Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Adaptive Data Push/Pull for the DejaVu Pattern Matching System

Master Thesis

by Patrick Lau

ETH Zurich, Systems Group

Department of Computer Science

31.10.2009

supervised by Prof. Nesime Tatbul, Nihal Dindar

Abstract

In traditional database systems where the query processor operates by "pulling" data, a different approach of input handling is needed to deal with real-time requirements complex event processing (CEP) applications do request. Thus CEP Systems, which are built over a traditional database framework such as DejaVu [11], need to support a "pushing" mechanism where incoming data is being pushed to and processed by the query processor on-the-fly. Since the data rate may be extremely high or bursty, switching from "Push" to "Pull" reduces the "pressure" for the query processor.
The goal of this thesis is to provide such a "pushing" mechanism for DejaVu and apply the switching between those two modes in an adaptive way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement and Scope

Complex Event Processing (CEP) has always been a topic of interest for both research and industry due to its complexity and applicability in many business areas. Well-known CEP applications include financial market data analysis, RFID-based asset tracking, operational business intelligence, and network intrusion detection and most of them require some form of complex pattern analysis on the data. For example an interesting pattern for financial market analysis is looking for a price increase in a stock of a company followed by a price decrease which is at least 30 percent of the increase. Thus in all of these applications, pattern matching over live, archived or their hybrid streams of events has become a key requirement and not a small number of CEP engines have been built to serve this purpose - such as Cayuga[9], SASE[12] or DejaVu [11]. They differ from each other in many aspects but all of them have to ensure good performance with limited computer resources. The goal of this thesis is to analyze and optimize the DejaVu CEP system as it is in terms of performance with bounded memory resources.

1.2 Related Work and Contribution

Most of the CEP systems handle the input stream in a similar way. Incoming data is preprocessed at first for example by "Event Receivers" [9] or in the "Cleaning and Association Layer"[12] and directly pushed to the query processor for pattern detection. In DejaVu however input streams are treated differently. Compared to existing CEP engines the focus of DejaVu is to find patterns over live and archived event streams as well. And for that purpose the MySQL open-source database system [4] is extended with the ability of pattern matching according to [10]. With the MySQL database system as its core system, the way how data gets into the query processor

is pull-based - a thorn in the stream community's side since in their view [7] "a query processor must instead react to arriving data". As far as it concerns simple data stream processing the statement above has surely its validity. However for CEP applications where a complex pattern has to be found - the query processor may not be able to react due to its workload - pulling data, whenever the processor needs it, makes definitely sense. On the other hand an "under-stimulated" query processor should be able to react to arriving data in order to face the real-time requirements of a CEP application. Thus one of the thesis contributions is to provide a "pushing" mechanism for such a case. Another interesting aspect of CEP systems is how they do handle the memory management. Since pushing data to the query processor may exceed the internal buffer of a stream processing system in general, every stream processing system including CEP systems has to manage the available memory in an optimal, efficient and careful way. In Cayuga for example a specific Garbage Collector is built for that purpose. A radical but not uncommon strategy is to discard some data when available memory is not enough - especially for many sensors e.g. RFID, the correct scheme is to discard the oldest samples, as they are least relevant to the current state of the sensor - as applied in Fjord-based[14] architecture for Input Handling in stream processing system e.g. [7]. However since DejaVu as an extended MySQL database system is able to operate by "pulling" data, switching between the modes "Push" and "Pull" becomes a key aspect in memory management. Doing this adaptively may lead to an optimal way to handle the memory resources.

1.3 Thesis Organization

The remainder of this thesis is organized as follows:

- Chapter 2 provides the necessary background information of DejaVu.
- Chapter 3 explains how incoming data is provided to the Query Processing Engine in DejaVu
- Chapter 4 describes a strategy to switch between two Input Handling modes adaptively
- Chapter 5 shows the difference of Pull/Push in terms of performance and memory usage and illustrates the potential of Adaptive Switch
- Chapter 6 reviews this thesis in summary and proposes some future works based on the conclusions from the previous chapters

Chapter 2

Background

2.1 The DejaVu project

DejaVu is an event processing system that integrates declarative pattern matching over live and archived streams of events. The system is built on top of the MySQL open-source database system [4] and uses a recent proposal for an SQL-based declarative pattern matching language standard [15] as its query language (Listing 2.1). A pattern is defined as regular expressions over sequences of rows. An example pattern for financial market analysis is specified in Listing 2.2: A falling price, followed by a rise in price that goes higher than the price when the fall began.

Listing 2.1: MATCH_RECOGNIZE clause for pattern recognition according to the ANSI standard proposed in 2007 [15]

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

Listing 2.2: an example pattern for financial market analysis

```
PATTERN (A B+ C* D)
  DEFINE /* A defaults to True, matches any row */
    B AS (B.Price <= PREV(B.Price))
    C AS (C.Price > PREV(C.Price) AND C.Price <= A.Price)
    D AS (D.Price > PREV(D.Price) AND D.Price > A.Price)
```

2.2 Architecture of DejaVu

The architecture of DejaVu (Figure 2.1) is an extension to the basic skeleton of MySQL in support of running different forms of pattern matching queries (one-time, continuous, and hybrid). Where one-time queries operate on static data, continuous queries as well as hybrid queries detect patterns over live event streams and in the later case over both live and archived data streams. By making use of one of the key architectural features of MySQL - the pluggable storage engine API (Section 2.4) - two new types of storage engines for the Live Stream Store and the Archived Stream Store (Section 2.3) are built in order to support data streams for continuous and hybrid queries. Pattern matching happens inside the Query Processing Engine (Section 2.5), the core of DejaVu, which is able to operate in two modes - Push and Pull (Chapter 3). By pulling the data from a particular store whenever the Query Processing Engine needs, it's able to find the specified pattern and report the matches immediately to the client. Whereas in Push Mode the query processor waits for the input stream and process it as soon as it arrives. As one can see in Figure 2.1 the query processor is designed to operate in Push **and** Pull only with the Live Stream Store.

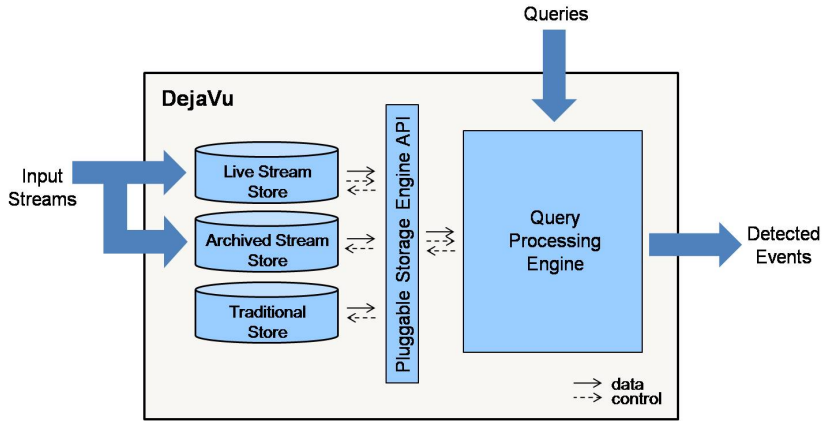

Figure 2.1: DejaVu System Architecture

2.3 Stream Storages

DejaVu supports two storages for data streams - the Live Stream Store (Section 2.3.1) and the Archived Stream Store (Section 2.3.2). Both stores serve their own purpose and can be accessed through their predefined low-level API's. Moreover in case of bursty or high-rated streams the Archived Stream Store can be used along with the Live Stream Store to prevent any

data loss. More details on the Live Stream Store and its access methods are provided in the corresponding section since one of the focuses of this thesis is to enhance the performance of pattern matching on live streams.

2.3.1 Live Stream Store

The Live Stream Store (Figure 2.2) is an in-memory store that accepts push-based inputs. More specifically it resides in the shared memory (paragraph **Shared Memory**) which enables concurrent access for the streaming application and the query processor. A streaming application generates an input stream by writing data periodically to the Live Stream Store. Depending on the input handling mode, data from the Live Stream Store is pushed to or pulled by the Query Processing Engine. In order to operate with the Live Stream Store, first a data schema has to be provided. For that purpose DejaVu is extended with the following syntax:

```
CREATE STREAM <name>(<column_name> type(length),...);
```

The data schema along with other meta data like type information etc. is stored as a shared memory resource for the access to the Live Stream Store. And all the basic access methods such as read() or write() are provided in a low-level API. It exists in form of a shared library which can be dynamically loaded by any processes such as the streaming application or the Query Processing Engine. A queue is used as an internal structure to ensure the order of incoming data. Read/write-operations on the Live Stream Store are implemented as dequeue() and enqueue() respectively. Since this store can be accessed by two concurrent processes (streaming application and query processor), operations on the queue are guarded by semaphores (paragraph **Semaphores**). A mentionable fact is that whenever a process attempts to read from an empty queue, its execution is blocked (by abusing the semaphore mechanism) until the queue becomes non-empty. This mechanism allows accessing threads responsible for Input Handling to be suspended.

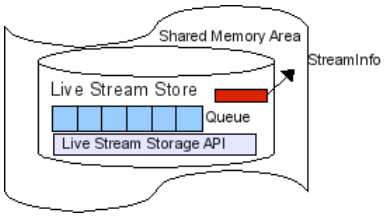

Figure 2.2: The Live Stream Store and its components

Shared Memory

Advantages of using shared memory for inter-process communication (IPC) are:

- Memory Management - when the processing rate is slower than the arrival rate of the input stream, the internal queue grows and it leads to a higher usage of the shared memory. In case of insufficient shared memory, the operating system takes care by file swapping.
- Performance - since both processes can access the shared memory area like regular working memory, this is a very fast way of communication.

On the other hand, it is less powerful, as for example the communicating processes must be running on the same machine (whereas other IPC methods can use a computer network). Thus streaming from other machines requires a socket which runs on the same machine as DejaVu. In summary a streaming application (itself as a local source) or its socket (for external sources) generates the input stream by writing data to the Live Stream Store.

Semaphores

Semaphores are used to control racing conditions between the concurrent processes. Special care has to be taken since they create too much latency to the streaming rate. On closer observation as shown in Figure 2.3 one can see that concurrent access is only safe if the queue contains more than one element. Therefore the queue is only locked if its cardinality is 1 or less.

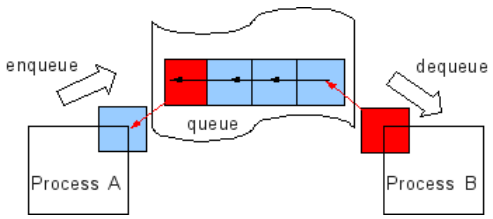

Figure 2.3: Concurrent access to the queue by Process A and B

2.3.2 Archived Stream Store

Live input events can also be fully or selectively materialized into an archive store. Archive store respects the pre-defined order of the events, and only allows updates in the form of appends. It is also designed to provide features such as data compression and efficient access methods for historical pattern matching queries. Furthermore, since the archive is a persistent store, it can support the live store in dealing with bursts and failures.

2.4 MySQL Storage Engine API

The Query Processing Engine uses the MySQL Storage API to access a particular storage. This API is implemented as an abstract class, which provides methods for basic operation such as opening, closing a table (in the view of the query processor in MySQL, a data source is always regarded as a table with a corresponding data schema) or reading a record. A storage engine implements the interface methods to translate the basic operations into low-level API calls for that particular storage. Therefore a storage engine loads the corresponding shared library containing this low-level API whenever it is instantiated by the query processor.

2.5 Query Processing Engine

The Query Processing Engine is the centerpiece of DejaVu and extends the MySQL query processor with pattern matching capability. As such it is able to handle the input in MySQL style - by pulling data with the Storage API and evaluating it one by one. But when it comes to pattern matching queries the execution model and logic differ from the original query execution. Based on Finite State Machines (FSM) (Section 2.5.1) the Query Processing Engine in combination with the DejaVu Router (Section 2.5.3) is able to detect patterns within a (non-)contiguous dataset which is buffered in the InputHolders (Section 2.5.2).

2.5.1 Finite State Machines

A FSM consists of three main components (Figure 2.4):

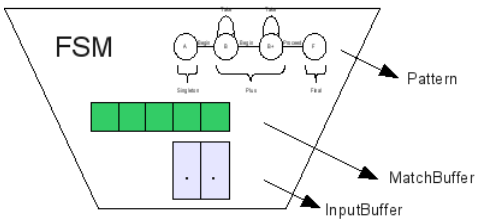

Figure 2.4: a Finite State Machine and its main components

- Pattern - the target pattern specified in the query is represented as a set of states and edges. For example given the following pattern (A, B+, C*), the corresponding FSM has a singleton state A, a plus state B+, a star state C* and a final state F as shown in Figure 2.5. Transitions from a state X to a state Y can only happen if the

corresponding edge predicate evaluates to **true** where Y can be X itself (reflexive transition e.g. C* in the figure). The target pattern is found whenever the FSM reaches the state F.

- MatchBuffer - holds the current partial matches. Whenever the whole match is found, the FSM reports the match immediately and empties the MatchBuffer.
- InputBuffer - holds two pointers which represent the begin and the current status of the semantic window.

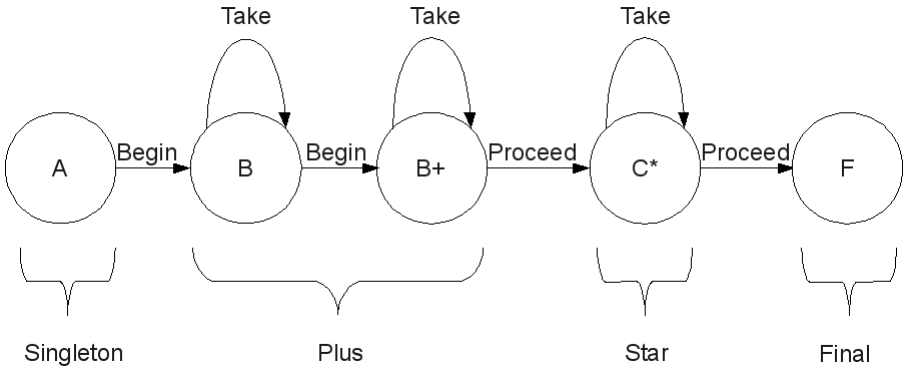

Figure 2.5: pattern (A, B+, C*) represented as states

Given a 'MATCH_RECOGNIZE' query, the query processor executes one or more instances of a FSM - referred to as a run - for (non-)contiguous pattern-search within the data set. (In case of a contiguous pattern matching only one FSM will be running.)

2.5.2 InputHolder

Data records are preprocessed to tuples and stored in memory buffers called InputHolders. Each InputHolder has a corresponding FSM which looks for the specified pattern within the buffered data (Figure 2.6). Since a record may belong to more than one pattern matches, each tuple has to be buffered until it is no more needed. This happens when the tuple is not in the range of the semantic window any longer (paragraph **Sliding the window**).

