and n being the length of the pattern.

Algorithm 4: Searching for optimal tree structure

Input: Number of event classes n
Output: buffer ROOT recording roots of optimal subtrees

```
1  Initialize two dimensional matrices Min, ROOT, CARD
2  for s ← 2 to n do
3      for i ← 1 to n - s +1 do
4          for r ← i + 1 to i+ s do
5              opc ← CARD[r − i][i] ∗ CARD[s − r + i][r] ∗ 0.5
6              cost ← Min[r − i][i] + Min[s − r + i][r] + opc
7              if Min[s][i] > cost then
8                  Min[s][i] ← cost
9                  root[s][i] ← r
10                 CARD[s][i] ← opc∗getSelectivity (i, i + s − 1, r)
```

3.5.4 Calculating selectivity in the root of a subtree

A parameterized predicate is relevant for a join operator in a tree, if it has all the information needed to evaluate, and cannot be evaluated in one of the subtrees of the join. If the inputs are numbered from 1 to n, a predicate can be denoted as $[start, end]$ where $start$ is the leftmost input that is involved in the predicate, and end is the rightmost input involved. A subtree handling inputs a to b with root at the r^{th} position can be described as an interval $[a, b]$ with a point $r \in [a, b]$. The root position r is the location of the leftmost leaf in the right subtree. When calculating the reduction factor for a root in a subtree, all reduction factors for the predicates that satisfy the following constraints should be taken into consideration.

- The predicate interval should be included in the subtree interval. If not, the subtree will not contain the necessary inputs for the predicate to be evaluated, and is therefore not relevant at the root.
- The point r should be located in the predicate interval. If it is not, then the predicate could be evaluated in one of the subtrees, thus lower in the tree, deeming it irrelevant for this root position. However, this should already be taken into consideration when computing the input size from the relevant subtree.

In algorithm 4 $\boldsymbol{O}\left(n^3\right)$ calls to getSelectivity with different parameters for each calculation of optimal shape could occur. Two solutions for the problem of locating the relevant predicates will be described here and compared after their description.

Solution using Range Trees

With the conditions described in the previous section, the problem can be rephrased as a geometric problem in the following setting. Let the predicates be points in a plane located at $(start, end)$, and the subtree interval $[a, b]$ with the point r be the

Algorithm 5: Building optimal tree structure from root matrix

Input: Roots *matrix*, start index *start*, end index *end*, root position *root*
Output: Root operator

```
1  if end − start == 1 then
2      return new sequence join with children buffers at start and end in list
3  if root − start > 1 then
4      left ← BUILDTREE(matrix,start, root − 1,matrix[root − start][start])
5  else
6      left ← buffer at start in list
7  if end − root > 0 then
8      right ← BUILDTREE(matrix,root, end,matrix[end − root + 1][root])
9  else
10     right ← buffer at end in list
11 return new sequence join with children left and right
```

problem of finding all points in the plane that are located in the axis aligned rectangle with x-coordinate interval $[a, r[$ and y-coordinate interval $[r, b]$. The root r cannot be included as the right endpoint in the x interval, as it would include predicates that could be evaluated in the right subtree (from the definition of root position). Figure 3.3 shows the geometric situation where there are three predicates $(1, 2)$,$(2, 4)$, and $(4, 5)$. Note that $(1, 2)$ should be evaluated in the left subtree, $(4, 5)$ in the right subtree but $(2, 4)$ in the root of the subtree that the problem describes.

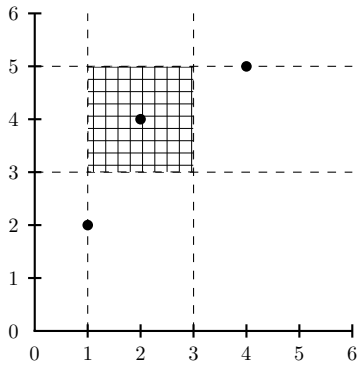


Figure 3.3: Geometric interpretation of finding predicates relevant for subtree with inputs 1 to 5 and root in 3.

Letting m denote the number of parameterized predicates, a solution using interval trees with fractional cascading[11] and the parameterized predicates as data points can be made in time $\boldsymbol{O}(\log m + k)$ where k is the number of relevant predicates (output size) using size $\boldsymbol{O}(m \log m)$ and using $\boldsymbol{O}(m \log m)$ time for preprocessing.

Solution using larger preprocessing and more space

Using $\boldsymbol{O}\left(n^3\right)$ space and more time spend on preprocessing the parameterized predicates, a solution with no unnecessary searching at each call to getSelectivity can be constructed.

Algorithm 6: Preprocessing parameterized predicates

Input: Number of event classes N
Output: buffer ROOT recording roots of optimal subtrees

```
1 Initialize three dimensional matrices reduction with empty lists
2 foreach Parameterized Predicate p do
3     pl ← left endpoint of p
4     pr ← right endpoint of p
5     for start ← 1 to pl do
6         for end ← pr to n do
7             for r ← pl + 1 to pr do
8                 add p to list in reduction[start][end][r]
```

Once algorithm 6 has run, all predicates relevant for a node being a root in a subtree can be located, as they are in the cube. Only predicates that must be included in the calculation of the selectivity are considered, thus the processing being optimal $\boldsymbol{O}\left(1+k\right)$, when no approximations are used.

The drawback of this solution is the $\boldsymbol{O}\left(n^3\right)$ space requirement for the cube, and a precomputation cost of $\boldsymbol{O}\left(mn^3\right)$. A comparison of the two solutions can be found in table 3.1. If only a limited number of rebuild are made, the solution using range trees should be chosen, due to the limited preprocessing, but if rebuilds are frequent ($\boldsymbol{\Omega}\left(m\right)$) or the query is running for a long time, then the enlarged preprocessing would be offset. Due to the small problem size, it is probably less important which solution is used.

	Range Trees	Preprocessed Cube
Preprocessing	$\boldsymbol{O}\left(m \log m\right)$	$\boldsymbol{O}\left(mn^3\right)$
Size	$\boldsymbol{O}\left(m \log m\right)$	$\boldsymbol{O}\left(n^3\right)$
Single evaluation	$\boldsymbol{O}\left(\log m + k\right)$	$\boldsymbol{O}\left(1+k\right)$
Total for rebuild ($\boldsymbol{O}\left(n^3\right)$) calls	$\boldsymbol{O}\left(n^3 \log m + n^3 k\right)$	$\boldsymbol{O}\left(n^3 + n^3 k\right)$

Table 3.1: Comparison of the two solution methods for finding relevant predicates for a root in a subtree. Here m is the number of predicates, n is the length of the pattern, and k is the size of the output.

3.5.5 When should Rebuilding Happen?

Checking if the current shape of the tree is optimal takes time, so checking too often will be too expensive, especially if it is not necessary to rebuild. On the other hand, checking too infrequently would waste time and work, using a tree that is shaped in a way that generates too many unusable sub-results.

Elements that have an effect on the optimal shape of the tree are:

Selectivity of the parameterized predicates If the selectivity of a parameterized predicate changes, then it will affect the output of the sequence join operator that evaluates it. This causes the number of propagated results to change, which could change the cost of the tree. If the sequence join operator is already placed near the leaves and the selectivity drops, it would probably be unnecessary to rebuild the tree, as the shape would still be optimal.

Number of inputs in a time window When the number of inputs for one of the leaves changes in a time window, the number of possible outputs from the sequence join that handles the input will also change. This change can arise from a change in the speed of data arrival or from the nature of the data changing, so the selectivity of the static predicate in the selection operator lower in the tree changes.

Total saving Rebuild should only happen if the total saving is large enough to cover the cost of rebuilding. There could easily be a situation where the input from one of the leaves increases, but the selectivity of a predicate in the lowest sequence join that handles this input decreases, so the shape of the tree is still optimal, or the savings from rebuilding to a different shape tree would be too small compared to the time spend rebuilding and recalculating intermediate results. The total evaluated cost of a tree should therefore also have a threshold, so no unnecessary rebuilding happens.

The parameters p for single predicate threshold, i for input size threshold and t for total tree threshold are related as can be seen in table 3.2.

t	High	Rarely calculate, often rebuild	Often calculate, often rebuild
	Low	Rarely calculate, rarely rebuild	Often calculate, rarely rebuild
		Low	High
		p and i	

Table 3.2: Relation between threshold parameters for rebuildning.

When the time window of a query is increased, it will contain more results before they time out (assuming the same input speed). This will intuitively also produce more intermediate results that would be discarded in case of a rebuild. One method could be to let the time between checking for optimal shape depend on the length

of the query. This method of implementation is used in this thesis. Other solutions could be checking once a fixed number of events have occured, once a fixed time interval has passed, or checking part of the tree for every input.

Chapter 4

Other Optimizations

The content of this chapter present possibilities for optimizations of the tree model described in chapter 3 especially for pattern queries. First, an elaborate description of the propagation plans briefly presented in section 3.1. Secondly, a few observations that can save some work. Finally, the pass-down optimization which can be used in certain cases, passing information on to an earlier part of the pattern.

4.1 Push-based and Pull-based Propagation Plan

In the article by Mei and Madden, [23], a batch of events is collected before processing of them begins. This could potentially give a higher latency than processing any event as soon as it arrives. In the pull propagation plan, a batch type of approach is used, but the size of the batch is defined by the length between triggering events. As soon as an output could be produced, the tree starts processing.