Sliding the window

There are two cases where a FSM resets itself and moves the semantic window according to a particular strategy:

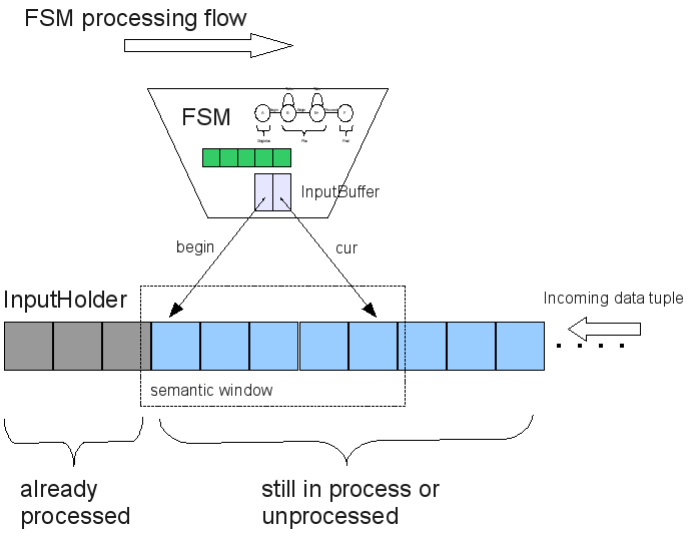

Figure 2.6: a FSM during its execution on an InputHolder

- pattern match - after a match has been found by the FSM inside the window, sliding happens according to the strategy specified in the query.
- pattern non-match - when the processed data does not fulfil the pattern condition, the sliding strategy *AFTER MATCH SKIP TO NEXT ROW* is applied.

Currently DejaVu supports two different sliding strategies:

- AFTER MATCH SKIP TO NEXT ROW - the FSM starts to search for a new pattern with the tuple after the first tuple of the previous data inside the InputBuffer.
- AFTER MATCH SKIP PAST LAST ROW - pattern search starts with the tuple after the last tuple of the previous data inside the InputBuffer.

Figure 2.7 illustrates a FSM before and after sliding according to the mentioned strategies. Sliding the window has a great impact on memory management since it increments the set of "already processed" tuples at least by 1. Those tuples are considered as "garbage" in the view of the Query Processing Engine.

2.5.3 DejaVu Router

In case of non-contiguous pattern matching, an InputHolder along with a FSM is created for each partition and incoming tuples are forwarded by

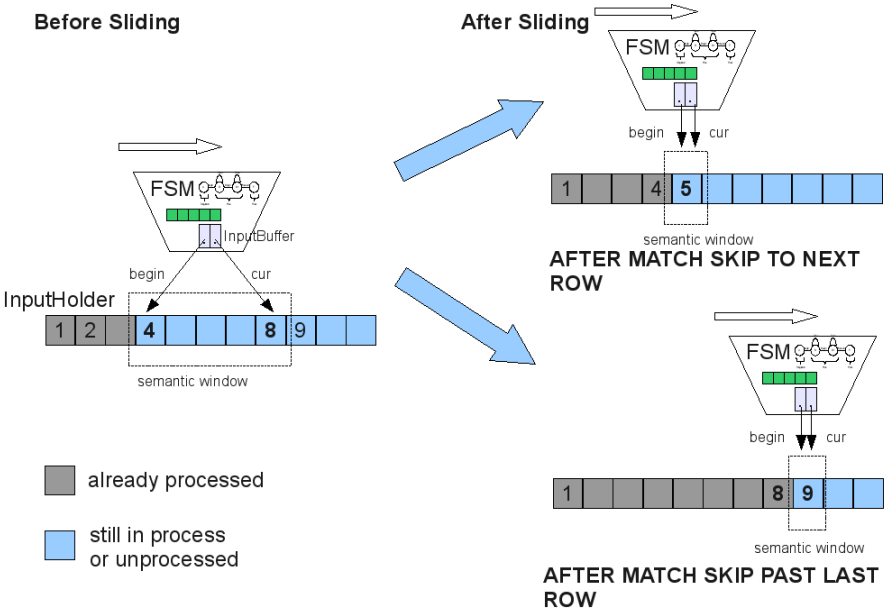

Figure 2.7: Effect on a FSM and its InputHolder after Sliding

the DejaVu Router [13] to those buffers wrt. their partition. Figure 2.8 illustrates the routing mechanism for incoming data tuples. Currently all runs are executing in a single thread and scheduled by a simple loop through the runs. As for the contiguous case all the data is considered as a single partition - therefore only one FSM and one InputHolder exist in run-time.

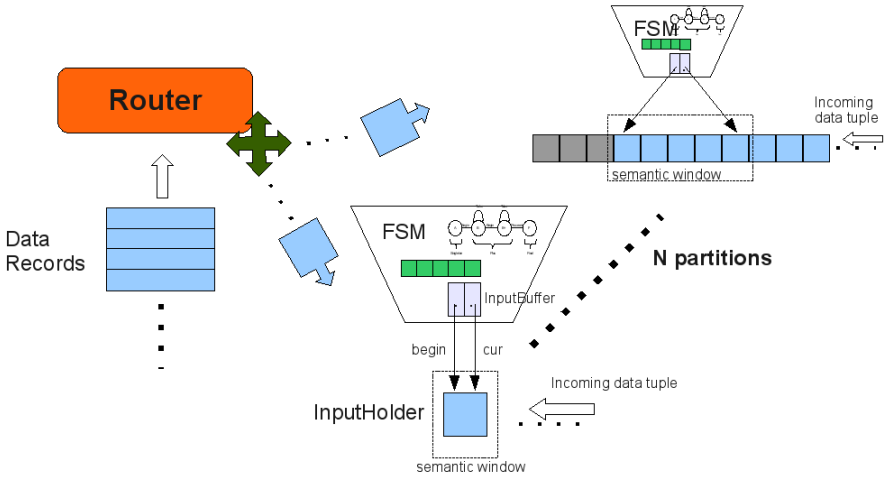

Figure 2.8: Input Handling in DejaVu

Chapter 3

Input Handling in DejaVu

In the previous chapter we have seen that DejaVu is able to handle push-based inputs (paragraph **Never-ending stream**) with the Live Stream Store in two modes: Pull and Push. Both methods describe different approaches how input streams reach the Query Processing Engine. And both of them have pros and cons in terms of memory usage and performance (Chapter 5). The following sections describe the two input handling modes in detail and compare them based on a simple cost model.

Never-ending stream

An input stream is considered as an endless data stream. Thus its ending has to be defined explicitly. For that purpose the low-level API of the Live Stream Store includes such a method for a streaming application which indicates the "end" of the stream table in the view of the Query Processing Engine. As a future work a signal mechanism has to be implemented to catch the event of abrupt unplugging of a streaming source.

3.1 Pull mode

In traditional database systems where data sets exist in form of static tables, the query processor operates by pulling record by record from a table source. DejaVu is able to handle streaming and static data in a similar way. Whenever data is needed for a pattern search, it pulls data from the (stream) table using the MySQL Storage API. A record is preprocessed to a data tuple and forwarded by the DejaVu Router to an InputHolder. If the corresponding InputHolder does not exist at that time, the router creates a new InputHolder and instantiates a new FSM running on that data holder. In case of non-contiguous pattern matching, there are as many FSM's/InputHolders as partitions. A record requested by a FSM may be routed to other partitions. In that case the demander run is skipped and

the next FSM in the waiting list is executed. Figure 3.1 illustrates a general view of DejaVu's architecture in terms of Input Handling in pull-mode while Figure 3.2 shows the Input Handling and Event Processing in a work-flow diagram.

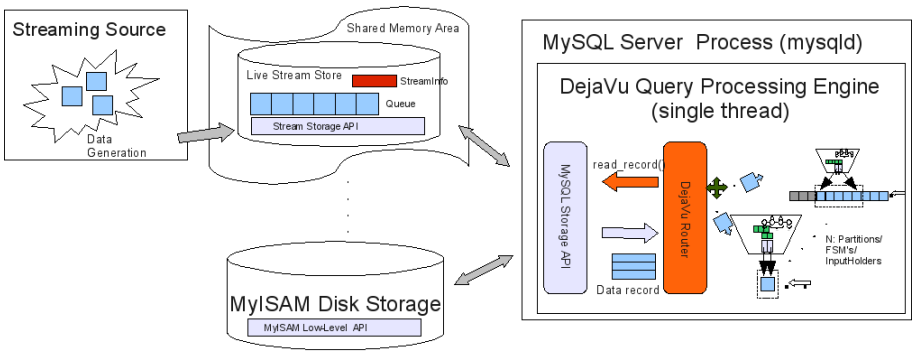

Figure 3.1: DejaVu in Pull-Mode: from Streaming to Processing

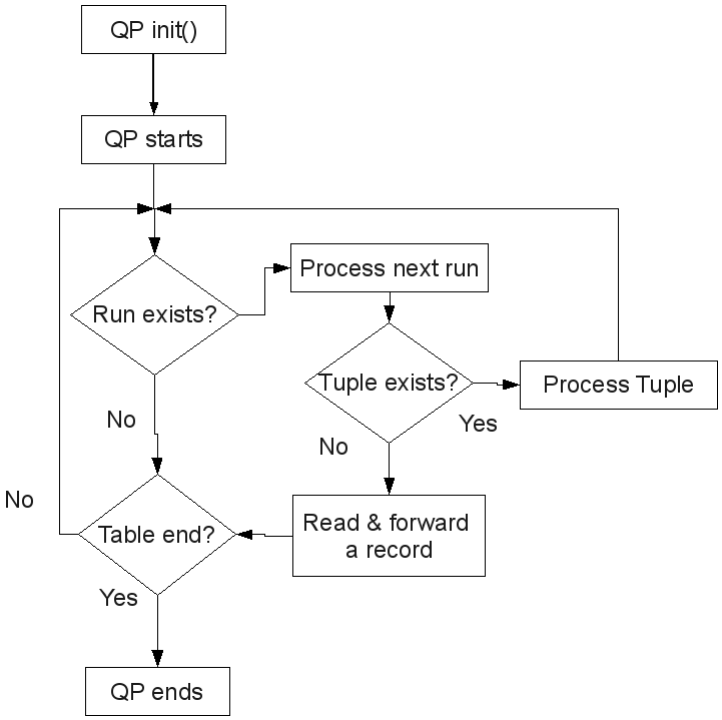

Figure 3.2: DejaVu in Pull-Mode: Work-flow of Event Processing

Remark - Operating with Live Stream Store

Since runs are currently operating in the same thread and scheduled by a simple loop, a traditional pull may block all the running FSM's when the Live Stream Storage contains no data. A small modification is needed for non-blocking pulling (Listing 3.1):

Listing 3.1: Non-blocking Pull

```
/*
 * sql_select.cc
 * MySQL Optimizer module
 */
/* procedure call for executing a run*/
static enum_nested_loop_state run_window(JOIN *join,JOIN_TAB
     *join_tab, int time_tick)
{
...
/* info represents the MySQL Storage API (handler object)*/
READ_RECORD *info= &join_tab->read_record;
...
/*
 * Modification:
 * Whenever the query processor wants a new record,
 * it checks the queue status of the Stream Storage
 */
if (info->file->get_record_no()>0){
  error= info->read_record(info);
...
}
```

The first read by the query processor is a blocking operation in case the streaming application is not fully initialized (Section 2.3.1). Streaming data accesses the Live Stream Store using its low-level API. Once the queue is non-empty, read-operations become non-blocking as shown above.

3.1.1 Cost Analysis

In order to find the "weakness" of running in Pull Mode, it's important to analyze the operating cost for the Query Processing Engine. As shown in Figure 3.3, the basic work-flow of the query processor can be simplified to the following four processing steps:

1. read_record() - read a record from a table source using the MySQL Storage API.

2. forward_record() - forward a record to the corresponding InputHolder

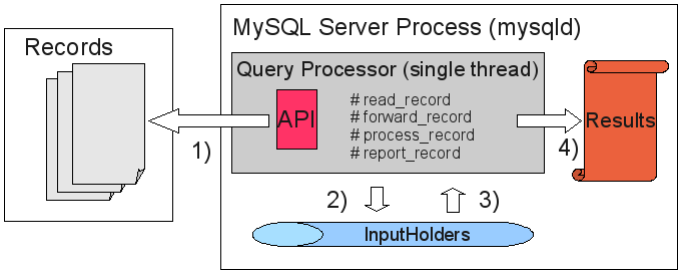

Figure 3.3: a simplified architecture and its work-flow in Pull Mode for Cost Analysis

3. process_record() - read a record from InputHolder and evaluates it by a FSM

4. report_record() - reports a record

For step 1 and 2, their maximal number of calls equals the cardinality of the input set $|D|$. Whereas in step 3 and 4 this is not the case. Each data tuple can be processed or reported several times - depending on the sliding strategy and the match mode - denoted by X_i and Y_i respectively. Based on the assumption (paragraph **Assumption**) that those four operations run in a constant time, the overall cost for the query processor C_{QPtot} would be:

$$C_{QPtot} = |D| \times (C_{read} + C_{forward}) + \sum_{i=1}^{|D|} X_i \times C_{process} + \sum_{i=1}^{|D|} Y_i \times C_{report} \quad (3.1)$$

Assumption

It must be pointed out that the cost model above does not match the reality exactly. In case of the reading cost C_{read} for example, an access time of a record in a table or stream - which exists in disk or memory - varies from record to record because of spatiality (see also **Operating with Stream Storage**). As long as the fragmentation remains low, the differences in access time are negligible. The cost to forward a data tuple varies as well. Because of the fact that when a corresponding InputHolder doesn't exist at that time, the query processor - more specifically the DejaVu Router - has to create a data holder and instantiate a FSM. As for the processing cost - especially the evaluation part since reading from an InputHolder in memory is regarded as constant (same as C_{read}) - its variation depends on the pattern specification or more detailed on the edge condition. If an edge condition consists of more than one pattern variable, then the correct tuple in the MatchBuffer has to be found for comparison. Thus the difference in

cost for forwarding and processing is in general not negligible. But for simplicity of the cost formula $C_{forward}$ as well as $C_{process}$ are noted as constant. Especially in terms of the processing cost $C_{process}$ it doesn't matter whether it is constant or not since the variation remains the same for both modes (Push/Pull) and thus no impact for their comparison (Section 3.3). The only operation, which can be regarded as really constant, is C_{report}.

Run scheduling

The fact that runs are scheduled in a simple loop generates a latency for each FSM execution. Moreover each run has its own total latency cost Z_i since some terminates earlier and some not. Thus the total latency for all runs due to the current scheduling for the query processor is $C_{run\_latency\_tot} = \sum_{i=1}^{K} Z_i$ where K is the total number of runs which equals the number of partitions.

Operating with Live Stream Store

Since the access to the Live Stream Store is based on semaphores, the corresponding storage engine may have to wait until the semaphore is increased (released). This implicates a waiting cost for each reading. Thus:

$$C_{read} = C_{sem\_latency} + C_{real\_read}$$

3.2 Push Mode

Operating in Pull Mode the query processor has to read and forward the data in order to get it processed. Comparing to other CEP systems like Cayuga or SASE where Input Handling and Event Processing happen in separated layers, this kind of additional workload affects the reactiveness of the Query Processing Engine negatively. The following sections propose an architectural approach and its implementation to avoid this problem. Furthermore a cost analysis for the Push Mode is presented based on the same cost model in Section 3.1.1.

3.2.1 Architecture

With the proposed architecture as shown in Figure 3.4 Input Handling - read records from a table through the MySQL Storage API and forward them using the DejaVu Router - is done in a separated thread and the Processing Engine can concentrate on its main task - to process records for pattern matching.

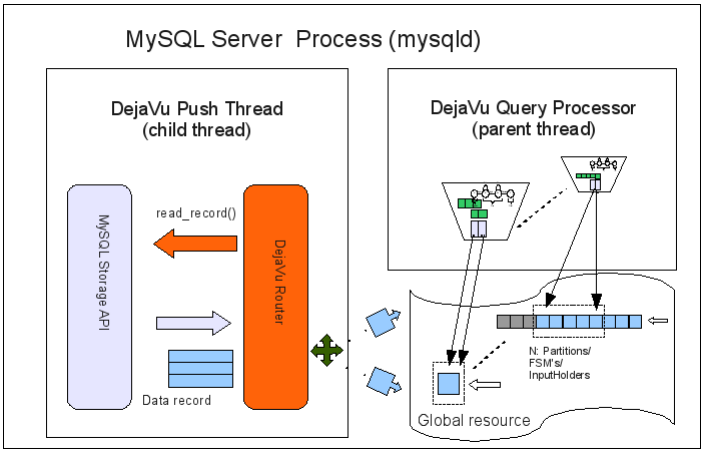

Figure 3.4: Push Mode Architecture

3.2.2 Implementation

The MySQL core system already uses a thread-based model to handle multiple client connections to the server. This model is implemented according to the POSIX standard [6] for Unix-systems. By following the same practice, the DejaVu Push Thread is created and cleaned (paragraph **Push Thread Creation and Termination**). Since the Query Processing Engine and the Push Thread may access global data at the same time and thus cause race conditions, special care has to be taken for running them in parallel (paragraph **Parallelization**).

Push Thread Creation and Termination

Whenever the extended DejaVu Query Processing Engine - referred to as the parent thread - runs in Push Mode , it initializes and executes the DejaVu Push Thread encapsulated in an MySQL THD object - referred to as the child thread - with some static information e.g. table, join, FSM, pointer to the thread procedure etc. needed for Input Handling. Since a newly created POSIX thread shares all of the calling process's global data with the other threads in this process (although it has its own set of attributes and private execution stack.), an access to shared resources e.g. InputHolders is provided by default. In case the streaming ends (paragraph **Never-ending stream**), the Push Thread terminates and cleans itself by joining its parent.

Remark - Pools of equals and their scheduling

In the section above and also in Figure 3.4 the terms parent/child thread are used for simplicity, e.g. to illustrate the order of the thread initializations. But with POSIX threads this hierarchical relationship doesn't exist. While a main thread may create a new thread, and that new thread may create an additional new thread, the POSIX threads standard views all your threads as a single pool of equals. So the concept of waiting for a child thread to exit doesn't make sense. The POSIX threads standard does not record any "family" information. This lack of genealogy has one major implication: if you want to wait for a thread to terminate, you need to specify which one you are waiting for by passing the proper tid to pthread_join()On the other hand a newly created thread inherits the calling thread's signal mask, possibly, and scheduling priority by default (Pending signals for a new thread are not inherited and will be empty). In this sense the usage of the terms parent/child is thoroughly suitable. The scheduling of the DejaVu Push Thread works like in other MySQL threads - the scheduling properties are left untouched, each thread runs with the same priority by the operating system.

Parallelization

Since two threads are working on the same resource - namely the set of InputHolders - one may notice the lack of locks. The reason behind this is to make full parallelization for the two concurrent threads possible. After careful consideration one can see that no race condition exists at all as shown in Figure 3.5. In view of the DejaVu Router (executed by Push Thread) only the last element of an element is of interest. Either the last tuple is at the end of the semantic window or not. In both cases no real concurrent access exists since the router changes the target of the *next* pointer where the other thread may process the *record* inside this tuple. Forwarding a tuple may lead to a run creation. Here again information about a run is encapsulated in an element with a next pointer. Thus the router modifies the next pointer of the last element in the list of runs, leaving the rest untouched while the processing run reads the run information.

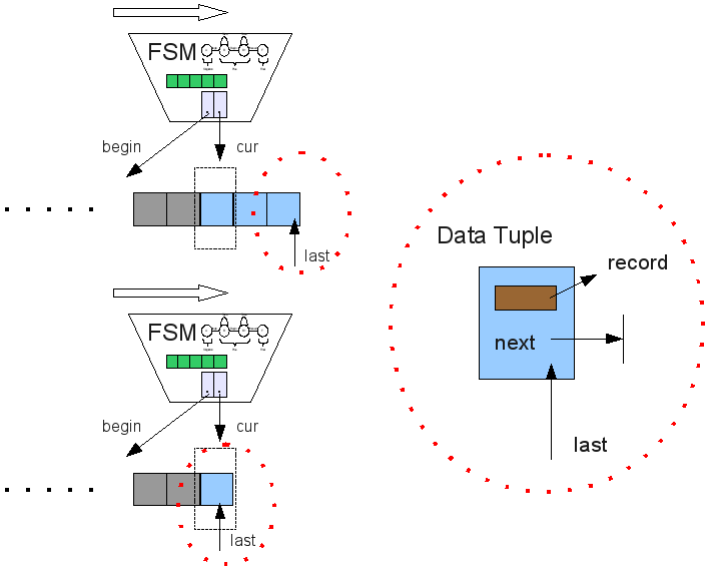

Figure 3.5: No race condition due to the Data Tuple structure

3.2.3 Basic Work-flow

The basic work-flow of Event Processing consists of two independent work-flows as separated by the fine dotted line in the middle of Figure 3.6. First of all the Query Processor Engine is initialized and started. Once it's done, it creates the DejaVu Push Thread for Input Handling. Its work-flow starts right after the initialization and keeps on reading and forwarding records until the streaming ends (paragraph **Never-ending stream**). In the meantime the Query Processor executes available runs one by one until there are no more tuples in the InputHolders AND the end of the stream table is indicated as true. At that point it waits for the Push Thread to join. A FSM execution basically consists of the following main steps:

- get a data tuple
- if there is a data tuple, process it
- return to query processor

Processing means - as mentioned earlier - evaluating the edge condition and if possible, report any pattern matches found.

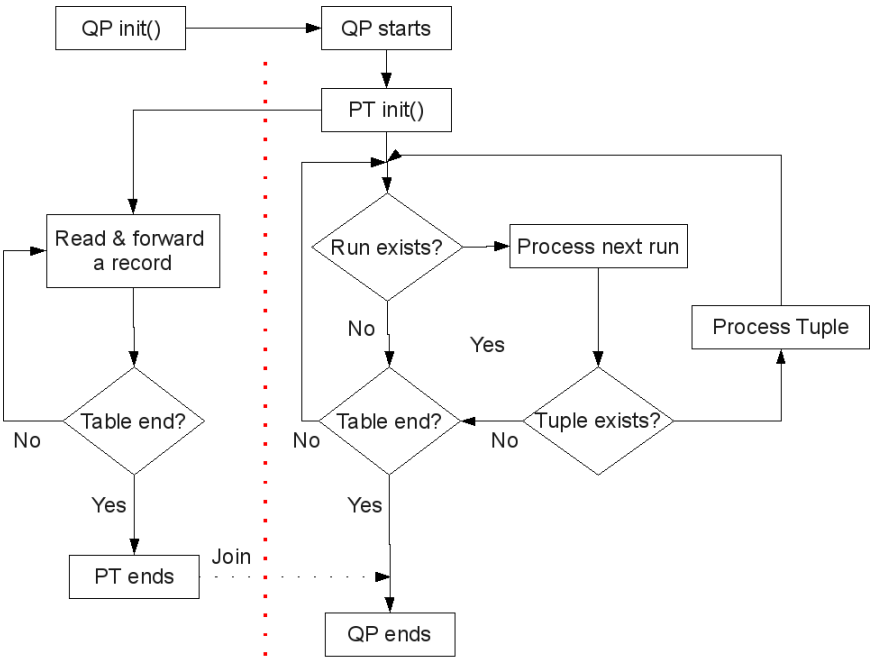

Figure 3.6: Basic work-flow of CEP in Dejavu operating in Push mode

3.2.4 Cost Analysis

With the proposed architecture - Figure 3.7 illustrates a simplification of Push Mode - the overall system cost can be separated into cost of query processing C_{QPtot} and cost of input handling C_{PTtot}. Based on the same assumptions and idea as for Pull Mode (section 3.1.1), the formulas would look like:

$$C_{PTtot} = |D| \times (C_{read} + C_{forward}) \tag{3.2}$$

$$C_{QPtot} = \sum_{i=1}^{|D|} X_i \times C_{process} + \sum_{i=1}^{|D|} Y_i \times C_{report} \tag{3.3}$$

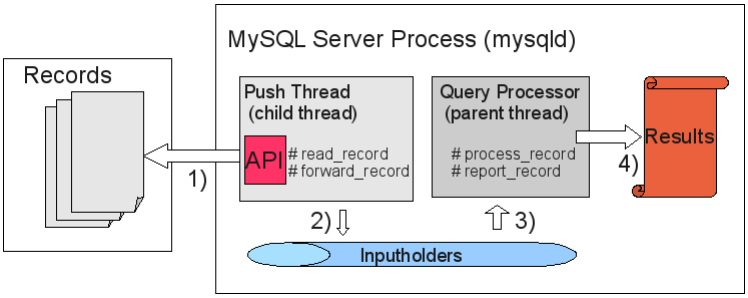

Figure 3.7: a simplified architecture and its work-flow in Push Mode for Cost Analysis

3.3 Conclusions

From the cost formulas for Pull (3.4) and Push (3.5) one can see that the overall cost for the query processor operating in Push mode is less - especially the overhead of pulling (3.6) comes to appearance when the size of the data set $|D|$ is large. As already mentioned in Section 3.1.1 the variation of $C_{process}$ as well as the variables X_i and Y_i depend on the semantics and thus remain the same for Push and Pull.

$$C_{QPtot\_Pull} = |D| \times (C_{read} + C_{forward}) + \sum_{i=1}^{|D|} X_i \times C_{process} + \sum_{i=1}^{|D|} Y_i \times C_{report} \tag{3.4}$$

$$C_{QPtot\_Push} = \sum_{i=1}^{|D|} X_i \times C_{process} + \sum_{i=1}^{|D|} Y_i \times C_{report} \tag{3.5}$$

$$C_{Overhead} = |D| \times (C_{read} + C_{forward}) \tag{3.6}$$

Operating in Pull mode, the Query Processing Engine requests data when necessary. Along with the cost model the following conclusions can be made:

+ A **low memory usage** in the query processor can be expected. Since the InputHolders are only filled if there are no more tuples for the current FSM execution.

- Since the query processor has to handle the inputs by itself, it makes the **processing rate slower**. Thus the Query Processing Engine is **less reactive** in terms of incoming data.

- When the input stream becomes high-rated or bursty, the usage of the **shared memory increases**.

As for Push Mode where data is pushed to the Query Processing Engine by a separated thread, the following pros and cons can be concluded:

+ The query processor doesn't have to deal with Input Handling. Thus the **processing rate** is **higher** which affects the **reactiveness** of the Query Processing Engine in a **positive** way.

+ The data load of the Live Stream Store - which resides in the **shared memory** - is **lower** since the DejaVu Push Thread will forward incoming data directly to the relevant InputHolders.

- The Query Processing Engine claims **more memory** than it's actually needed.

- An additional thread has to be created for the pushing architecture which requires **context switches** of threads during runtime.

Chapter 4

Adaptive Switch

In the previous chapter the two operating modes for Input Handling are presented in detail. Each mode has its pros and cons. As it concerns the Pull Mode the Query Processing Engine uses less memory. On the other hand pushing data to the query processor leads to a better performance in pattern detection. Therefore switching between those modes according to the context is the optimal way to handle the trade-off between performance and memory usage. The remainder of this chapter explains the motivation behind the switching condition, the condition itself and provides detailed information about the switching mechanism in terms of its architecture and implementation.

4.1 Motivation

The only reason to go from Push to Pull is because of the memory constraint every system has. Thus a straightforward strategy to switch to Pull is when a certain limit of available memory is exceeded and to toggle back when enough memory is freed (Memory blocks are only freed whenever a pattern is found or a non-match happens):

```
# define a threshold T
If mem_usage <= T Then doPush
Else doPull
```

The main disadvantage of following this simple plan is illustrated in Figure 4.1. Let's assume the specified query uses the sliding strategy AFTER MATCH SKIP PAST LAST ROW and there is one InputHolder with one FSM running on it to detect the patterns. The InputHolder has already allocated τ percent of its available memory. Thus by applying the switching condition above all the following data tuples 6,7,8 are pulled until the memory usage goes down below τ. Now the following happens: the latest tuple 8 pulled by the Query Processing Engine produces a match. Thus after sliding

all the tuples of the semantic window can be discarded and switching back to Push is applied. As one can see under such a circumstance staying in Push from the beginning would have been worthwhile since 6,7,8 could have been pushed.

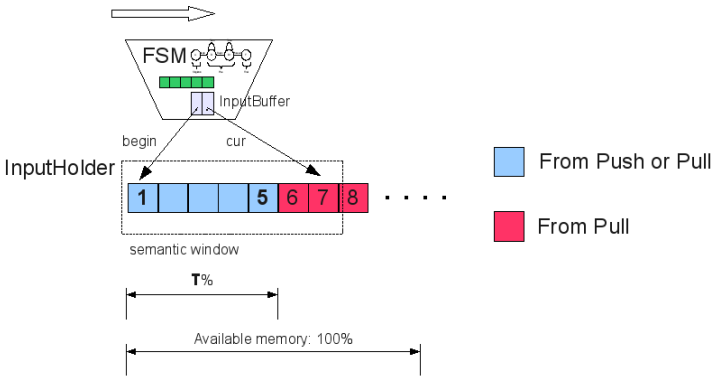

Figure 4.1: simple switch and its weakness

Remark - An Analogy to "Tetris" I

Tetris is a very popular game during the 90's and challenged by young and old. Especially in advanced levels where the blocks fall very quickly. And at the moment where only small spaces are left to maneuver the pieces, not a few had wished that they might come a little slower. DejaVu can grant you this wish comparing to other CEP engines. By pulling data by data whenever the Query Processor needs, the system has enough time to prepare and react to the worst case. Furthermore by switching the modes adaptively, the good pieces come in high speed where the bad ones are slowed down. The difficulty is to make a decision for each block whether it's good or bad...

4.2 Introduction

The idea behind the switching condition is try to predict whether the incoming data is "good" or "bad" based on some statistic information collected during run-time. "Good" tuples are those who fulfill the corresponding edge conditions and can potentially complete a pattern. The bad ones don't fit the right edge conditions and thus the causes for non-matches. The key to switch is the probability of a "good" sequence to appear. For example in combination with a tumbling window the good ones should be pushed as

soon as possible to the corresponding FSM's in the hope to get a match along with a memory reduction caused by a slide. Even the bad ones can be used as shown later. Such a strategy is kind of risky since it assumes that data underlies a certain distribution and thus the naming "Go4Risk". The next subsection shows the run-time status of a FSM and its InputHolder in order to understand when and what to predict.

4.2.1 Run-time status

A running FSM with n states has a current state S_{cur} and divides its InputHolder into three parts during run-time as illustrated in Figure 4.2. L elements are already processed and can be deleted right away. N elements reside inside the semantic window and match the states S_1 until S_{cur}. Note that N equals the number of edge transition made from S_1 until S_{cur}. Finally R unprocessed tuples are left and may fulfill the edge conditions for the states S_{cur+1} until S_n.