Looking at figure 4.1, the empty subtree T_3 would prevent any results from being generated, but how are empty subtrees detected? This poses a challenge since there are $n-1$ subtrees overall: All sequence join operators are root in a subtree. Using the pull propagation plan this search is not done explicitly, but only when there is a chance a result can output from the whole tree.

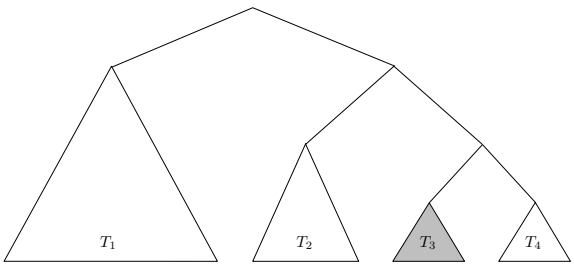


Figure 4.1: Tree with empty subtree to demonstrate why the pull-plan could be beneficial.

If there is no triggering event matching the last event in a pattern, there is no chance of a possible output, and no reason to pass anything on to the parent. If how-

ever a result is being produced this must be input to the parent. Applying this principle recursively, the operators on the entire rightmost path must propagate results whenever possible. All other sequence join operators can utilize the pull propagation plan and only evaluate results when asked to do so.

If the length between triggering events is smaller than the time window of the query, no intermediate results would time out before they would have been used to match something in a pattern, and the push propagation plan could equal the pull propagation plan.

This lazy evaluation technique can also be applied to relational queries, but the advantage of its use has not been tested in this project.

The next observation will prevent input to the rightmost operator, if there is no chance of being able to generate output, thus limiting the chance of a triggering event.

4.2 Input only to relevant operators

From the start of the matching process, there is no need to try to match anything but the first event in the pattern until an initial event has been seen. After this, there is no need to try to match a_3 until something have matched a_2 etc. This can be generalized, when the skip till any selection strategy is used, to whenever an input buffer becomes empty in a time window, there is no need to try and match any later events in the pattern.

In the dynamic tree, there are buffers storing inputs for the rebuilding process, so for the operators located lower in the tree than these buffers (the input operator and the selection operator) must therefore pass on this information from the buffer operator. Particularly if the static predicate does not match the input, the selection operator must ask the buffer operator to discard events that are outdated from the timestamp of the event not matched, and answer if the buffer has become empty.

4.3 Do not Pull when it is not needed

If the event selection strategy is the *skip till any* and the pull plan is in use, one input event can propagate several result tuples to the parent operator. If the parent receives this input from the right child, and the left child is a subtree with the pull plan, there is no need to pull results for each of the inputs received, as they have the same end timestamp.

If the sequence join operator stores the ending timestamp of the last time it pulled results from a child, it can avoid the method call to (getTuples) that would not generate results anyway.

If events are input with the same ending timestamp, this does not conflict with this optimization. Events must be strictly non overlapping, and events with the same ending timestamp do not fulfill this requirement, so it would be impossible to generate results with the events pulled.

4.4 Passing Down Events and Predicates

This technique was developed at the time, when only the static left-deep tree was used to match pattern queries. First, a short motivating example for using pass-down. Assume a left-deep tree for the pattern $a_1 a_2 \dots a_n$ using the pull based propagation plan and there is a predicate $p := (a_i, op, a_n)$, and the frequency of events matching a_n is low[1]. Note that the type of the operator is not limited to equality, as it is when hash tables are used for optimization. Once an input event e arrives that matches a_n, results are pulled from the left part of the tree, including all possible partial matches. At this time, the events in the input queue for a_i could be filtered, since they must be accepted by the predicate involving e.

If the sequence join that handles a_i would know p and e, it could skip propagating results that would be filtered out, when they reach the sequence join that handles a_n. Passing down p and e when making the pull would enable this filtering. The lowest part of the tree handling $a_1 \dots a_{i-1}$ is pulled normally, but the the remaining results are built and passed on in a separate list, not influencing the input buffers of the sequence joins. The technique is named *pass-down* when referenced in the remaining part of this thesis.

An analysis for a left-deep tree can be found in appendix A showing when it is beneficial to apply the tecnique.

4.4.1 Not a left-deep tree

The principle is very simple when the structure is the left-deep tree, as all sequence joins from the root and down know if asked to match a_i and it is not the right input, then a_i is located in the left subtree. With the adaptable tree, there are a few more cases to consider.

An essential observation is that a_i will be in a left subtree of a sequence join on the rightmost path, as the only input located on the rightmost path is a_n and $i \neq n$. The rightmost sequence join can apply the pass-down procedure using algorithm 7 on itself, called with a list containing the incomming event as a tuple matching a_n. Once the left child that contains a_i is found, the results generated are input to the parent as usual, as these tuples are just a standard match to the sub-pattern $a_i \dots a_n$. The passUp algorithm (algorithm 7), uses the pass-down algorithm (algorithm 8) for those sequence join operators not placed on the rightmost path in the tree.

An important difference from the left-deep tree is that the parameterized predicate, relevant for passing down, is not necessarily located in the sequence join that process a_n. This predicate must, when the tree is built, explicitly be placed there instead of in the sequence join higher in the tree that ordinarily would contain it.

4.4.2 Frequency of Triggering Events

Why should the frequency of triggering events be low? When using the pass-down procedure, not all subresults are produced, and since it is not registrered which are,

[1] The time between events that match a_n is greater than the timewindow in the query.

Algorithm 7: PassUp algorithm, for sequence joins on rightmost path.

```
   Input: List list, Predicate p, Event e
 1 if left input is only a_i then
 2 |   match events from left input with Tuples in list using p and e and input
   |   generated tuples normally to parent
 3 |   return
 4 if left child contains a_i then
 5 |   leftlist ← leftChild.passDown(e,p)
 6 |   match events in leftlist with events in rightlist and input normally to
   |   parent
 7 else
 8 |   pull normally from left child and store events in left input list
 9 |   build list newList storing partial results for left inputs matched with
   |   events from list
10 |   parent.passUp(newList,p,e)
```

those intermediate results must be discarded so no doublets suddenly present themselves when the next triggering event arrives. Should two triggering events arrive in the same time-window, the subresults not produced by the arrival of the first of those events could have been used to match results for the second.

Should more than two triggering events arrive in a time-window, the subresults will be missing only once, as when the second arrives in a time-window the complete pull is made in the tree. The worst case scenario is therefore using twice the work.

4.4.3 Several possible predicates

If there are more predicates satisfying the format (a_i, op, a_n) which one should then be selected? There are two factors that influence this decision.

Selectivity The selectivity of the predicate clearly influence the gain. If the selectivity is 1, then no partial results are dropped, and the pass-down technique is useless. If the selectivity is low, many partial results are discarded early, and do not propagate, making the gain high.

Length between i and n If $i = n - 1$ nothing is saved, as all intermediate results are produced and skipped at the same time, as if the pass-down technique was not used. The earlier a_i appears in the pattern the better, since fewer useless intermediate results are produced from the ones that are skipped.

The exact tradeoff between the two parameters is not clear, but this could be a topic for future research.

4.4.4 Correctness of results

In order to argue correctness, all results must be produced, and no invalid results including doublets can be produced.

Algorithm 8: PassDown algorithm, for sequence joins not on rightmost path.

```
   Input: Predicate p, Event e
 1 if left input is only a_i then
 2 |   pull results from right child if needed and store events in right input list
 3 |   match events from right input with events from left input that satisfies p
   |   with e and store result in list l
 4 |   return l
 5 if right input is only a_i then
 6 |   pull results from left child if needed and store events in left input list
 7 |   match events from right input that satisfies p with e and left input and
   |   store result in list l
 8 |   return l
 9 if left child contains a_i then
10 |   leftlist ← leftChild.passDown(e,p)
11 |   pull results from right child if needed and store events in right input list
12 |   match events in right input list with events in leftlist and store result in
   |   list l
13 |   return l
14 else
15 |   rightlist ← rightChild.passDown(e,p)
16 |   pull normally from left child if needed and store events in left input list
17 |   match events in rightlist with events from left input list and store result in
   |   list l
18 |   return l
```

The pass-down procedure uses the entire tree to generate results for the incomming event matching a_n, so all results should be generated with this event being the last. Since only outdated events are removed, when performing the pass-down procedure, no events are missing for a later attempt to produce results. The buffers in the remaining tree will be left as if the passing down was not active.

Since the last event in the pattern is used immediately, no duplicate patterns will ever be produced, as only a new event matching the last in the pattern can initiate another output.

Chapter 5

Analytical Comparison

The first argument of this chapter is one of the best arguments for using a tree-based query plan to handle pattern queries, on the way to a DSMS that can efficiently handle both relational queries and pattern queries, namely that an NFA is not a good choice for relational queries.

This chapter also contains an analythical comparison of the NFA based model and the tree based model for pattern queries, and finally it has cost model considerations for the shape of the tree.

5.1 Evaluating Relational Queries with NFA

Can a NFA based model be used for evaluating relational queries?

Assume we have a relational query like query 4.