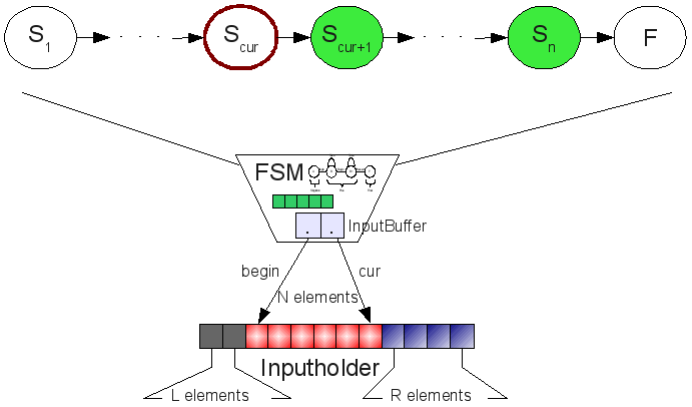

Figure 4.2: FSM runtime status

4.2.2 Impact of an unprocessed data sequence

Let's assume the following scenario (Figure 4.3) and the fact that this FSM completes a pattern when a k-length sequence of good tuples are coming. T is the number of blocks that has been allocated so far. No unprocessed tuples exist at the moment (R=0).

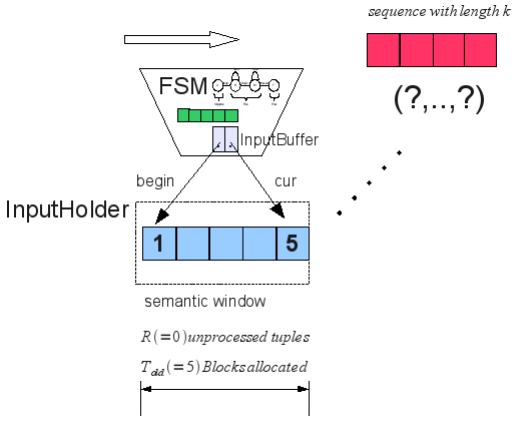

Figure 4.3: Impact of an unprocessed data sequence

Table 4.1 shows the impact of an input sequence with size k. Depending on the incoming data and the sliding strategy, moving the semantic window - as a consequence of a pattern match or non-match - affects the corresponding InputHolder differently. In this context T_{old} and T_{new} are the numbers of allocated blocks before and after the sliding respectively, R represents the unprocessed tuples after the slide and G resp. B denotes the good and bad tuples of a sequence. Here one can see that a good sequence of length k (G,G,G,..,G) (Figure 4.4) does not always affect the InputHolders in a positive way. In combination with a tumbling window it's obviously optimal since this sequence leads to a pattern along with a slide which makes $T_{new} = 0$. After the FSM-reset R becomes 0 as well. On the other hand sliding to NEXT ROW decreases the holder size by one but increases it as well by the length of the sequence. Thus $T_{new} = T_{old} + k - 1$. The FSM starts with the next tuple which means $R = T_{new} - 1 = T_{old} + k - 2$.

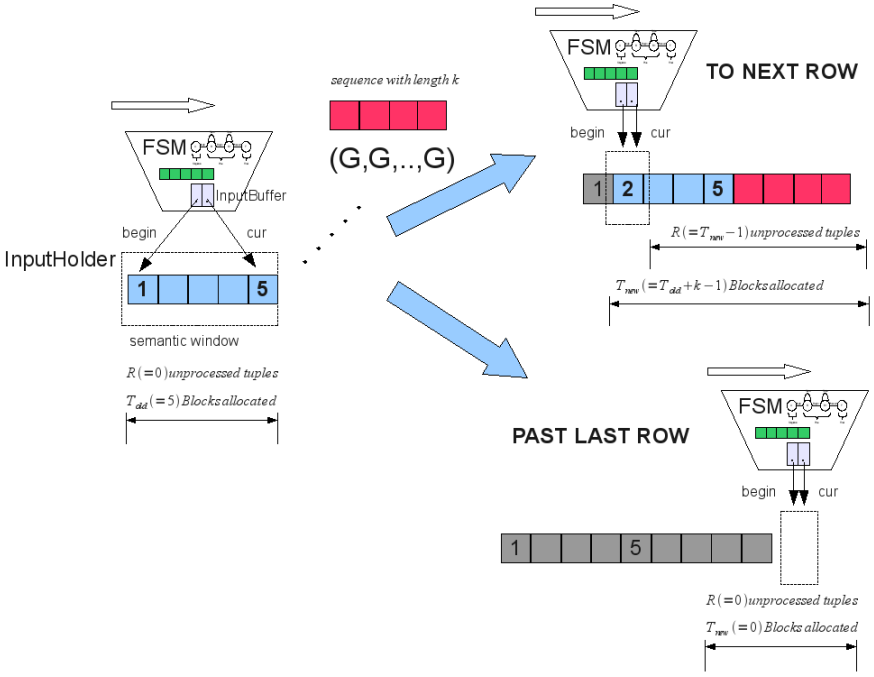

Figure 4.4: Impact of a good sequence of length k (G,G,G,..,G)

Having one bad tuple ((B); k=1) (Figure 4.5) is not really bad since a non-match leaves T unchanged (1 tuple comes to cause a non-match and 1 tuple leaves because of the non-match). But this is not true for a sequence containing a bad tuple ($k \neq 1$) (Figure 4.6) since the first bad one causes a non-match and due to the FSM reset the other tuples of this sequence can not be considered as good $G^?$ or bad $B^?$ anymore because of the changed context.

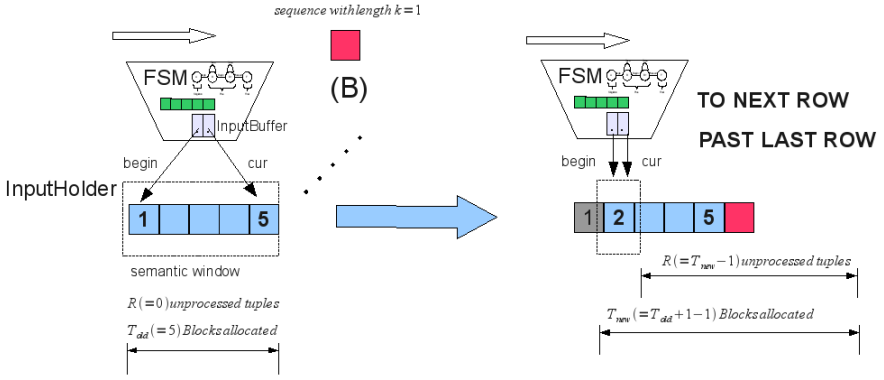

Figure 4.5: Impact of one bad tuple ((B); k=1)

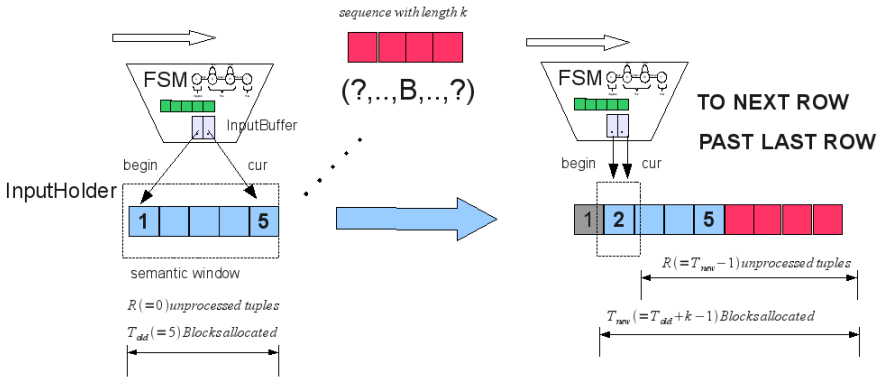

Figure 4.6: Impact of a sequence containing a bad tuple

In summary whenever a sequence contains a tuple which doesn't satisfy the corresponding edge condition, all the tuples of this sequence including itself lose their context property (good/bad) due to the FSM reset. On the other hand when no unprocessed data exist (R=0), processing a bad tuple won't cause any harm since the number of allocated blocks remains unchanged. The biggest impact however happens when a sequence of tuples completes a pattern in combination with a tumbling window. Thus data prediction may help to slow down (Pull mode) or speed up (Push Mode) the impact of an incoming data sequence.

tuple sequences with length k	NEXT ROW	PAST LAST ROW
(G,G,G,..,G)	$R = T_{new} - 1;$ $T_{new} = T_{old} + k - 1;$	$R = 0;$ $T_{new} = 0;$
(B) (k=1)	$R = T_{old} - 1;$ $T_{new} = T_{old};$	
$(B, B^?, B^?, ..)$ $(B, G^?, .., B^?, ..)$ $(G^?, .., B, G^?)$ ...	$R = T_{new} - 1;$ $T_{new} = T_{old} + k - 1;$	

Table 4.1: Impact of incoming tuple sequences in combination with the two sliding strategy: NEXT ROW and PAST LAST ROW

4.3 Data Prediction

It seems almost impossible to predict whether a data tuple is good or not since these properties are tightly coupled with the current FSM context as mentioned in the previous section. But first let's start with a simple example. Given a pattern (A) and the stream s=A:X:A:X:A:X the probability of the next tuple being a good one (=A) is $\frac{1}{2}$ since the input size is 6 and the match found is 3. One may conclude that the probability of a data tuple "belonging" to state S_i is

$$p_{S_i} = \frac{nr_{S_i}}{|s|},$$

where nr_{S_i} is the number of tuples "assigned" to S_i and $|s|$ denotes the size of the considered tuples so far. The flaw of this model is that a tuple may be processed several times and thus it may "belong" to several states as shown with the next example. Given the pattern specification ($A < 50$, $B < 40$) and the stream s={T1(=10),T2(=20),T3(=30)}, T2 may be considered as B and A as well. A more appropriate model is to make the switching decision based on the **selectivity** of the states (defined in Section 4.3.1). Applying this model changes the focus of prediction. Instead of predicting whether incoming tuples are good or bad this approach tries to foresee the selectiveness of the query processor and thus the way how it will react to unknown data based on historical information.

4.3.1 Statistic Model

The statistic model behind the prediction is based on the **selectivity** of the states inside a FSM. A state corresponds to a pattern variable specified in the query and three basic types of the states are differentiated:

- Singleton - holds one single match tuple
- Star - holds zero or more match tuples
- Final - represents the end of the pattern and does not contain any tuples

(Note that in the previous sections, the Plus state is mentioned which holds one or more tuples. It is a composite type (Singleton and Star) and thus not listed.) Two states S_i and S_j are attached to each other by an edge $e_{i,j}$ and in order to transit from state S_i to S_j, the condition on $e_{i,j}$ has to be evaluated to **true**. In case of Star state, reflexive transitions are allowed to hold more than one tuple. The Final state has no outgoing edge and represents the end of the pattern search. Figure 4.7 shows the three basic types of states and their edges. Note that the outgoing non-reflexive edge of a Star state evaluates always to true - denoted as $\epsilon - condition$ - and therefore the following **assumption** has to be made:

Assumption 1 (Mutual exclusiveness of edge predicates)**.** *Consecutive edge predicates are* ***mutual exclusive****. This assumption prevents a tuple from being "selected" by more than one state. For example given the pattern (A*,B) a tuple may be "consumed" by A* or by B since a transition to B by the non-reflexive edge of A* can always happen. In the implementation of FSM the edge predicates are evaluated in a particular* ***order*** *to avoid this scenario.*

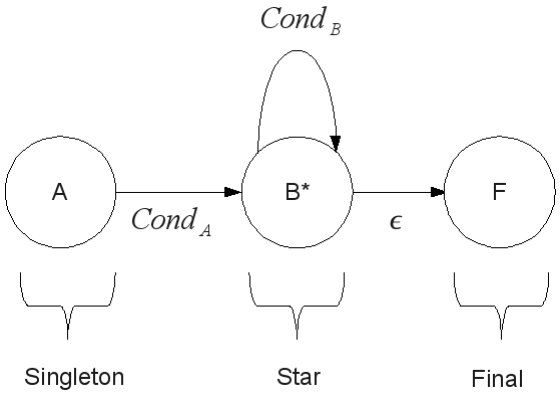

Figure 4.7: three basic types of states

A common definition of selectivity $S_{op}(R)$ of an operator *op* on a set R in the database community [8] is the percentage of number of tuples in R that satisfy the condition of *op*. For our purpose a similar formula is deduced. The selectivity sel_{Si} of a state S_i is defined as (Definition 4.1), where nr_{in} denotes the number of tuples coming in and nr_{out} the number of tuples leaving the state S_i. In order for a tuple to leave a State either the outgoing edge is evaluated successfully (Singleton) or the condition of the reflexive edge is not fulfilled (Star). Note that sel_{Si} can also be seen as the probability of leaving the state S_i and each of them are **conditionally independent** [3]. Thus multiplication of selectivities is allowed analog to the **Bayesian network** [1].

Definition 1 (Selectivity of a State S_i)**.**

$$sel_{Si} = \frac{nr_{out}}{nr_{in}} \tag{4.1}$$

Let's consider a simple example to see how prediction and selectivities of the states are connected. Given the pattern (A,B), the stream s=A1,A2,B1,B2 and tumbling window behavior, the first window w1 opens with A1. A1 fulfills $Cond_A$ and the FSM moves to B. Therefore $nr_{in} = nr_{out} = 1$ and $sel_A = \frac{1}{1}$. A2 can not match the pattern variable B and no further transition

happens. Thus the selectivity of state B after processing w1 is $sel_B = \frac{0}{1}$. The second window w2 starts with A2 and ends with B1. The match (A2,B1) is found and sel_A resp. sel_B changes to $\frac{2}{2}$ and $\frac{1}{2}$. At the end the last window w3 is processed and closes with a non-match of B2 to state A. sel_A resp. sel_B has the value $\frac{2}{3}$ and $\frac{1}{2}$. So what do sel_A and sel_B actually mean? sel_A says that from 3 opened windows (w1,w2,w3) only 2 of them (w1,w2) fulfill the A condition. For sel_B it means that whenever the previous state A has been passed, 1 (w2) out of 2 (w1,w2) windows contains another tuple matching B as well. Thus $sel_A * sel_B = \frac{1}{3}$ represents the so-called **Overall Match Statistic** (Definition 2) - from 3 opened window (w1,w2,w3) only 1 (w2) contains the pattern (A2,B1).

Definition 2 (Overall Match Statistic)**.** *The Overall Match Statistic is defined as the number matches found divided by the number of windows opened.*

As mentioned earlier selectivities of the states can also be seen as probabilities of the successful outgoing edge evaluations - illustrated in Figure 4.8. From state A's point of view, a new data tuple gets to state B with probability $\frac{2}{3}$ since the FSM opened a window 3 times but only for windows w1 and w2 a tuple could be found for transition A→B. The same is valid when the FSM's current state is B and it has to predict whether the next tuple is selected or not. Based on the collected data the probability to finish the pattern starting from B is $\frac{1}{2}$.

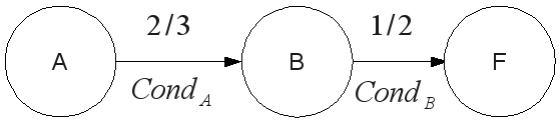

Figure 4.8: pattern (A,B) and the edge probabilities gained after processing A1:A2:B1:B2

In general for a FSM in the current state S_{cur} the multiplication of selectivities from S_{cur} to the last state (before Final state) S_n describes how likely it is to leave those states to complete the pattern (4.2). Thus when the remaining selectivity Sel_{remain} is high ($>= K$), the more likely it is to have the FSM finished the pattern search.

$$Sel_{remain} = \prod_{i=cur}^{n} sel_{Si} >= K \tag{4.2}$$

But how to determine K? Let's start with a simple example. Given a singleton pattern (A), the remaining selectivity is exactly the selectivity of the state A $Sel_{remain} = sel_A$. For a singleton there are only two options: leave or stay. Leaving means that a tuple has been found that satisfies the outgoing

edge condition. Following a uniform distribution, the expected outcome K would be 0.5 and thus if $Sel_{remain} >= 0.5$, the system has high acceptance. For a Star State however it's more difficult as shown in Figure 4.9.

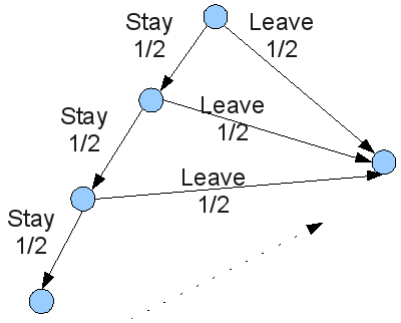

Figure 4.9: Selectivity of a Star state following an uniform probability distribution

Assume that the probabilities for Stay and Leave are uniformly distributed and both 0.5. But unlike a singleton, Leaving can also happen after a Stay. Consider the state A* and the data sequence X,Y. Processing X remains in A* with probability $\frac{1}{2}$ and Y will cause a non-match and leave A* with the same probability. Thus the probability to leave A* is $\frac{1}{2} * \frac{1}{2}$. The chance to leave (or selectivity) is $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...$ which converges to 1 as well as the expectation K. The formula (4.2) has to be rewritten as:

Definition 3 (Selectivity of the remaining states)**.**

$$Sel_{remain} = \prod_{i=cur}^{n} sel_{Si}, \quad sel_{Si} = 1 \ if \ S_i \ is \ Star. \tag{4.3}$$

Definition 4 (High/Low Sel_{remain})**.**

$$Sel_{remain} \ is \begin{cases} high & if \ Sel_{remain} >= K \\ low & otherwise \end{cases} \tag{4.4}$$

Definition 5 (Expected Outcome K following a uniform distribution)**.**

$$K = \prod_{i=cur}^{n} a_i \begin{cases} a_i = 0.5, & if \ S_i \ is \ Singleton, \\ a_i = 1, & otherwise. \end{cases} \tag{4.5}$$

On the first sight that seems not very helpful since it only says that the FSM will leave the state for sure after some time. But exactly this convergence - for any distribution as shown in the following proof - guarantees that the FSM must leave this state at a point of time.

Selectivity of a Star state converges to 1. As shown above the chance to leave a Star state (uniform distributed) is $sel_{Star} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...$ This is actually a geometric series $s = \sum_{i=1}^{\infty} \frac{1}{2^i}$ with its limes $\lim_{n \to \infty} s = 1$. For any distribution with the probabilities α to Stay and $\beta = 1 - \alpha$ to Leave, $sel_{Star} = \beta + \alpha * \beta + \alpha^2 * \beta + ... = \sum_{i=1}^{\infty} \beta * \alpha^i$. The limes is $\lim_{n \to \infty} sel_{Star} = \frac{\alpha}{(1-\beta)} = \frac{\alpha}{(1-(1-\alpha))} = 1$ □

As an example the following pattern (A*,B,B*,C) and the stream s={A1,A2,B1,C1} are considered to calculate the selectivities of each state step by step as shown in Table (4.2). Figure 4.10 shows the corresponding states and their edges.

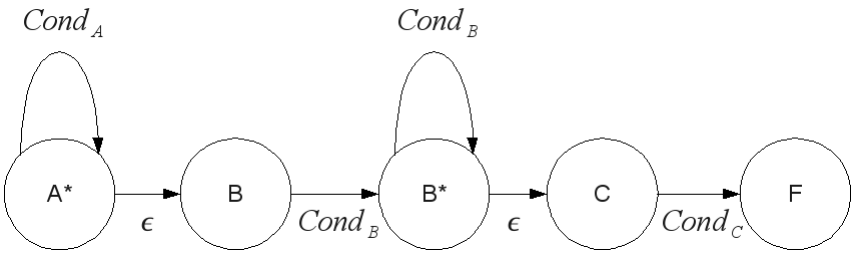

Figure 4.10: pattern (A*,B,B*,C) and their edges

In Step 1 the first tuple A1 fulfills $Cond_A$. Thus its selectivity is $\frac{0}{1}$ and the others remain 0. The same happens in Step 2 for A2 and $sel_{A^*} = \frac{0}{2}$. When the third tuple B3 comes, $cond_A$ is false and the outgoing edge from A* to B is taken, therefore $sel_{A^*} = \frac{1}{3}$. In state B, B3 makes the transition from B to B* possible and thus $sel_B = \frac{1}{1}$. For the last tuple C1 since it can't fulfill $Cond_B$ of the state B* but the other condition to C, the selectivity for B* is $sel_{B^*} = \frac{1}{1}$. C1 also causes the transition from C to F, therefore $sel_C = \frac{1}{1}$ and the pattern (A1:A2:B1:C1) is found by the FSM. In combination with a tumbling window, the pattern search is finished. In that case the remaining selectivity and K for the next window with current state A* is calculated as follows according to (4.3) and (4.5):

- $Sel_{remain} = \prod_{i=cur}^{n} sel_{Si} = 1 \times sel_B \times 1 \times sel_C = 1$
- $K = \prod_{i=cur}^{n} a_i = 1 \times 0.5 \times 1 \times 0.5 = 0.25$

Steps 5-10 show the further development when the sliding TO NEXT ROW is used.

Step	processed tuples	transitions made	sel_{A^*}	sel_B	sel_{B^*}	sel_C
1	A1	A*→A*	$\frac{0}{1}$	0	0	0
2	A1:A2	A*→A*	$\frac{0}{2}$	0	0	0
3	A1:A2:B1	A*→B→B*	$\frac{1}{3}$	$\frac{1}{1}$	0	0
4	A1:A2:B1:C1	B*→C→F	$\frac{1}{3}$	$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$
Sliding: TO NEXT ROW - discard A1, start with next tuple A2						
5	A2	A*→A*	$\frac{1}{4}$	$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$
6	A2:B1	A*→B→B*	$\frac{2}{5}$	$\frac{2}{2}$	$\frac{1}{1}$	$\frac{1}{1}$
7	A2:B1:C1	B*→C→F	$\frac{2}{5}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$
Sliding: TO NEXT ROW - discard A2, start with next tuple B1						
8	B1	A*→B→B*	$\frac{3}{6}$	$\frac{3}{3}$	$\frac{2}{2}$	$\frac{2}{2}$
9	B1:C1	B*→C→F	$\frac{3}{6}$	$\frac{3}{3}$	$\frac{3}{3}$	$\frac{3}{3}$
Sliding: TO NEXT ROW - discard B1, start with next tuple C1						
10	C1	A*→B	$\frac{4}{7}$	$\frac{3}{4}$	$\frac{3}{3}$	$\frac{3}{3}$
Sliding: TO NEXT ROW - discard C1						

Table 4.2: Evolution of the Statistic

Now having these tuples processed (TO NEXT ROW) the remaining selectivity and K for the next window with current state A* is calculated as follows according to (4.3) and (4.5):

- $Sel_{remain} = \prod_{i=cur}^{n} sel_{Si} = 1 \times sel_B \times 1 \times sel_C = \frac{3}{4}$
- $K = \prod_{i=cur}^{n} a_i = 1 \times 0.5 \times 1 \times 0.5 = 0.25$

One may ask for what purpose the selectivities of Star states are calculated since they are considered as 1 as stated above. The clou is that their recip-

rocals (4.6) are the average numbers of tuples "needed" per window avg_{S_i} to leave the star state.

Definition 6 (Average Number of Tuples inside a Star State)**.**

$$avg_{S_i} = sel_{S_i}^{-1} \textit{ if and only if } S_i \textit{ is a Star state.} \tag{4.6}$$

All in all based on this model first we can determine whether the remaining selectivity is high or low and second to predict the number of tuples needed to leave the Star State - both as part of the Go4Risk strategy introduced with the next section.

Excursus - Overall Match Statistic

In general the overall match statistic of singletons pattern is calculated as 4.7 where n denotes the number of states and p_{Si} the selectivity of a state S_i.

$$Match_{overall} = Sel_{overall} = \prod_{i=1}^{n} p_{Si} \tag{4.7}$$

Proof of 4.7. The proof is based on induction. The base case (n=1) holds always since the Match Statistic of a singleton state is exactly its selectivity. Let's say the $Match_{overall} = \frac{out}{in}$ where *out* are the numbers of tuples leaving S_n when *in* tuples are fed. Now the pattern is extended by one singleton state with a selectivity of $p_{S_{n+1}}$. Exactly *out* tuples are passed to S_{n+1} and only $p_{S_{n+1}} * out$ are not filtered out and can leave the final state S_{n+1}. Thus the new overall Match Statistic is $Match_{overall} = \frac{p_{S_{n+1}} * out}{in} = \frac{out}{in} * p_{S_{n+1}} = \prod_{i=1}^{n} p_{Si} * p_{S_{n+1}} = \prod_{i=1}^{n+1} p_{Si}$. □

The main difference between a Star and a Singleton state in terms of Match Statistic is that the selectivity of a Star state may consider zero, one or more tuples for an open window. Whereas for a singleton state there is exactly one tuple per window, if possible. Thus the formula above is only correct if the selectivities of Star states are ignored and the last pattern variable is a singleton.

4.4 Go4Risk

From the previous chapter it's shown that the Query Processing Engine has a better performance when operating in Push Mode due to the separated Input Handling. On the other hand the memory usage is very high from the query processor's view since incoming data is forwarded right away with the

speed of the input rate. But actually it's not necessary to push all of the data as soon as possible, especially when the processing rate is slower than the input rate. In this case the amount of unprocessed tuples increases and remains untouched until they get processed. At worst these tuples reserve such an amount of memory that other FSM executions are blocked until the corresponding run hopefully can find a match or non-match and releases the obsolete tuples. Thus in Push mode the number of tuples to forward should be bounded by R_{Bound}. If the number of unprocessed tuples R is higher than R_{Bound} ($R > R_{Bound}$) then the query processor should operate in Pull Mode - which actually means that Pushing is stopped until $R <= R_{Bound}$. Actually in order to avoid a permanent toggling between the two modes - since the difference between the conditions is only 1 - Push should be suspended until $R <= 1$. (Note that a run executing in Pull Mode is slower if and only if R equals 0 because in this case the FSM has to request data by itself and causes some latency to the processing time.)

4.4.1 Match Length Prediction

But how should R_{Bound} look like? Let's analyze the FSM and its InputHolder run-time status. Let M tuples reside in the semantic window and the number of unprocessed tuples R=0. If the selectivity of the remaining states is high - which means that it's likely to have a match with the next l_{seq} tuples - then a slide will happen. When a tumbling window is used, no data is left to be processed and the FSM has to pull the next tuple. To prevent this from happening R_{Bound} should be $l_{seq}+1$. Additional to that if the overall Match Statistic is high ($>= 0.5$), a **constant U** can be specified along with $l_{pattern}$ to reduce the switching cost (see **Excursus - User specific** R_{Bound}). On the other hand when the probability of a match is low - which means it's likely to have a non-match - then the next l_{seq} tuples cause a slide by 1 (and the number of unprocessed tuples is $R = M + l_{seq} - 1$). R_{Bound} is set to l_{seq} although the memory increases by $l_{seq}-1$. These tuples have to be tested anyway to cause the non-match.
In case of a sliding window (NEXT ROW) - where a match and a non-match have the same effect - R_{Bound} is set to $l_{seq} + U * l_{pattern}$ and l_{seq} respectively similar to a tumbling window. Table 4.3 shows the boundary of R according to the situation in summary.
Now what are l_{seq}, **U** and $l_{pattern}$? Since the data sequence with length l_{seq} will "test" the remaining states, it should be as described in Formula (4.8) with avg_{S_i} as the average number of tuples needed by the Star state to leave. Since the Star state may cause many tuple allocations, the first one should be passed first.

Definition 7 (Match Length Prediction - Calculation of l_{seq})**.**

$$l_{seq} = \sum_{i=cur}^{n} a_i \begin{cases} a_i = 1 & \textit{if } S_i \textit{ is Singleton,} \\ a_i = \lceil avg_{S_i} \rceil & \textit{else if } S_i \textit{ is the first Star state} \\ a_i = 0 & \textit{otherwise} \end{cases} \tag{4.8}$$

As for the constant **U** and $l_{pattern}$ - where they are used to predict the length of future matches (see **Excursus - User specific** R_{Bound}) - they are defined as follows:

Definition 8 (Future Match Length Prediction - **Constant U** and Calculation of $l_{pattern}$)**.**

$$U \in \aleph_0 \quad [\textit{Default: } U = 0;] \tag{4.9}$$

$$l_{pattern} = \sum_{i=1}^{n} a_i \begin{cases} a_i = 1 & \textit{if } S_i \textit{ is Singleton,} \\ a_i = \lceil avg_{S_i} \rceil & \textit{else if } S_i \textit{ is the first Star state} \\ a_i = 0 & \textit{otherwise} \end{cases} \tag{4.10}$$

	TO NEXT ROW	PAST LAST ROW
Sel_{remain} high & $Match_{overall} > 0.5$	$l_{seq} + U * l_{pattern}$	$l_{seq} + 1 + U * l_{pattern}$
otherwise	l_{seq}	$l_{seq} + 1$

Table 4.3: Values of R_{Bound} according to the situation

Based on the Statistic Model and the conclusion made for prediction, the switching strategy "Go4Risk" works as follows: Whenever a tuple is processed by a FSM, it updates the Statistic Model, calculates the selectivities of the remaining states and set R_{Bound} according to Table 4.3. If the number of unprocessed tuples R is smaller than R_{Bound}, more tuples should be forwarded (Push Mode). Otherwise the Adaptive Switch should stop pushing (Pull Mode) more data to the InputHolders. Let's make an example based on the previous one (Table 4.2) starting with the current State A*. The remaining selectivity is high since $Sel_{remain} = \frac{3}{4}$ and the expectation K=0.25. Thus R_{Bound} is calculated as follows:

- $R_{Bound} = l_{seq} = \sum_{i=1}^{n} a_i + b_i = (\lceil \frac{3}{4}^{-1} \rceil + 0) + (1+0) + (0+0) + (1+0) = 3$

Now if R is larger than 3 pushing is stopped since there is no need to push more data and will be resumed if R decreases to < 3.
With this setup Adaptive Switch should reach a performance close to "always pushing" (since R is at least 1) but with less memory usage in average

in the query processor's perspective. Note that in combination with a tumbling window, staying in Push is more likely compared to the sliding TO NEXT ROW since R decreases slow in the later case and may not fulfill the criteria $R <= R_{Bound}$.