Query 4 *Relational query to argue NFA's are not usable for this query type*

```
SELECT *
FROM [RANGE 10 SEC] S1 ; S2 ; ... ; Sn
WHERE ....
```

Since the NFA model only uses loop and forward edges, but the query does not specify any ordering. Even when merging identical suffix' of partial results, the number of states required is n to partial matches of length 1, $\binom{n}{2}$ states to match partial matches of length 2, $\binom{n}{3}$ states to match partial matches of length 3, and so on. The number of states needed totally is at least[1] the sum of a row in Pascal's triangle:

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n$$

The number of edges going forward will be:

$$\sum_{i=0}^{n-1} \binom{n}{i} \cdot (n-i)$$

[1] States generated by a Kleene plus operator is not included in the argument.

since all states corresponding to a length i must point to $n - i$ new states. This sum is by symmetry in the pascal triangle equal to

$$\sum_{i=1}^{n} \binom{n}{i} \cdot i$$

adding both sums results in

$$n \cdot \sum_{i=0}^{n} \binom{n}{i}$$

dividing by two the number of edges is at least $n \cdot 2^{n-1}$.

Having both an exponential number of states and edges in the number of input streams, the use an NFA for relational queries has not received further attention in this thesis.

5.2 The NFA model compared to the tree model

First, the NFA model will be compared to a left-deep tree with the push plan to show that they work in a similar way. Then the left-deep tree is compared to an optimally shaped tree to show that a different shape is sometimes better. These two results combined prove that it is better to use a tree for pattern queries than a (simple) NFA.[2]

5.2.1 The NFA model compared to the Left-deep tree with the push plan and *skip till any* selection strategy

The left-deep tree with the push plan have some glooming similarities with the NFA model. Both use an eager evaluation technique and both generate results from the start of the pattern to the end.

The settings are: Assume $a_1 a_2 \dots a_n$ is the pattern with possible Kleene plus' that the query specifies and that the skip till any event selection strategy is used. In the NFA model, upon input of an event, all active configurations are evaluated, as well as the possibility of starting a configuration from the initial event. This includes evaluating predicates on the edges in the state of the configuration and possibly creation of new configurations (if more than one edge evaluates to true). Work is defined as creating new tuples and evaluating predicates.

Proposition 5.2.1. *The left-deep tree with the push propagation plan does at most as much work as the NFA model for pattern queries with the skip till any selection strategy.*

To show this proposition, two lemmas are needed.

Lemma 5.2.2. *The sum of left inputs in all the sequence joins are the same as the number of configurations in the NFA model.*

[2] If several NFAs are combined, each computing a part of the pattern, this can possibly change this result, but no such work has been published to my knowledge.

Proof. If the NFA model starts a new configuration, the event inputted must match the static predicates on a_1, and the tree will place the event in the left input of the lowest sequence join. Any configuration in the NFA in the second state, must have matched some events. This event must have matched in the tree too, generating a new tuple that has been input to the left buffer of the second lowest sequence join. This principle can iteratively be done for the following states/sequence joins.

A Kleene Plus will not affect the preceding argument, as once the NFA follows the take edge, the tree model will add an input to a left buffer in the sequence join itself or its parent. ∎

Lemma 5.2.3. *The tree at most evaluates the left inputs once, and not more than that.*

Proof. Since we are considering the skip till any case, we can easily see that if the NFA has a configuration in the i^{th} state, it must have a configuration in each of the previous, as it could just have skipped the last event in the configuration. Only if the first event in the pattern times out, these configurations will be removed. The same principle applies to the left-deep tree: It can input the event until it reaches a sequence join that does not have any left input and skip the remaining as no inputs can be located there. ∎

Proof of proposition 5.2.1. Let *con* denote the number of configurations in an NFA and let lef denote the sum of the number of left inputs in the left-deep tree. Lemma 5.2.2 establishes that $con = lef$. Lemma 5.2.3 argues that each left input in the tree model is evaluated at most once, and this includes the relevant predicates which baring the structure must be the same as in the NFA model, so work spent evaluating $con \leq$ work spent evaluating lef. Lemma 5.2.2 used inductively also demonstrates that the tree creates the same number of new left inputs (outputs of the child sequence join) as the NFA creates new configurations. ∎

Can the tree then be more efficient than the NFA?

The left-deep tree can potentially save some work compared to the NFA based model. If a static predicate evaluates to false before reaching a sequence join, all the left inputs in the sequence joins are not considered. For each of them, the NFA will have to evaluate the static predicate (a partition of the configurations into states, so static predicates can be evaluated for each partition, could possibly be an optimization of the NFA model).

The NFA model is only described with inputs being non-overlapping, thus having only a single timestamp. Unless the NFA model is modified with a partition on time, it must examine all active configurations upon input, where as the tree in each sequence join can potentially stop once the starting timestamp of the new event has been surpassed by the end timestamp of the event in the left input.

5.2.2 Better shape than left-deep for the tree model?

Assume a pattern $a_1 a_2 \dots a_i a_{i+1} \dots a_n$, and let $i = \lceil \frac{n}{2} \rceil$ using the skip till any event selection strategy and having a predicate (a_i, op, a_{i+1}) which is highly selective.

The NFA (and the left-deep tree) will in this setting generate half the pattern before being able to evaluate this predicate and hold the configurations in the corresponding state until they timeout.

Consider the tree in figure 5.1 and the pull propagation plan.

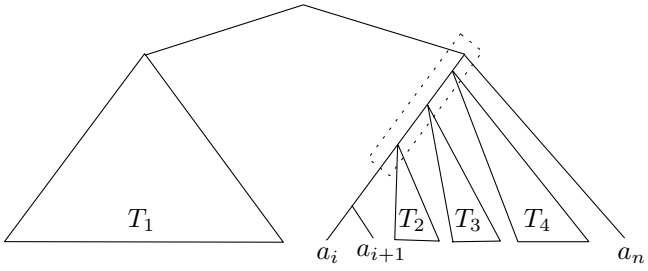


Figure 5.1: Tree with empty subtree used to demonstrate why the pull-plan can be beneficial.

In this setting, once a triggering event arrives, the sequence joins on the path in the dotted box are pulled for results, until the join that handles a_i and a_{i+1} which in most cases will return no results, causing the tree to not pull results from the subtrees T_1, T_2, T_3 and T_4. Assuming the subtrees T_2, T_3 etc. each have roughly half of the inputs in their subtree, the path in the dotted box will have logarithmic height, so the number of joins visited in an attempt to produce results can in this case be logarithmic compared to the linear number of states visited in the NFA model.

It is evident from this example that the predicates can have a large influence on which tree shape will be optimal. Proposition 5.2.1 argues that the left-deep tree is at least as good as the NFA for evaluating pattern queries, and this example showing that, in some cases, a different shape is better than the left deep, the NFA would not be the best choice for pattern queries.

5.2.3 The NFA model compared to the Left-deep tree with push plan and strict selection strategy

Assuming that the strict selection strategy is used, the NFA can now have states that have no active configurations between those that have active configurations, as the NFAs ignore edge will evaluate to false and not leave a copy of all the partial matches in the early states. However, since the length of the pattern is constant, n, the two models will, when an input arrives, both check if the input matches a_1 and match the incoming event against all partial results. The tree model, not knowing which joins have empty left inputs, could check $n - 1$ sequence join operators with extra. This being constant work, the two models will perform asymptotically the same.

5.3 Cost Model Observation for Dynamic Tree

This is an on-line problem, as the total data set is unknown at the time of processing. This also makes it impossible to know the optimal shape of the tree for the input yet to come, but a reasonable estimate is a shape optimal for the input already seen.

In order to be able to evaluate which shape of a tree is better than another, we need a more formal definition of work done. The work done evaluating static predicates and buffering the input for rebuilding will be the same regardless of the optimal shape of the tree. This cost is therefore omitted. The shape that can change is the way the sequence join operators are combined into a specific tree shape, the work that must be included is therefore the work done in the part of the tree that are composed by sequence join operators.

We estimate the work in a sequence join operator as:

$$cost(op) := |\text{left input}| * |\text{right input}| * \frac{1}{2}$$

where the $\frac{1}{2}$ comes from the time constraint that one event must come before the other. This is a loose approximation, as they could be overlapping.

The size of the input from left or right must be within a specified time, and the natural choice is the size of the time window defined in the query. If the input comes as events, then the size can be measured from the buffer storing events for rebuilds. If, however, the input comes as tuples, the child must be a sequence join, and the size of the input can be estimated by the size of the output of the child:

$$output(op) := |\text{left input}| * |\text{right input}| * \frac{1}{2} * \text{selectivity}(op)$$

The $\frac{1}{2}$ again comes from the time constraint. The selectivity of the operator is calculated by the relevant parameterized predicates, where relevant is described in section 3.5.4. It is calculated as:

$$\text{selectivity}(op) := 1 \quad \text{if } \nexists \text{ relevant predicates}$$

and

$$\text{selectivity}(op) := \prod_{p \in \text{ relevant predicates}} \text{selectivity}(p) \quad \text{if } \exists \text{ relevant predicates}$$

.

The cost for a tree now be defined as the

$$cost(T) := \sum_{op \in \text{sequence joins in T}} cost(op)$$

5.3.1 Optimality of subtrees

In order to argue that a tree has an optimal structure, a small lemma from [23] is needed.