Excursus - User specific R_{Bound}

Actually if R_{Bound} is set to 1, it leads to the best memory household possible since this guarantees that R is never 0 and thus the query processor always reads from InputHolder instead from the Storage. But the problem is the switching cost since a permanent Push/$Stop_{Push}$ may happen. Reducing this switching cost is the only reason for prediction. If the predicted selectivity is **high**, we make a gamble if the proposed data sequence l_{seq} can end the pattern search or not. But we can still go further and say: if it's likely to find a match, then why not pushing more data to reduce the number of switching. For that purpose the length of the future match $l_{pattern}$ is predicted similar to l_{seq} but starting with the first pattern variable (Formula 4.10). The **constant U** represents the number of future matches and is specified by the user when he wants more performance in cost of memory. The tradeoff here is:

the higher U (and therefore R_{Bound}) the less the switching cost but also the risky the memory management

4.4.2 Coordination with non-contiguous data set

In case of non-contiguous data where several FSM's and InputHolders exist in run-time, special care has to be taken since the FSM executions are scheduled in a single thread inside a loop. Thus a decision for Push/Pull made during a run affects the other FSM's as well. The idea to overcome this problem is to analyze only those InputHolders to which data is forwarded mostly. By following this approach the impact of switching should also appear on those InputHolders instead of the other ones. For this purpose the statistic model has to be extended with the global data distribution to each holder and that partition is chosen with the best distribution. Once the target InputHolder is set, the Go4Risk strategy is applied to it.

4.5 Architecture

The Adaptive Switch acts as an intermediate between the Push Thread and the Query Processing Engine as shown in Figure 4.11. When the query

processor operates in Pull Mode, it calls the Adaptive Switch (Go4Risk) before executing a FSM to decide whether it should keep operating in the same mode or switch to Push. In the later case the Push Thread is resumed for Input Handling. To switch back from Push to Pull the Push Thread checks after forwarding a data tuple whether it is worthwhile to stay in Push by asking the Adaptive Switch. If Push does not pay off, the Push Thread suspends itself and signalizes the Query Processing Engine to operate in Pull Mode again.

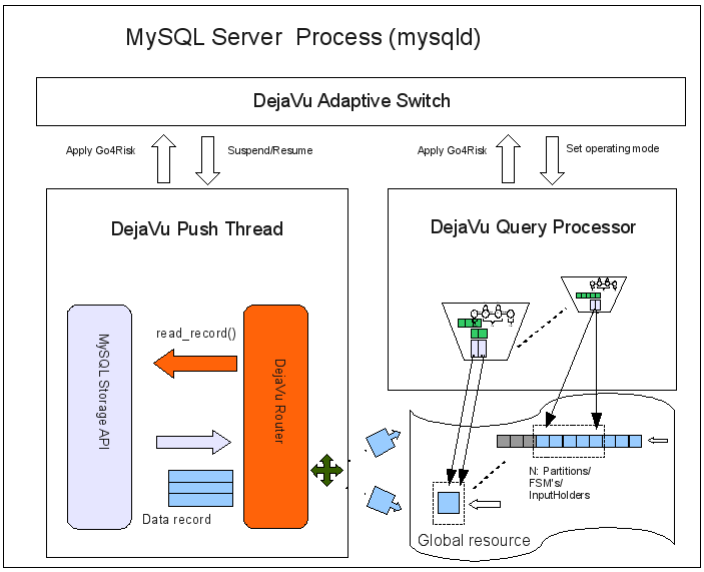

Figure 4.11: Adaptive Switch Architecture

4.6 Implementation

The implementation of the Adaptive Switch consists of a mechanism to suspend/resume the Push Thread and to set the operating mode of the Query Processing Engine in a consistent way. In POSIX this mechanism is realized using a condition variable (Listing 4.1). Whenever the Push Thread is active, it asks the Adaptive Switch whether the pushing condition $R <= R_{Bound}$ still holds or not. In the first case nothing happens and the next data tuple is pushed but for the later case ($R > R_{Bound}$) the operating mode changes to Pull and the thread is suspended by pthread_cond_wait(). At the same time the particular FSM (Section 4.4.2) checks the condition $R <= 1$ after each tuple evaluation. If it is true, the Push Thread is revoked by pthread_cond_signal(). Otherwise nothing happens.

Listing 4.1: Suspend/Resume mechanism in Adaptive Switch

```
/*
 * Suspend/Resume mechanism of the Adaptive Switch
 * adaptive_switch_proc() in sql_select.cc
 */

if ((router->current_mode()==PushMode)&&sw_cond_pull){
    // Switch From Push Mode to Pull Mode
    router->set_mode(PullMode);

    pthread_cond_wait(&ad_switch_cv,&ad_switch_mutex);
  }
  else{
    if ((router->current_mode()==PullMode)&&sw_cond_push){
      //Switch From Pull Mode to Push Mode
      router->set_mode(PushMode);
      pthread_cond_signal( &ad_switch_cv );
    }
  }
```

In addition to the suspend/resume mechanism a statistic module is implemented to collect all the relevant information in run-time. More specifically each FSM holds an instance of this statistic module which is updated after a tuple evaluation. Besides some update functions the module also provides basic methods to calculate R_{Bound} and the overall match statistic. All the public methods are guarded by locks which allow concurrent accesses by the Push Thread (get R_{Bound}) or query processor (update information). Listing 4.2 shows the calculation of R_{Bound} according to the selectivity.

Listing 4.2: Calculation of R_{Bound}

```
/*
 * getRbound()
 * public method of class AdSwitch_statistic in
    dejavu_statistic.h
 */
int getRbound(int cur){
    int i=0;
    pthread_mutex_lock(&_mutex);
    if (FullMatch > 0){
      // calculate llseq
      i=calcllseq(cur);
      // calculate Sel_remain and K to check the selectivity
      if (calcSelRemain(cur)>=calcK(cur)){
        i+=llm;
        // in case U is set >1 and high matching rate
        if ((OpenWindow > 0)&&((FullMatch/OpenWindow) >= 0.5))
          i+=U*calcllseq(0);
      }
    }
    else{
      // if no match statistics are available
      i=nr_states;
```

```
22      }
23      pthread_mutex_unlock(&_mutex);
24      return i;
25    }
```

Chapter 5

Measurements

From previous chapters we know that DejaVu is able to handle inputs in two different ways: Pull and Push. Both have their pros and cons in terms of memory usage and performance. While pulling data allows an optimal memory management on the query processor's side, the push mode offers a better performance for pattern matching. With the introduction of the Adaptive Switch their strengths are combined by predicting the query processor's speed and only pushing a certain amount of tuples at a given time. Therefore the query processor should demonstrate a performance close to "Always Push" with a careful memory usage when switching is applied adaptively according to the context. In this chapter we present some experimental results to first show the difference between Push and Pull and second to signify the potential of Adaptive Switch.

5.1 Experimental Setup

All the following experiments were conducted on a laptop with a Intel Pentium Core2 Duo 1.8 Ghz and 2.0 GB memory on Ubuntu Intrepid 9.04 (Kernel 2.6.28-16-generic). The installed version of DejaVu is based on mysql-6.0.3-alpha. Measurements are divided into two sections: "Pull vs. Push" (Section 5.2) and "Adaptive Switch vs. Pull/Push" (Section 5.3). In both sections the memory usage and performance are evaluated with increasing data rates. More specifically they are measured with a particular data rate within the interval [50;5000] (tuples per second) and plotted with the Bezier curve [2] which is widely used to show an approximation of the data trend. In terms of memory usage we calculated the **average number of tuples** allocated by the query processor during run-time. That means whenever a new tuple is routed to the InputHolders, a memory snapshot of all current holders is taken and summed to the previous snapshots. At the end the average is computed by dividing the sum through the number of data tuples. For performance evaluation in general we measured on the one hand

the **throughput** of the system by dividing the total processing time of the query through the number of input tuples. In some experiments we also quantify the **latency in average** - the average time for a tuple residing in the storage until the first time being selected by the FSM - in order to support the conclusions made in the corresponding sections.

5.1.1 Data Set Specification

A real financial dataset from NYSE TAQ (Trade and Quote) database for January 2006 [5] is used for the experiments. Therefore all the entries can be regarded as randomly distributed. We set the size of the data to 5000 records.

5.1.2 Query Specification

The following code snippets describe our test queries Q1 (Listing 5.1), Q2 (Listing 5.2), Q3 (Listing 5.2) and Q4 (Listing 5.2). They are of financial nature and try to detect a tic-shape pattern within a data set. Q1 and Q3 consider the data as a single partition (contiguous pattern matching) whereas Q2 and Q4 looks for tic-patterns for each company (non-contiguous pattern matching) - partitioned by their names. As one may have noticed the only difference between Q1 and Q3 resp. Q2 and Q4 lies on the sliding strategy - using either a sliding (TO NEXT ROW) or a tumbling window (PAST LAST ROW). Both of them have to be considered in the experiments since they have great impact on the memory usage and performance. The choice fell upon a varying pattern instead of a fixed pattern to make the experiments more generally.

Listing 5.1: Test query Q1

```
SELECT * FROM trade MATCH_RECOGNIZE(
  MEASURES MATCH_NUMBER AS matchno
  AFTER MATCH SKIP PAST LAST ROW
  ALL MATCHES
  PATTERN(A B+ C* D+)
  DEFINE /* A defaults to True, matches any row */
    B AS (B.Price < A.Price AND B.Price <= PREV(B.Price))
    C AS (C.Price > B.Price AND C.Price >= PREV(C.Price) AND C.
       Price <= A.Price)
    D AS (D.Price > PREV(D.Price) AND D.Price > A.Price)
);
```

Listing 5.2: Test query Q2

```
SELECT * FROM trade MATCH_RECOGNIZE(
  PARTITION BY symbol
  MEASURES MATCH_NUMBER AS matchno
  AFTER MATCH SKIP PAST LAST ROW
  ALL MATCHES
```

```
  PATTERN(A B+ C* D+)
  DEFINE /* A defaults to True, matches any row */
    B AS (B.Price < A.Price AND B.Price <= PREV(B.Price))
    C AS (C.Price > B.Price AND C.Price >= PREV(C.Price) AND C.
        Price <= A.Price)
    D AS (D.Price > PREV(D.Price) AND D.Price > A.Price)
);
```

Listing 5.3: Test query Q3

```
SELECT * FROM trade MATCH_RECOGNIZE(
  MEASURES MATCH_NUMBER AS matchno
  AFTER MATCH SKIP TO NEXT ROW
  ALL MATCHES
  PATTERN(A B+ C* D+)
  DEFINE /* A defaults to True, matches any row */
    B AS (B.Price < A.Price AND B.Price <= PREV(B.Price))
    C AS (C.Price > B.Price AND C.Price >= PREV(C.Price) AND C.
        Price <= A.Price)
    D AS (D.Price > PREV(D.Price) AND D.Price > A.Price)
);
```

Listing 5.4: Test query Q4

```
SELECT * FROM trade MATCH_RECOGNIZE(
  PARTITION BY symbol
  MEASURES MATCH_NUMBER AS matchno
  AFTER MATCH SKIP TO NEXT ROW
  ALL MATCHES
  PATTERN(A B+ C* D+)
  DEFINE /* A defaults to True, matches any row */
    B AS (B.Price < A.Price AND B.Price <= PREV(B.Price))
    C AS (C.Price > B.Price AND C.Price >= PREV(C.Price) AND C.
        Price <= A.Price)
    D AS (D.Price > PREV(D.Price) AND D.Price > A.Price)
);
```

5.2 Pull vs. Push

Figure 5.1 and 5.2 show the memory usage and performance of Q1 (contiguous pattern matching, tumbling window) denoted by their y-axis where the x-axis represents the interval of the data rate from 0 to 5000 tuples per second.
What we expect is that the query processor operating in Pull Mode uses a constant amount of memory for all the input rates (Figure 5.1). The reason behind this is because of the nature of pulling. It only allocates new tuple if there is no unprocessed data left in the InputHolders. On the other hand in Push Mode the usage of the memory increases along with the data rate since more data is pushed at a given time.

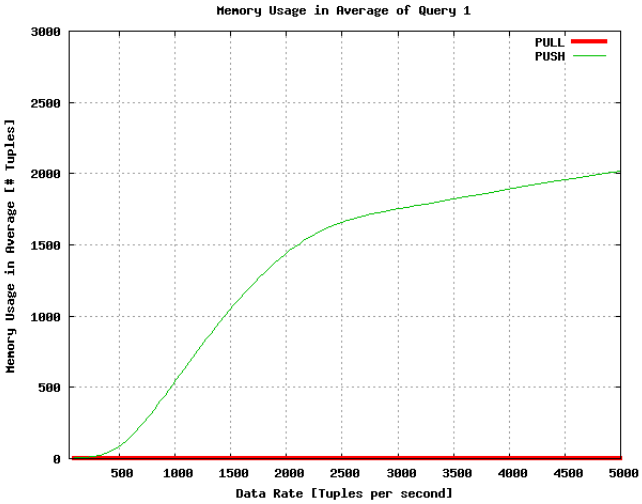

Figure 5.1: Pull vs. Push: Memory Usage of Q1 (contiguous, tumbling window), Result size: 7457

As it concerns the performance (Figure 5.2) one can conclude that for input rates below ∼2000 tuples/sec the query processor is under-stimulated. It could have processed more tuples (and thus could have a higher throughput) at a given time but the data was not coming fast enough. We also can see that the throughput remains almost constant (∼1100 tuples/sec) when the data rate is higher than 2000 tuples/sec. That's the maximal throughput a query processor can achieve in Pull mode in this example. On the other hand operating in Push mode, where the query processor does not have to handle the inputs, the throughput grows with increasing data rate until ∼1500 tuples/sec. Thus by having the data pushed into the InputHolders, the query processor is able to show off its true capability - an increase of the throughput by almost 30 percent. This performance gain is also reflected in latency as shown in Figure 5.3. Note that the latency for Pull is also ∼30 percent higher than Push and the curves increase with the data rate because the query processor cannot cope with the higher speed.

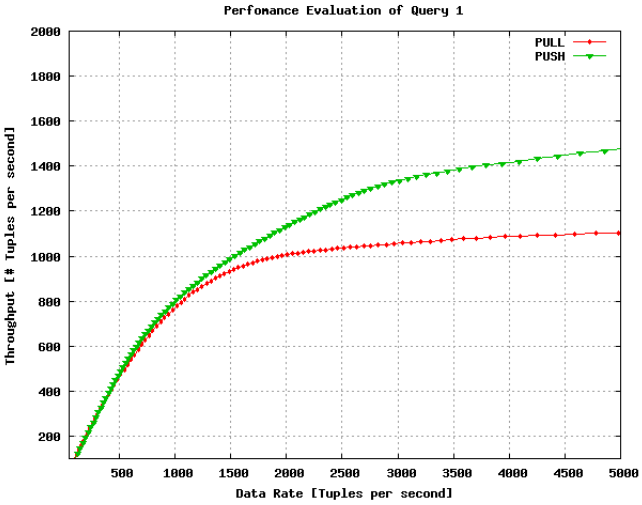

Figure 5.2: Pull vs. Push: Performance Evaluation of Q1 (contiguous, tumbling window), Result size: 7457

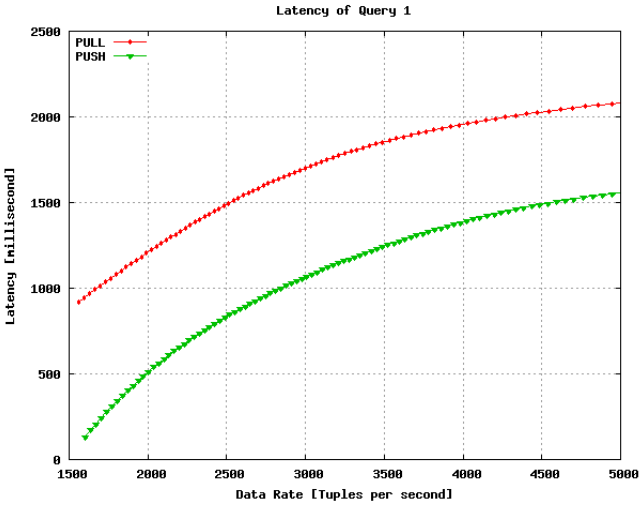

Figure 5.3: Pull vs. Push: Latency of Q1 (contiguous, tumbling window), Result size: 7457

As for the non-contiguous case (Q2 shown in Figure 5.4 and 5.5) the curves may seem a little bit strange at first sight since they don't cope with the statements above. But on closer observation the graphs surely make sense since routing a tuple in Pull mode to other holders has a **similar effect as pushing**. Whenever a FSM requests data and the next tuple of the queue

does not "belong" to the caller's partition, the data will be forwarded to the corresponding run instead. Next time when this run's turn comes, its FSM does not need to handle any inputs since there is already an unprocessed tuple in the InputHolder forwarded in a previous step. As one can see in the figures the mentioned side effect leaves a great impact on the memory usage and performance. Although the query processor is operating in Pull mode, the curves in memory usage and performance are very push-like - the increase of memory allocation and throughput are only around 20 and 12 percent respectively.