Lemma 5.3.1. *If a tree has the optimal shape, so will all subtrees*

Proof. Assume we have an optimally shaped tree T with a non-optimal subtree T_r with the cost of the subtree c_r. Let the cost of an optimally shaped subtree T'_r with the same nodes as T_r be c'_r. Since the new subtree is optimal, we have that $c'_r < c_r$, and the cost of a new tree T', made from T with the subtree T_r replaced by T'_r, will now also have lower cost than T, contradicting the assumption that T was optimal. ∎

5.3.2 Total tree evaluation

Lemma 5.3.1 claims that the problem of finding the optimal tree shape has optimal substructures. If all possible tree shapes are tested, then a small subtree would occur in several of these shapes, so the problem also has overlapping subproblems. These are the conditions for using dynamic programming to try all legal tree shapes[3].

Algorithm 4 uses dynamic programming to calculate the optimal tree structure, by calculating the optimal shape of all subtrees of size $2, 3, \ldots n$, registrering where the root is placed in the best solution.

The change in the algorithm from [23] is the parameters to getSelectivity, where Mei and Madden only supply the root. This is not enough information to establish which predicates are to be evaluated in the root of the subtree in question. Not knowing the the size and which inputs are included in the tree could cause the wrong parameterized predicates to be included in the calculation, either inclusion of predicates that cannot be evaluated, or predicates that are already evaluated in a subtree and thus generate a non-optimal tree shape.

[3] A tree shape will be legal, if the inputs for the leaves are in the correct order in an in-order treewalk.

Chapter 6

Experimental Work

This chapter begins with a presentation of the technical details for the equipment used in the experiments. The focus is on how the NFA based model and the tree based model perform on pattern queries, so no experiments have been performed on relational queries.

The tests reveal the performance when the frequency of event are changed, the influence of the time window size, if the structure of the query is important for the result, among this the Kleene star and how the models perform when the input data has characteristics that change over time. The last test demonstrate the effect of the pass-down optimization.

6.1 Technical specification

The implementation has been done using the JAVA programming language from Sun version 1.6, running on an Ubuntu 9.10 Linux operating system, kernel version 2.6.31-16-generic.

The scanner was created using the tool JFlex [21], and the parser was made using CUP [24]. The *Eclipse IDE* was used during development, *JUnit* used for testing, and *R* for producing graphs.

All experiments have been performed on a Intel dual core CPU 3.00GHz with 2 GB RAM, where the JAVA VM was allowed to use 1 GB. The implementation is threaded such that the inputting process(es) runs separate threads, and the actual DSMS runs a single thread.

The Range tree implementation has been done by stud.scient Magnus Gausdal Find as a project in the course "Geometric algorithms" and modified to be able to handle data. The MyList implementation is a modified version of the JAVA API's `Linked List` in order to make concatenation $\boldsymbol{O}(1)$. All other code has been developed by the author.

In all of the tests, the events are allowed to enter the system as fast as possible, generated in a separate thread. In order to have the same work done in all cases, the events are timestamped with one second between each. All queries use the *"select all"* projection, as the conversion from tuple to event is the same for the NFA model and the tree model, so the two models would perform the same work.

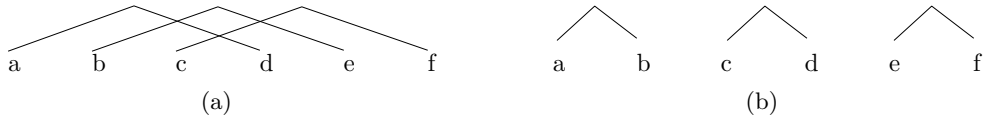


Figure 6.1: Overlapping and non-overlapping predicates.

Tests have been conducted on which predicate location method (section 3.5.4) performs best, but the difference was not measurable. In all tests described, the "precomputed cube" method is used. A pattern of length six has been chosen for testing, as this has a total of 50 different tree shapes. Had a pattern of length three been chosen, only two shapes would have been possible, likely making the left-deep tree optimal in half the situations.

The code for the implementation can be found here:
`http://www.imada.sdu.dk/~kok04/speciale/code.zip`

6.2 Relative Frequency

This test will show how the NFA and the tree will perform when the frequency of events in a pattern are uneven. A pattern of length six will be used, and the data chosen at random, so no particular shape of a tree will be favored. Query 5 and 6 will be used in this test. Query 5 has overlapping parameterized predicates, see figure 6.1(a), and query 6 has non-overlapping predicates, figure 6.1(b). Both queries will be used with time windows of 20 and 40 seconds (i.e. $X \in \{20, 40\}$).

In these tests, the schema of the input is $(name, price)$ and $price \in_R \{0, 1\}$. The selectivity of the parameterized predicates is expected to be 50%. The inputs are grouped in pairs, so IBM and *Google* will be handled as the first part of the pattern, Sun and $Oracle$ as the middle, and $Microsoft$ and $Yahoo$ as the last. All these tests have been run with 100.000 events, averaged over ten different seeds.

Query 5 *Overlapping parameterized predicates for frequency testing*

```
SELECT *
PATTERN abcdef
FROM [RANGE X SEC] S : a,b,c,d,e,f
WHERE SKIP_TIL_ANY(a,b,c,d,e,f) {
     a.name = "IBM" AND b.name = "Google" AND
     c.name = "Sun" AND d.name = "Oracle" AND
     e.name = "Microsoft" AND f.name = "Yahoo" AND
     a.price = d.price AND b.price = e.price AND
     c.price = f.price}
```

Query 6 *Non-overlapping parameterized predicates for frequency testing*

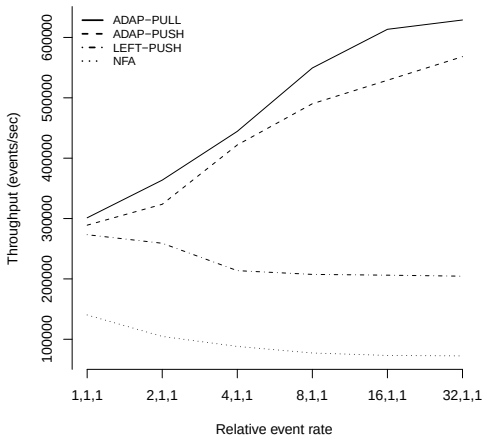
```
SELECT *
PATTERN abcdef
FROM [RANGE X SEC] S : a,b,c,d,e,f
WHERE SKIP_TIL_ANY(a,b,c,d,e,f) {
a.name = "IBM" AND b.name = "Google" AND
c.name = "Sun" AND d.name = "Oracle" AND
e.name = "Microsoft" AND f.name = "Yahoo" AND
a.price = b.price AND c.price = d.price AND
e.price = f.price }
```

In figure 6.2, the first two events in the pattern are frequent, with the relative frequency depicted on the x-axis, and the throughput on the y-axis. In figure 6.3 it is the middle events in the pattern that are frequent, and in figure 6.4 it is the last events that appear frequently.

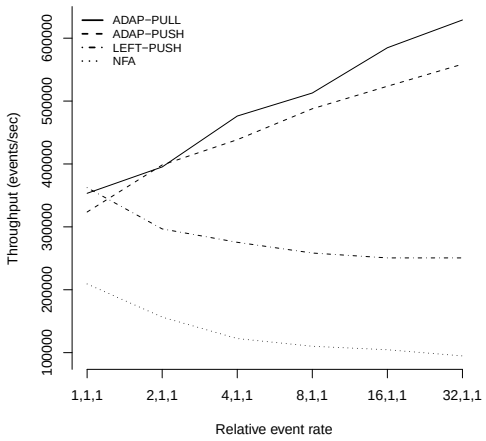
Figure 6.2 clearly shows that the NFA spends too much time generating configurations for the first states that later time out. The throughput for the NFA is poor compared to the adaptive tree. Furthermore, it is obvious that when the triggering event is rare, the pull-based approach is better than the push-based, as some of the events from the first states have timed out before being processed. The throughput is better for the shorter window size, which is expected from the smaller number of active events in the time window.

Figure 6.3 shows a more even match between the tree based and the NFA based model. Please note that when data is uniformly distributed, then the tree model is superior. Only when the frequency of the data is quite skewed, the NFA based implementation can compete with the tree based.

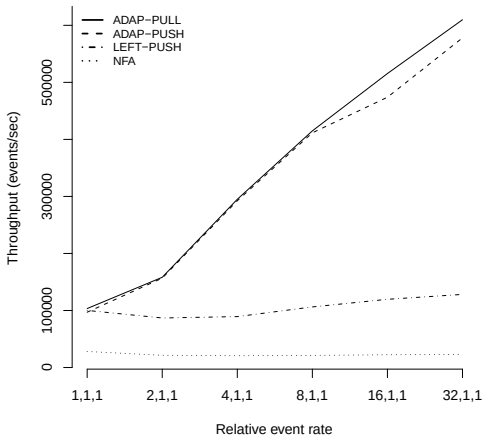
In figure 6.4 the trend continues: Only when the frequency of the data is very skewed does the NFA model perform slightly better than the tree based. This is not unexpected, as this is the best case scenario for the NFA model. Having very few configurations in the first part of the query makes very few intermediate results that will time out and be discarded. Asymptotically the two models look similar, unlike in figure 6.2 where the tree model is much better.



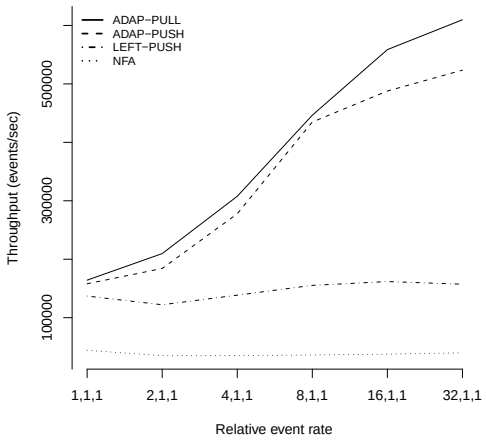
(a) Query 5 with 20 second time window



(b) Query 6 with 20 second time window

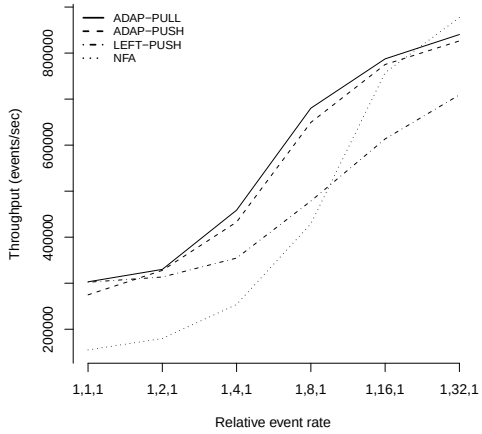


(c) Query 5 with 40 second time window

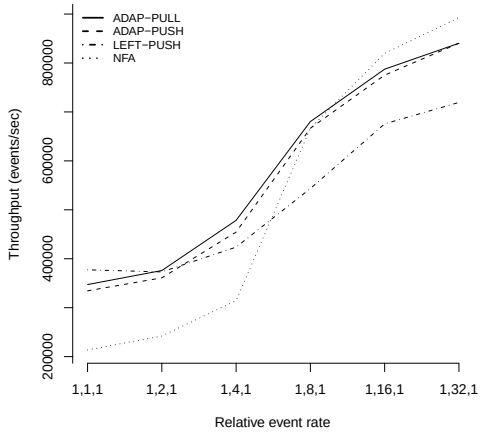


(d) Query 6 with 40 second time window

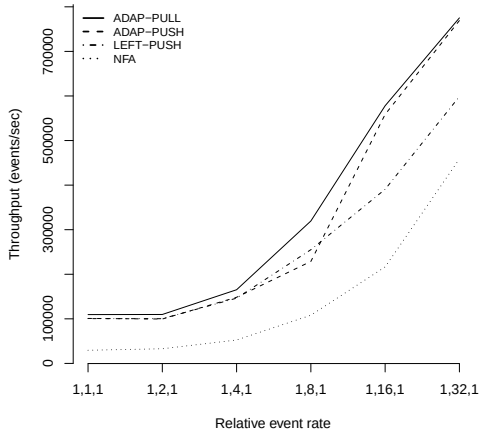
Figure 6.2: Throughput when the first input of a query occurs frequently



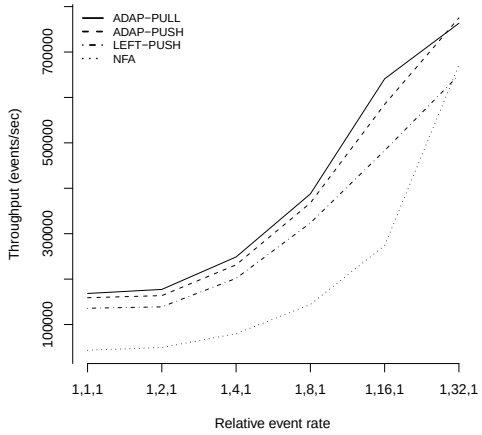
(a) Query 5 with 20 second time window



(b) Query 6 with 20 second time window

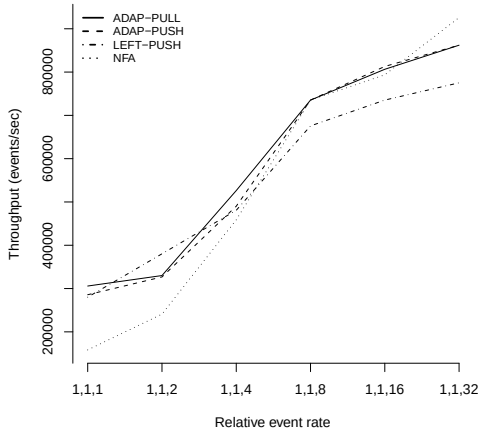


(c) Query 5 with 40 second time window

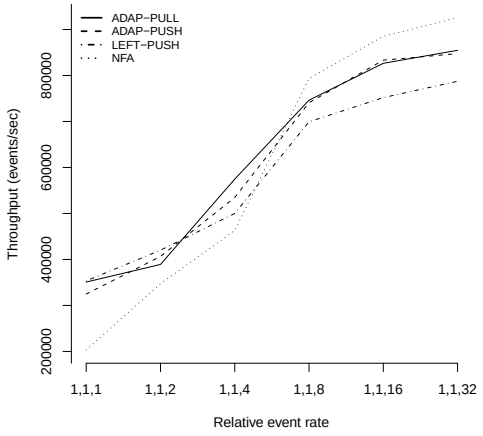


(d) Query 6 with 40 second time window

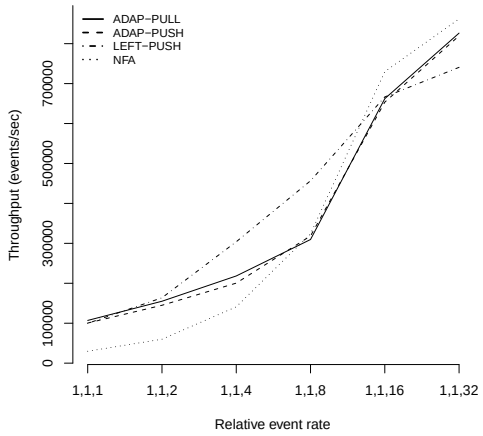
Figure 6.3: Throughput when the middle input of a query occurs frequently



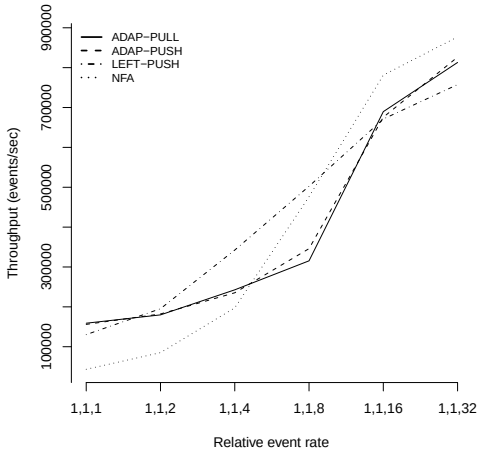
(a) Query 5 with 20 second time window



(b) Query 6 with 20 second time window



(c) Query 5 with 40 second time window



(d) Query 6 with 40 second time window

Figure 6.4: Throughput when the last input of a query occurs frequently

6.3 Query Structure

In this experiment, the structure of the query will be examined. Here, structure is the way the parameterized predicates relate to each other. Three structures are examined: In the first, all parameterized predicates relate to the initial event (6.5(a)), in the second, all relate to the fourth event in the pattern (6.5(b)), a middle one, and in the last, all relate to the triggering event (6.5(c)).

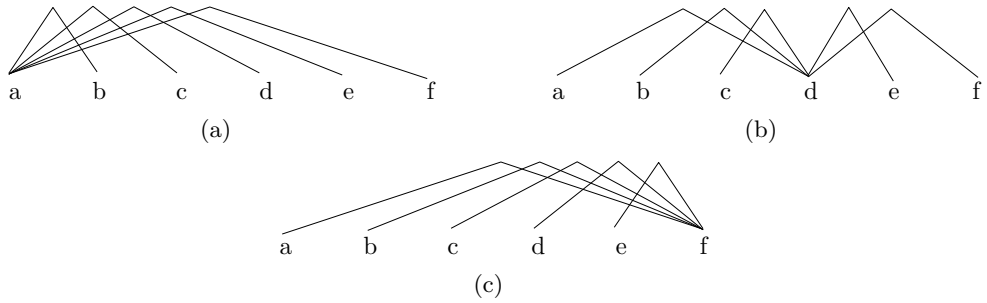


Figure 6.5: Overlapping and non-overlapping predicates.

The queries used are similar to query 5, with the time window $X \in \{20, 40\}$ but with the parameterized predicates changed to the ones shown in query 7, 8, and 9. The input data is again randomly generated as $(name, price)$, and all measurements have been done with 100.000 events and averaged over 10 seeds.

Figure 6.6(a) shows the results when the time window is 20 seconds, with the three different queries on the x-axis. Figure 6.6(b) displays the results with the time window being 40 seconds.