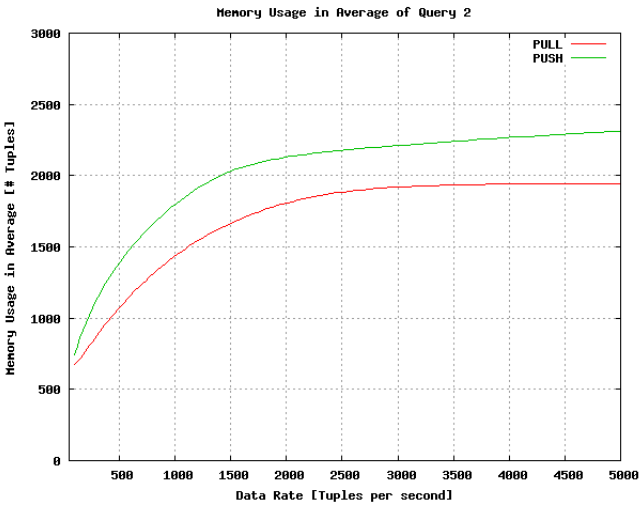

Figure 5.4: Pull vs. Push: Memory Usage of Q2 (non-contiguous, tumbling window), Result size: 3493

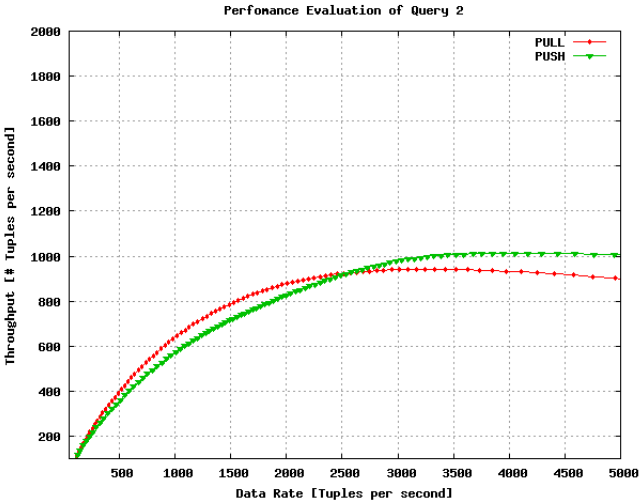

Figure 5.5: Pull vs. Push: Performance Evaluation of Q2 (non-contiguous, tumbling window), Result size: 3493

A different story it is when a sliding window is applied for the pattern search. Let's have a look at the results of Q3 (Figure 5.6, 5.7 and 5.8). Although the statements for contiguous pattern matching remain still valid (i.e. memory usage), the performance gain was less than 20 percent. This is because of the fact that whenever a match (or non-match) happens, all the tuples inside the semantic window are considered as unprocessed except the first one (which will be discarded by the slide). Thus the query processor does not have to request a tuple from the storage at that time comparing with Q1 (tumbling window) where a FSM has to request for data after every match in Pull mode.
Analog to Q1 the performance gain is also reflected in latency. It has to be mentioned that the time lags for Q1 and Q3 are different. Using a sliding window may increase the latency of a tuple a lot since at worst it processes the data in the semantic window several times before getting to a new tuple. In this example the latency in Q3 is multiplied by almost 600 times. But still one can recognize the same trends as for Q1.

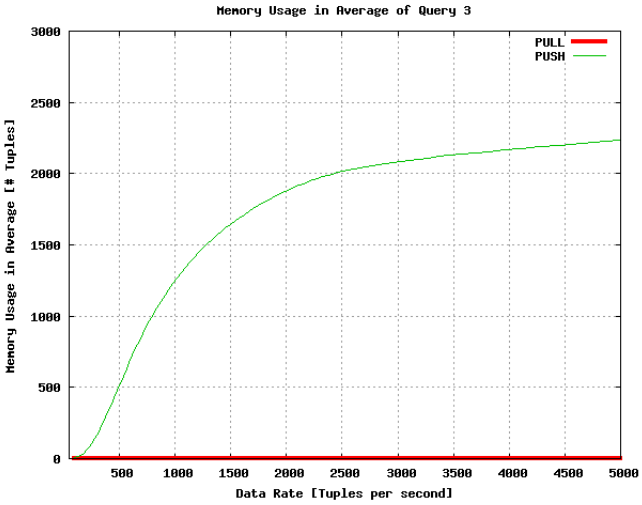

Figure 5.6: Pull vs. Push: Memory Usage of Q3 (contiguous, sliding window), Result size: 13083

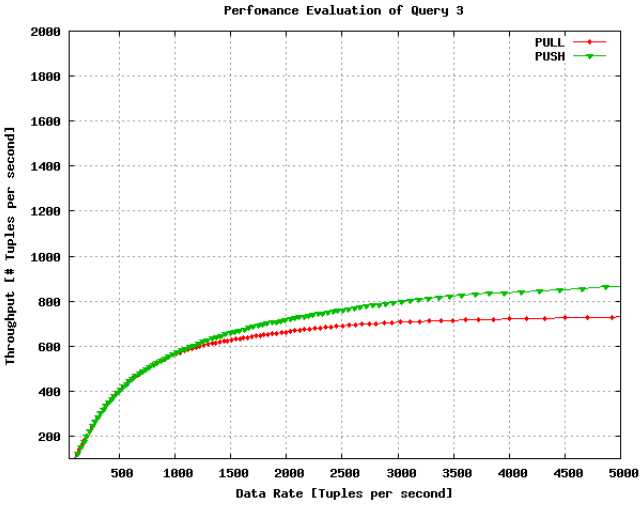

Figure 5.7: Pull vs. Push: Performance Evaluation of Q3 (contiguous, sliding window), Result size: 13083

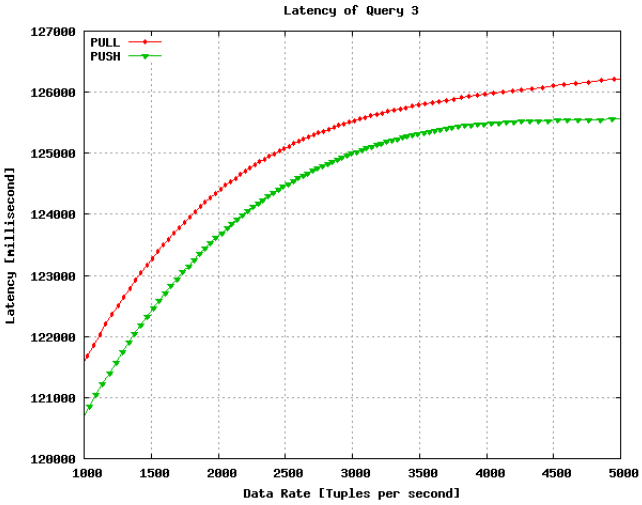

Figure 5.8: Pull vs. Push: Latency of Q3 (contiguous, sliding window), Result size: 13083

In Figure 5.9 and 5.10 the evaluation of Q4 are illustrated. Here again one can observe that Pull and Push behave very similar in case of non-contiguous pattern search because of the same reason as in Q2. But in addition to that the performance gain is also dampened by the "TO NEXT ROW" effect to around 5 percent. Table 5.1 shows the experiment results in summary.

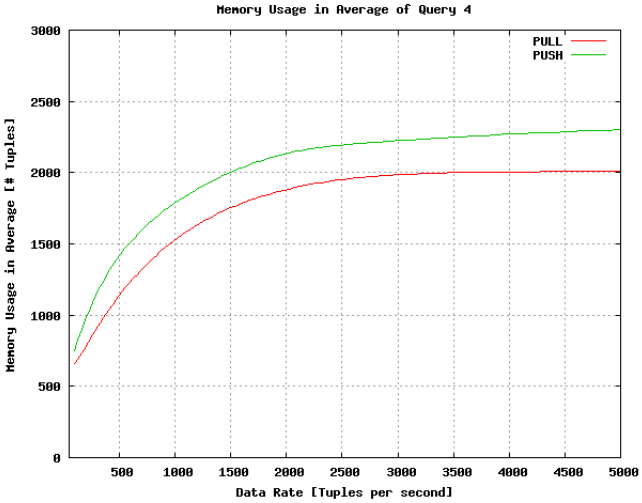

Figure 5.9: Pull vs. Push: Memory Usage of Q4 (non-contiguous, sliding window), Result size: 5180

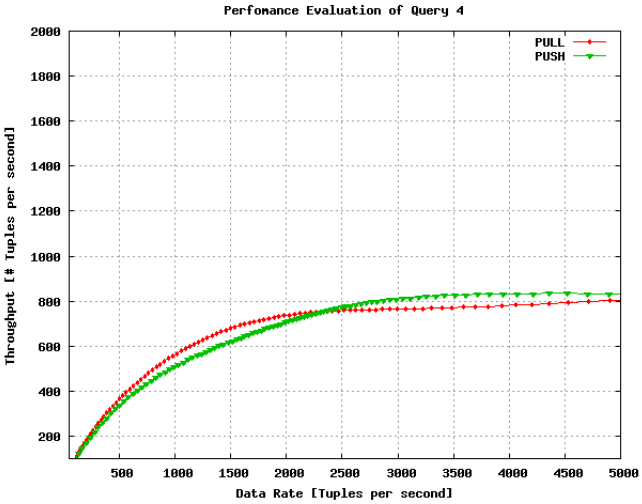

Figure 5.10: Pull vs. Push: Performance Evaluation of Q4 (non-contiguous, sliding window), Result size: 5180

Query	Sliding	Data Set	Memory	Throughput
			Push	Push
Q1	PAST LAST ROW	contiguous	+∼100%	+∼30%
Q2		non-contiguous	+∼20%	+∼12%
Q3	TO NEXT ROW	contiguous	+∼100%	+∼20%
Q4		non-contiguous	+∼20%	+∼5%

Table 5.1: Push compared to Pull in summary

5.3 Adaptive Switch vs. Pull/Push

As concluded from the previous comparison the performance and memory usage of Pull and Push are similar when pattern matching happens on a non-contiguous data set. Therefore to show the potential of Adaptive Switch only the queries Q1 and Q3 are evaluated. From the results we can see that by using the Adaptive Switch the memory usage can be kept very low as well (Figure 5.11 and 5.14) but the increase of performance vary from interval [0;∼ 30] percent (tumbling window) and [0;∼ 20] percent (sliding window)

(Figure 5.12 and 5.15). In terms of latency one can see that its curve takes course between Push and Pull as expected (Figure 5.13 and 5.16).

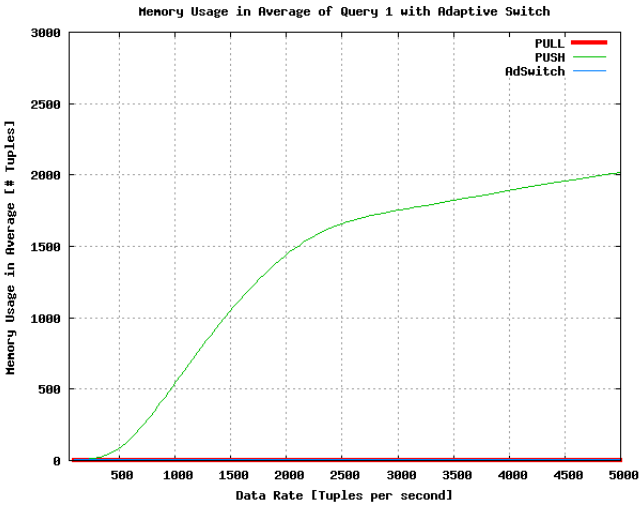

Figure 5.11: Adaptive Switch in action: Memory Usage of Q1 (contiguous, tumbling window), Result size: 7457

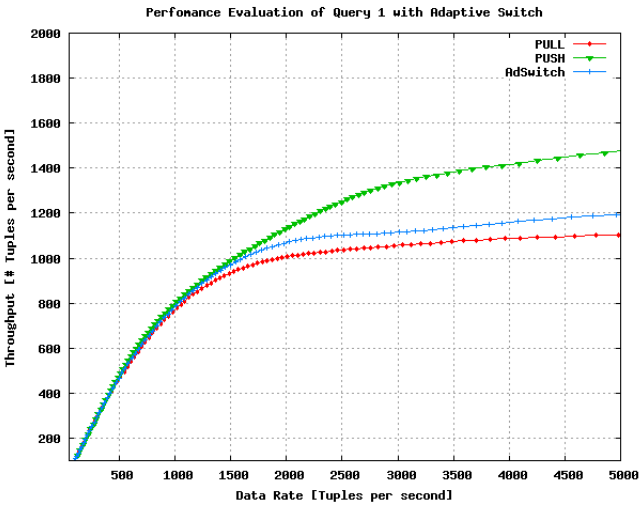

Figure 5.12: Adaptive Switch in action: Performance Evaluation of Q1 (contiguous, tumbling window), Result size: 7457

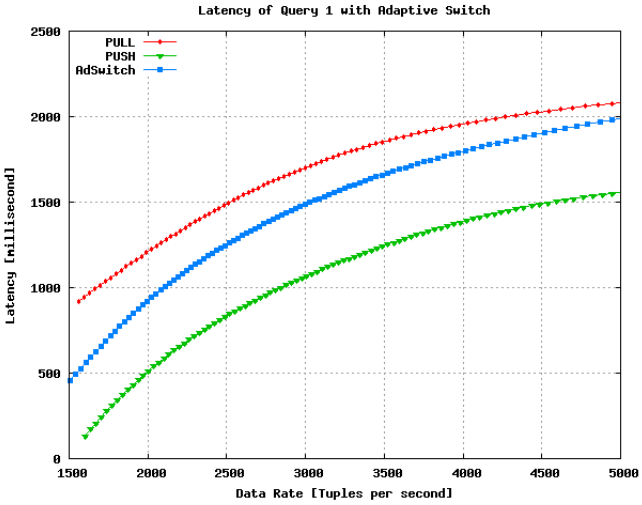

Figure 5.13: Adaptive Switch in action: Latency of Q1 (contiguous, tumbling window), Result size: 7457