Query 7 *All predicates relate to the initial event*

```
a.price < b.price AND a.price < c.price AND
a.price < d.price AND a.price < e.price AND
a.price < f.price
```

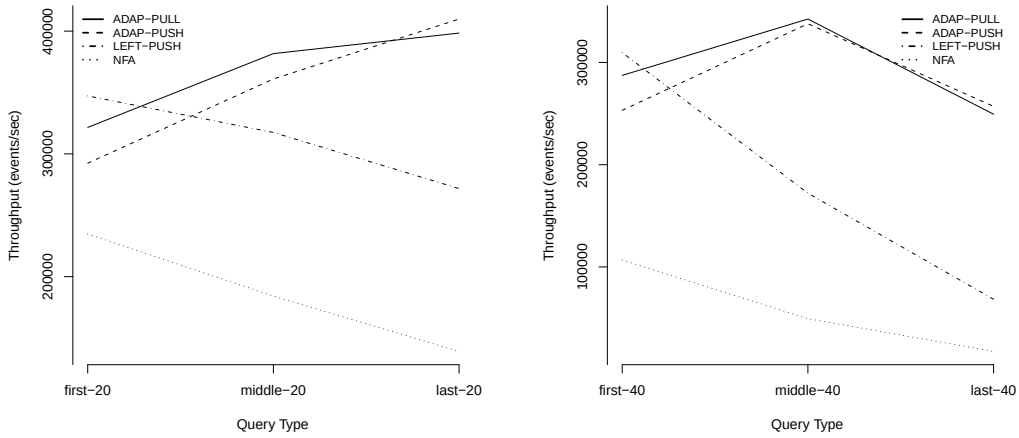
Query 8 *All predicates relate to the fourth event*

```
a.price < d.price AND b.price < d.price AND
c.price < d.price AND d.price < e.price AND
d.price < f.price
```

Query 9 *All predicates relate to the triggering event*

```
a.price < f.price AND b.price < f.price AND
c.price < f.price AND d.price < f.price AND
e.price < f.price
```

Both figure 6.6(a) and 6.6(b) show that the NFA model is inferior, especially when the initial event is not limited by any parameterized predicates. This creates a large



(a) With 20 second time window

(b) With 40 second time window

Figure 6.6: Throughput when the first, middle or last event is involved in all parameterized predicates, i.e. the structure of the query.

family of queries, where the tree model would be preferable. It is also clear from these figures that the left-deep shape is not always the best choice, as the adaptive tree is superior.

6.4 Time Window

This experiment will show if the size of the window has any effect on the pattern query evaluation. Queries 10 and 11 are used, with $X \in \{5, 10, 15, 20, 25, 30, 40, 45, 50\}$. These queries have independent parameterized predicates i.e. the attributes of each parameterized predicate is only involved in that single predicate. One query has overlapping and one has non-overlapping predicates. The data is randomly generated, now with the schema $(name, price, type, volume)$.

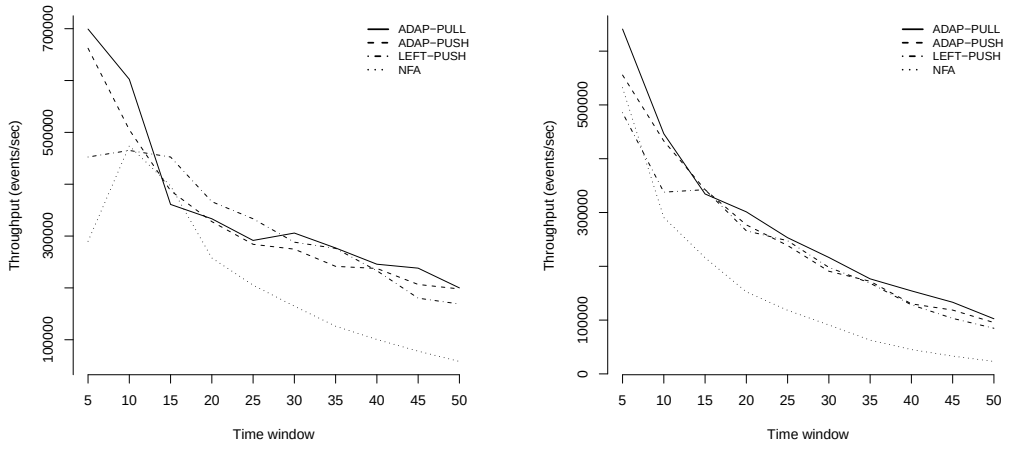
Query 10 *Query to test if the size of the time window has an effect (non-overlapping predicates).*

```
SELECT *
PATTERN abcdef
FROM [RANGE X SEC] S : a,b,c,d,e,f
WHERE SKIP_TIL_ANY(a,b,c,d,e,f) {
a.name = "IBM" AND b.name = "Google" AND
c.name = "Sun" AND d.name = "Oracle" AND
e.name = "Microsoft" AND f.name = "Yahoo" AND
a.volume < b.volume AND c.price < d.price AND
e.type < f.type }
```

Query 11 *Query to test if the size of the time window has an effect (overlapping predicates.*

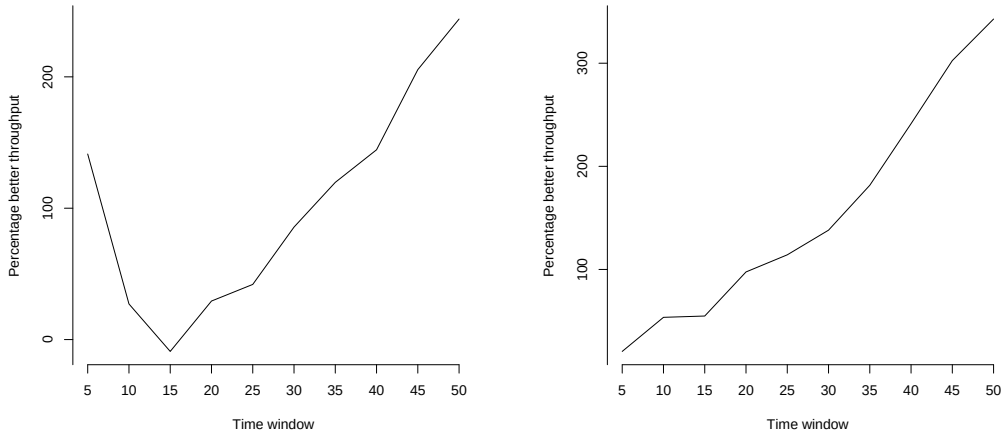
```
SELECT *
PATTERN abcdef
FROM [RANGE X SEC] S : a,b,c,d,e,f
WHERE SKIP_TIL_ANY(a,b,c,d,e,f) {
a.name = "IBM" AND b.name = "Google" AND
c.name = "Sun" AND d.name = "Oracle" AND
e.name = "Microsoft" AND f.name = "Yahoo" AND
a.volume < d.volume AND c.price < f.price AND
b.type < e.type }
```

Figure 6.7(a) displays the result of query 10 and figure 6.7(b) shows the result of query 11. The throughput for both models decreases as the window size increases. Why the NFA model performs well with the short time windows and the overlapping query is a bit curious. The throughput is however clearly the best for the tree model. The graphs in figure 6.7 shows percent increase of throughput with the tree based model compared to the NFA based model. This is quite substantial, especially for long time windows.



(a) Overlapping (b) Non overlapping

Figure 6.7: Influence of window size on throughput with random data.



(a) Overlapping (b) Non overlapping

Figure 6.8: Percentage of throughput, that the pull based adaptive tree model is better than the NFA model using random data.

6.5 Kleene Star

To test how pattern queries with repetition, Kleene star, influence the throughput, queries 12 having one Kleene star and 13 having two are used. The predicate on the event with the Kleene star is expecteed to be 0.1, as the number of results would otherwise become too large to handle in the memory available.

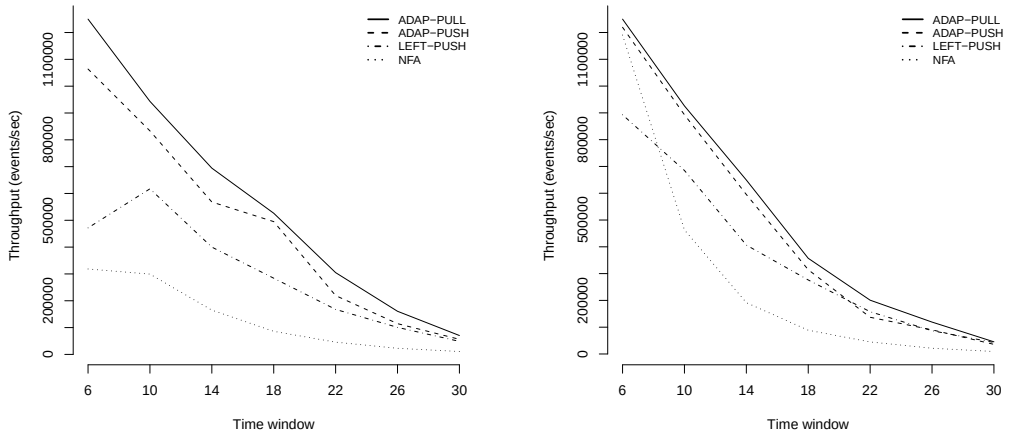
Query 12 *Query to test a single Kleene star in a pattern.*

```
SELECT *
PATTERN ab+cdef
FROM [RANGE X SEC] S : a,b,c,d,e,f
WHERE SKIP_TIL_ANY(a,b,c,d,e,f) {
a.name = "IBM" AND b.name = "Google" AND
c.name = "Sun" AND d.name = "Oracle" AND
e.name = "Microsoft" AND f.name = "Yahoo" AND
a.volume < d.volume AND c.price < f.price AND
b.price < 100 }
```

Query 13 *Query to test a two Kleene stars in a pattern.*

```
SELECT *
PATTERN ab+cde+f
FROM [RANGE X SEC] S : a,b,c,d,e,f
WHERE SKIP_TIL_ANY(a,b,c,d,e,f) {
a.name = "IBM" AND b.name = "Google" AND
c.name = "Sun" AND d.name = "Oracle" AND
e.name = "Microsoft" AND f.name = "Yahoo" AND
a.volume < d.volume AND c.price < f.price AND
b.price < 100 AND e.price < 100 }
```

Figure 6.9 shows a graph of the throughput of the two queries, with the size of the time window on the x-axis. There is no indication of the tree based models not being able to handle the Kleene plus in a satisfactory manner. The throughput for the tree based models is better than for the NFA, and the pull based propagation strategy seems to perform slightly better than the push based.



(a) One Kleene star

(b) Two Kleene stars

Figure 6.9: Throughput of the different implementations for queries including Kleene star

6.6 Data Changing Over Time

The random data used in the preceeding experiments, does not favor any parameterized structure, since once a structure for the tree has been selected, checking for a different structure will be unnecessary. In this experiment, the data will change over time to having three different characteristics. The queries used are 10 and 11.

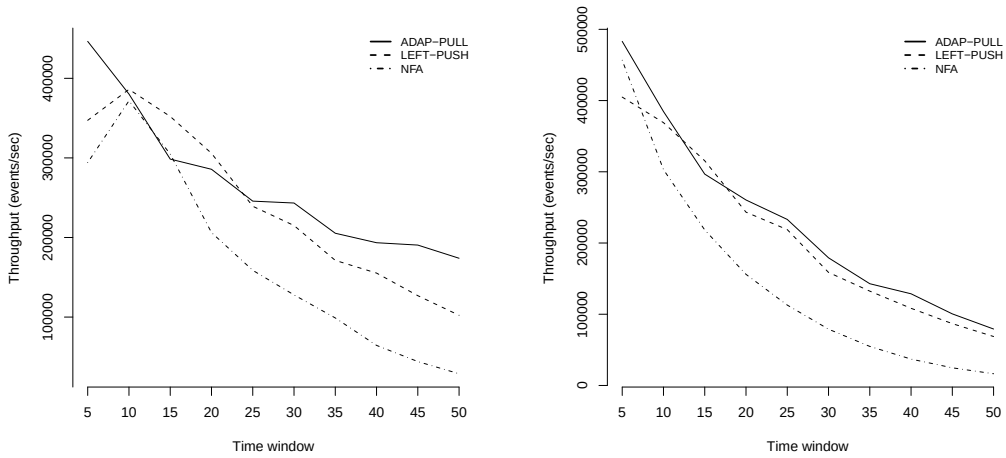
Unless specified below, $volume, type, price \in_R [0 : 1000]$

1. **batch** If $name$ = "IBM" then $volume$ is 950. For query 10 the predicate $a.volume < b.volume$ will have an expected selectivity of 0.05, and for query 11 the predicate $a.volume < d.volume$ will have an expected selectivity of 0.05.

2. **batch** If $name$ = "$Google$" then $type$ = 950, so in query 11 the selectivity for the predicate $b.type < e.type$ will have an expected selectivity of 0.05. If $name$ = "$Yahoo$" then $type$ = 50 causing the predicate $e.type < f.type$ to get an expected selectivity of 0.05 in query 10.

3. **batch** If $name$ = "Sun" then $price$ = 950. In query 10 the predicate $c.price < d.price$ will have an expected selectivity of 0.05, and for query 11 the predicate $c.price < f.price$ will have an expected selectivity of 0.05.

Each batch will consist of 5000 events, and predicates not mentioned above will have an expected selectivity of 0.50. The batches are then repeated in circular order until a total of 100.000 events have been processed.

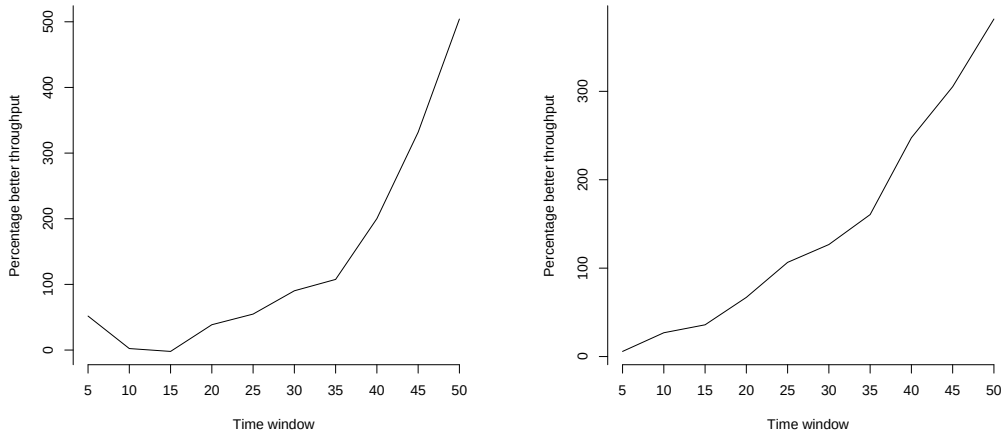


(a) Overlapping (b) Non overlapping

Figure 6.10: Influence of window size on throughput with data that changes characteristics over time.

Figure 6.10 shows the results from the changing data test. Especially with the overlapping predicates does the adaptive tree perform better than the left-deep tree and the NFA.

It is clear, when the percentages of figure 6.7(a) are compared to the percentages of figure 6.10(a) that the adaptive tree outperforms the NFA model, once the data has features that would make different tree shapes optimal.



(a) Overlapping (b) Non overlapping

Figure 6.11: Percentage of throughput that the pull based adaptive tree model is better than the NFA model with data that changes characteristics over time

6.7 Pass-down Optimization

The last test in this thesis will inspect the pass-down optimization described in section 4.4. The two queries 14 and 15 will be used, with data having the schema $(name, type, volume, price)$. The optimization should only have an effect when the time between triggering events are longer than the timewindow. This can be acomplished by the last event in the pattern being rare, so the data in this test is skewed, making the last events increasingly infrequent.

Each query has run with ten seeds, each time with 100.000 events input.

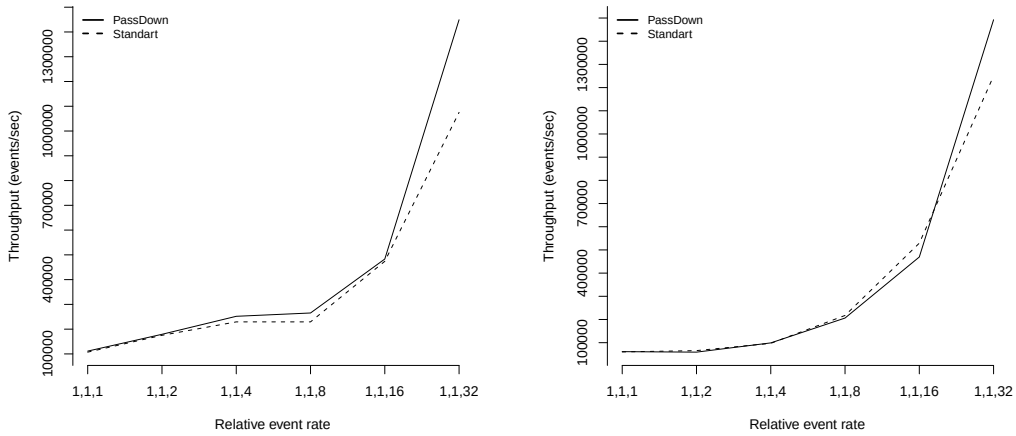
Query 14 *Query to test the pass-down optimization.*

```
SELECT *
PATTERN abcdef
FROM [RANGE 50 SEC] S : a,b,c,d,e,f
WHERE SKIP_TIL_ANY(a,b,c,d,e,f) {
a.name = "IBM" AND b.name = "Google" AND
c.name = "Sun" AND d.name = "Oracle" AND
e.name = "Microsoft" AND f.name = "Yahoo" AND
a.volume < d.volume AND c.price < e.price AND
b.type < f.type }
```

Query 15 *Query to test the pass-down optimization (large output).*

```
SELECT *
PATTERN abcdef
FROM [RANGE 50 SEC] S : a,b,c,d,e,f
WHERE SKIP_TIL_ANY(a,b,c,d,e,f) {
a.name = "IBM" AND b.name = "Google" AND
c.name = "Sun" AND d.name = "Oracle" AND
e.name = "Microsoft" AND f.name = "Yahoo" AND
b.type < f.type }
```

The result of the test is displayed in figure 6.12. As expected, the pass-down optimization performs better than the standard tree with the pull propagation plan, when the last event in the pattern does not appear too frequently.



(a) Query 14 (b) Query 15

Figure 6.12: Influence of the pass-down optimization on skewed data.

Chapter 7

Conclusion

Two different approaches for evaluating pattern queries in a DSMS have been studied and compared theoretically and experimentally. The NFA based approach known from litterature and a tree based approach partly developed in this thesis.

The theoretical results show that the NFA based model is not suitable for handling relational queries. Furthermore, the tree based evaluation should perform at least as well as the NFA based evaluation on pattern queries.

A prototype DSMS has been developed and used for testing the theoretical results. The experiments performed consolidate the theoretical results: The tree model performs at least as well as the NFA based.

Optimizations have been developed, implemented, and tested showing promising results for the tree based model.

7.1 Suggestions for Further Study

This thesis has left some topics to be considered.

- Allowing parenthesis with Kleene Plus, so more than a single event can be repeated. It is quite possible that the tree based model could handle this elegantly, by allowing sequence joins with tuple input to handle Kleene Star.
- How can the pull based approach be included in the tree shape calculation? And should the pass-down optimization be considered in the calculation?
- How the two criteria in the pass-down procedure trade off?

Appendix A

Pass-down in Left-deep Tree

Assume the root has a parameterized predicate (a_g, op, a_n) having selectivity $\lambda_{g,n} \in [0; 1]$. It is then clear that the tree $a_1 \dots a_{g-1}$ is not affected by the pass-down strategy, so the cost of this part of the tree is the same for both strategies and can therefore be ignored.

The intervals to consider is between events that match a_n, as this is the only times where output is produced, the time of such an interval is denoted f.

First some basic thoughts about comparing the two strategies. Consider an input sequence: $e_i|e_{i+1}e_{i+2}\dots e_j|e_{j+1}e_{j+2}\dots e_k|e_{k+1}$ where e_i, e_j and e_k matches a_n. If the time between e_j and e_k is less than the timespan of the query, the buffers $|a_{g+1}|r \cdots |a_{n-1}|r$ could still have some of the events $e_i \cdots e_j$ which isn't the case for the pull strategy, which have emptied those buffers at the time of arrival of e_j, propagating all possible events upwards. The two cases are treated separately. Note that the part of the tree $a_1 \cdots a_{g-1}$ can be ignored, as this will be exactly the same for both strategies. The following arguments only apply to the part of the tree where the two strategies are different. Notation for the arguments below is found in table A.1 Note that there is no explicit notation for the output, since the output of the operator handling a_i is the same as $|a_{i+1}|_l$.

To compare the cost, the work spent for producing output is estimated, thus checking how many comparisons the tree makes.

$\|a_g\|_r$	The size of the right input buffer of the operator treating a_g
$\|a_g\|_l$	The size of the left input buffer of the operator treating a_g
λ_{a_g}	Selectivity for the static predicate(s) on a_g
λ_{a_g,a_n}	Selectivity for the parameterized predicate between a_g and a_n
r	rate of events per seconds in the input
d	Duration of the time window in the query
f	Time between two events matching a_n

Table A.1: Definitions for the analysis

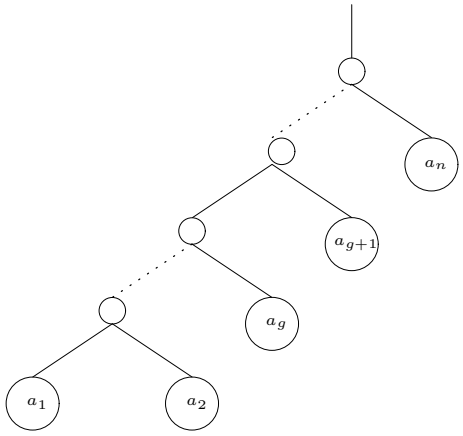


Figure A.1: Example of tree for reference

Firstly, the case where $f \geq d$

Regardless of the strategy used (of pull and pass-down) the following estimate can be made assuming uniform rate of events and uniform distribution of the data:

$$|a_s|_r \approx d \cdot r \cdot \lambda_{a_s} \quad \forall\ 1 < s < n$$

For the pull strategy the following approximation can be made:

$$|a_{g+1}|_l \approx \frac{1}{2}|a_g|_l \cdot |a_g|_r \cdot \prod_{i=1}^{g-1} \lambda_{a_i,a_g}$$

where the term of $\frac{1}{2}$ comes form the time constraint of two events (one must come before the other).

For the pass-down strategy the approximation becomes:

$$|a_{g+1}|_l \approx \frac{1}{2}|a_g|_l \cdot |a_g|_r \cdot \lambda_{a_g,a_n} \cdot \prod_{i=1}^{g-1} \lambda_{a_i,a_g}$$

In the layer above in the tree, for both strategies, the approximation of the left input becomes:

$$|a_{g+2}|_l \approx \frac{1}{2}|a_{g+1}|_l \cdot |a_{g+1}|_r \cdot \prod_{i=1}^{g} \lambda_{a_i,a_g+1}$$

The only difference is the term λ_{a_g,a_n} hidden in the left input.

Adding all layers from the a_g operator and above together, the factor that sets the two strategies apart is the constant λ_{a_g,a_n}, in favor of the pass-down strategy. To sum up, if at most one output triggering event comes in the span of the query time window, the pass-down strategy will perform better than the pull (although not asymptotically).

Secondly, the case where $f < d$.

Intuitively for pull, the input queues from the right are smaller, since they can be emptied when a pull is performed, propagating all results in the tree.

Since the pull now is made more often, the size of the right input queues can be estimated differently

$$|a_s|_r \approx f \cdot r \cdot \lambda_{a_s} \quad \forall\ 1 < s < n$$

For the pull strategy:

$$|a_{g+1}|_l \approx \frac{1}{2}|a_g|_l \cdot f \cdot r \cdot \lambda_{a_g} \cdot \prod_{i=1}^{g-1} \lambda_{a_i,a_g}$$

and

$$|a_{g+2}|_l \approx \frac{1}{2}|a_{g+1}|_l \cdot f \cdot r \cdot \lambda_{a_{g+1}} \cdot \prod_{i=1}^{g} \lambda_{a_i,a_{g+1}}$$

For the pass-down strategy:

$$|a_{g+1}|_l \approx \frac{1}{2}|a_g|_l \cdot d \cdot r \cdot \lambda_{a_g} \cdot \lambda_{a_g,a_n} \cdot \prod_{i=1}^{g-1} \lambda_{a_i,a_g}$$

and

$$|a_{g+2}|_l \approx \frac{1}{2}|a_{g+1}|_l \cdot f \cdot r \cdot \lambda_{a_{g+1}} \cdot \prod_{i=1}^{g} \lambda_{a_i,a_{g+1}}$$

From this it is clear that the difference between the two strategies is a term of f in the pull from a term of $d \cdot \lambda_{a_g,a_n}$.

The conclusion is therefore, letting q be a constant determining the difference in implementation, if $f < q \cdot d \cdot \lambda_{a_g,a_n}$ the pull strategy should be used. Otherwise it will be beneficial to use the pass-down strategy.

List of Figures

List of Tables

List of Algorithms

Index

Bibliography

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 147–160. ACM New York, NY, USA, 2008.

[3] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. *a book on data stream management edited by Garofalakis, Gehrke, and Rastogi*, 2004.

[4] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical report, Technical report, Stanford University, 2003. http://dbpubs.Stanford.edu/pub/2003-67.

[5] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 336–347. VLDB Endowment, 2004.

[6] R. Avnur and J.M. Hellerstein. Eddies: Continuously adaptive query processing. *ACM SIGMOD Record*, 29(2):261–272, 2000.

[7] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM New York, NY, USA, 2004.

[8] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[9] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. CIDR, 2003.

[10] J. Chen, D.J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. *ACM SIGMOD Record*, 29(2):390, 2000.

[11] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. Computational Geometry Algorithms and Applications. 2000.

[12] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. A General Algebra and Implementation for Monitoring Event Streams. 2005.

[13] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards Expressive Publish/Subscribe Systems. *LECTURE NOTES IN COMPUTER SCIENCE*, 3896:627, 2006.

[14] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W.M. White. Cayuga: A general purpose event monitoring system. In *Proceedings of the third Biennial Conference on Innovative Data Systems Research, Asilomar*, 2007.

[15] Y. Diao, N. Immerman, and D. Gyllstrom. Sase+: An agile language for kleene closure over event streams. Technical report, Citeseer, 2007.

[16] Luping Ding and Elke A. Rundensteiner. Evaluating window joins over punctuated streams. In *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 98–107, New York, NY, USA, 2004. ACM.

[17] L. Golab and M.T. Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

[18] L. Golab and M.T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, page 511. VLDB Endowment, 2003.

[19] Lukasz Golab, Theodore Johnson, Nick Koudas, Divesh Srivastava, and David Toman. Optimizing away joins on data streams. In *SSPS '08: Proceedings of the 2nd international workshop on Scalable stream processing system*, pages 48–57, New York, NY, USA, 2008. ACM.

[20] Lukasz Golab and M. Tamer Özsu. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 658–669, New York, NY, USA, 2005. ACM.

[21] JFlex homepage. `http://jflex.de/`.

[22] J. Li, D. Maier, K. Tufte, V. Papadimos, and P.A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 311–322. ACM New York, NY, USA, 2005.

[23] Y. Mei and S. Madden. ZStream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 193–206. ACM, 2009.

[24] Homepage of CUP. `http://www2.cs.tum.edu/projects/cup/`.

[25] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal bloom filter replacement. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 823–829, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[26] Stream Query Repository. `http://infolab.stanford.edu/stream/sqr/`.

[27] P.A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):555–568, May-June 2003.

[28] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 285–296. VLDB Endowment, 2003.

[29] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418, New York, NY, USA, 2006. ACM.