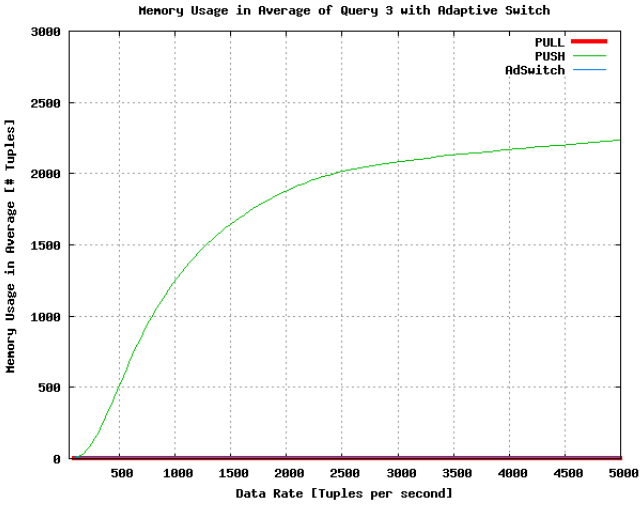

Figure 5.14: Adaptive Switch in action: Memory Usage of Q3 (contiguous, sliding window), Result size: 13083

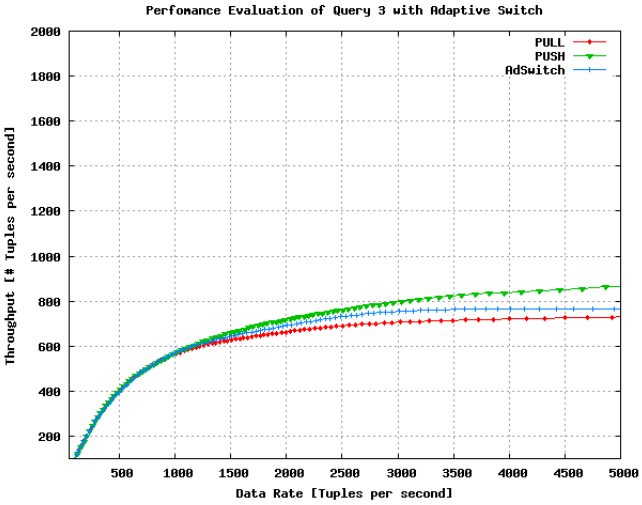

Figure 5.15: Adaptive Switch in action: Performance Evaluation of Q3 (contiguous, sliding window), Result size: 13083

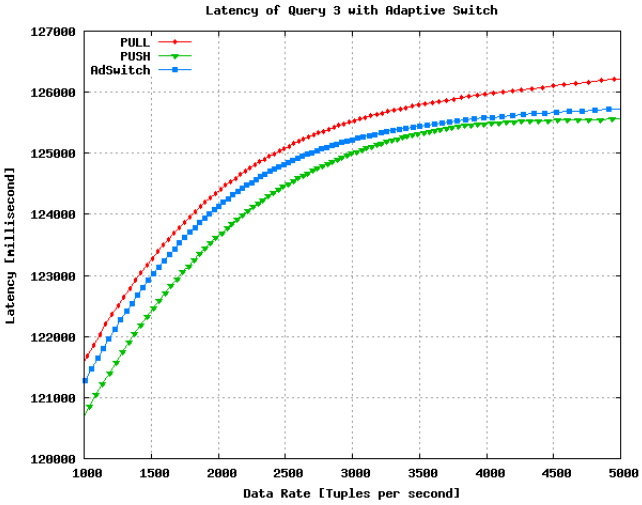

Figure 5.16: Adaptive Switch in action: Latency of Q3 (contiguous, sliding window), Result size: 13083

In combination with the specification of value U, the performance gain tends more to "Always Push". However the memory usage is still more than acceptable comparing to Push. The figures 5.17 show the memory and performance evaluation of Q1 with U varying from 1,2 and 10. They are focused on the data rate interval from [3000,5000] in order to see the difference between

the curves more clearer.

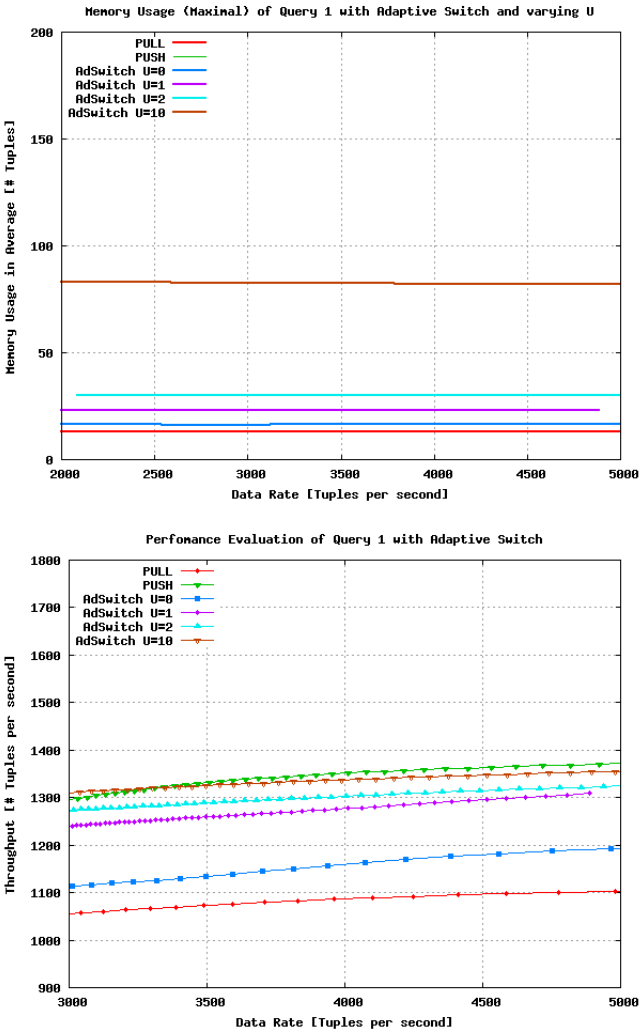

Figure 5.17: Adaptive Switch with varying U: Memory and Performance Evaluation of Q1 (contiguous, tumbling window), Result size: 7457

Chapter 6

Conclusions and Future Work

DejaVu used to handle streaming data and static data in the same way - whenever the Query Processing Engine needs data it tries to "pull" the tuples from the storage. But exactly this effects the processing time negatively - the more data is requested the clearer the latency a pull operation causes. By introducing a pushing mechanism where data is forwarded as soon as they arrive all the FSM's don't need to care about Input Handling anymore. In the view of the query processor the data is "magically" put into the InputHolders. Based on the cost models and also on the experiments one can see that separation of independent tasks like Pattern Detection and Input Handling increases the performance of processing considerably. Thus similar to Input Handling, reporting the matches should also happen in a separate layer like other CEP systems e.g. [9] and [12].
A problem that gives current CEP systems a headache is the memory constraint. Although they optimize the memory management e.g. by running an efficient Garbage Collector or reusing existing data structure, it's still not enough. There may be a point where the processing rate cannot keep up with the streaming rate and causes the arriving data to be buffered to nowhere. The system may still be able to allocate some bytes based on the concept of virtual memory and file swapping. But the access operations are slower since data has to be swapped from disc to memory and vice versa. In DejaVu it is not different. When the process memory is insufficient the same has to be done. However DejaVu is able to postpone this "worst-case" by using the Adaptive Switch. It tries to predict the query processing rate in advance and pushes only the needed amount of data at once. In this case only those tuples are allocated in the process memory area whereas the rest of the data stays in the Stream Storage. The memory household is more economic. Note that there is a big difference if data are in the process memory area e.g. InputHolders or in the Stream Storage. Although the store

resides in the shared memory (and therefore in the process memory area as well), its engine is able to redirect the data stream to the Archive Stream Store according to its usage and for example making it persistent. Still this feature leaves some question opened and requires some further research.
From the measurements one can conclude that the Adaptive Switch works well on a contiguous data set where a single FSM is working on one partition. But as it concerns multiple partitions with more than one runs, the selection of one InputHolder according to the global data distribution fails when data is uniformly distributed to the partitions. Besides that having multiple runs operating in a single thread reduces the performance dramatically. (A FSM has to wait for its turn until the other ones have finished theirs - although they could have run independently of each other.) Thus the introduction of multi-threaded runs does not only solve the performance problem but also allows Adaptive Switch working on each FSM separately. Another interesting aspect is to see how Adaptive Switch works when data is partitioned in a lower level. Currently whenever a FSM requests data from the Storage Engine, the record is routed to the corresponding runs - a side effect which may not be wanted by the concerned FSM's and disturbs the memory management of the particular runs.

Acknowledgments

First of all I want to thank Prof. Nesime Tatbul for her guidance and patience. She gave me the opportunity to work on the DejaVu project and even the chance to present a demo together with the DejaVu team in an academic conference. It was a once-in-a-lifetime experience and for that I am really grateful. Maybe with even more importance I would like to express special thanks to Nihal Dindar. She was a wonderful mentor during the past year and supported me in good and bad times. The input I received from her was invaluable. I also want to thank the other DejaVu team members - Baris Güç, Merve Soner and Asli Özal - for their effort and contribution to the system in general. Also many thanks go to Çağrı Balkesen and Roozbeh Derakhshan for their advice and encouragement. And last, but not least - special thanks to my girlfriend and my family for their patience and support during all these years.

Appendix A

Experimental Results

The following tables show the experimental results of Q1, Q2, Q3 and Q4 in numbers with the header described below:

- Data Rate - the input rate in tuples per second
- Avg Mem - the average memory usage in ♯ tuple allocations
- Max Mem - the maximal memory usage in ♯ tuple allocations
- Avg Queue - the average queue usage in ♯ tuple allocations
- Max Queue - the maximal queue usage in ♯ tuple allocations
- Result - the size of result set in tuples
- Time (ms) - the overall processing time in milliseconds
- Throughput - the throughput of the system in tuples per second
- Records - the size of input set in tuples
- Latency (only in certain sections) - the latency in milliseconds

A.1 Pull Mode

A.1.1 Contiguous Pattern Matching

Data Rate	Avg Mem	Max Mem	Avg Queue	Max Queue	Result	Time (ms)	Throughput	Records
98	3	13	1	85	7457	50901.96	98.23	5000
285	3	13	1	23	7457	17556.43	284.8	5000
461	3	13	3	41	7457	10885.86	459.31	5000
881	3	13	82	345	7457	5989.09	834.85	5000
1241	3	13	641	1277	7457	5081	984.06	5000
1584	3	13	1074	1965	7457	4854.87	1029.89	5000
2106	3	13	1297	2885	7457	4726	1057.87	5000
2552	3	13	1647	3453	7457	5122	976.16	5000
2995	3	13	1709	3576	7457	4640	1077.69	5000
3420	3	13	1788	3666	7457	4420	1131.15	5000
7793	3	13	2283	4562	7457	4540	1101.2	5000

Table A.1: Evaluation of Q1 in Pull Mode

Data Rate	Avg Mem	Max Mem	Avg Queue	Max Queue	Result	Time (ms)	Throughput	Records
99	3	11	0	52	13083	50808.84	98.41	5000
286	3	11	19	178	13083	17546.9	284.95	5000
466	3	11	26	114	13083	10859.47	460.43	5000
871	3	11	645	1371	13083	7477.35	668.69	5000
1224	3	11	1103	2421	13083	8674	576.43	5000
1544	3	11	1495	3133	13083	7497.35	666.9	5000
1765	3	11	2047	4160	13083	8328	600.35	5000
2078	3	11	1743	3687	13083	7065	707.67	5000
2544	3	11	1975	4038	13083	7171	697.21	5000
3030	3	11	1991	3995	13083	6911	723.43	5000
3334	3	11	2113	4254	13083	7012	713.07	5000
7586	3	11	2367	4746	13083	6660	750.77	5000

Table A.2: Evaluation of Q3 in Pull Mode

Latency

Data Rate	Latency
1558	920
2144	1324
2589	1577
3049	1795
3688	1953
4225	2084
7063	2213

Table A.3: Latency of Q1 in Pull Mode

Data Rate	Latency
1574	124058
2106	124824
2585	125424
3006	125609
3654	126040
4060	125961
6811	126646

Table A.4: Latency of Q3 in Pull Mode

A.1.2 Non-Contiguous Pattern Matching

Data Rate	Avg Mem	Max Mem	Avg Queue	Max Queue	Result	Time (ms)	Throughput	Records
98	674	1041	0	60	3493	51351.76	97.37	5000
282	808	1217	5	193	3493	18640.79	268.23	5000
464	1176	1940	11	225	3493	11993.29	416.9	5000
885	1417	2695	1	50	3493	7624.48	655.78	5000
1257	1646	3199	3	97	3493	6334.2	789.37	5000
1547	1786	3454	9	140	3493	5843.65	855.63	5000
2122	1896	3703	166	576	3493	5331.05	937.9	5000
2571	1925	3750	261	1105	3493	5291.28	944.95	5000
3009	1945	3763	624	1685	3493	5248.72	952.61	5000
3411	1947	3771	678	1622	3493	4993.04	1001.39	5000
7208	1948	3766	1871	3743	3493	6321.12	791	5000

Table A.5: Evaluation of Q2 in Pull Mode

Data Rate	Avg Mem	Max Mem	Avg Queue	Max Queue	Result	Time (ms)	Throughput	Records
98	659	986	0	47	5180	51583.78	96.93	5000
241	792	1277	0	17	5180	21920.26	228.1	5000
460	1233	2303	21	366	5180	12951.07	386.07	5000
875	1586	3068	1	45	5180	8575.19	583.08	5000
1252	1769	3489	9	120	5180	7180	696.38	5000
1592	1840	3670	35	283	5180	6797.87	735.52	5000
2156	1960	3859	314	834	5180	6314	791.87	5000
2570	1998	3893	613	1404	5180	6250	800.03	5000
3042	2008	3903	508	1356	5180	7321	682.97	5000
3238	2009	3909	926	2137	5180	6174	809.84	5000
7246	2008	3903	1826	3823	5180	5981	835.93	5000

Table A.6: Evaluation of Q4 in Pull Mode

A.2 Push Mode

A.2.1 Contiguous Pattern Matching

Data Rate	Avg Mem	Max Mem	Avg Queue	Max Queue	Result	Time (ms)	Throughput	Records
98	8	197	0	3	7457	51328.03	97.41	5000
278	7	41	0	10	7457	18078.12	276.58	5000
426	10	63	0	21	7457	11807.11	423.47	5000
608	65	371	0	24	7457	8490.27	588.91	5000
695	249	587	0	47	7457	7561.15	661.28	5000
793	70	280	0	11	7457	6366.84	785.32	5000
1237	1096	2145	0	25	7457	5220.51	957.76	5000
1534	1079	2685	0	34	7457	4773	1047.63	5000
1868	1467	3151	0	32	7457	4561.09	1096.23	5000
2324	1912	3874	1	105	7457	4280.07	1168.21	5000
2434	1661	3405	1	37	7457	3886.25	1286.59	5000
2676	1645	3330	2	66	7457	3654	1368.49	5000
3200	1782	3662	2	115	7457	3642	1372.69	5000
3675	1907	4019	1	55	7457	3596	1390.39	5000
6815	2206	4368	55	483	7457	3140	1592.16	5000

Table A.7: Evaluation of Q1 in Push Mode

Data Rate	Avg Mem	Max Mem	Avg Queue	Max Queue	Result	Time (ms)	Throughput	Records
98	5	133	0	4	13083	51206.78	97.64	5000
360	46	218	0	13	13083	13942.3	358.62	5000
503	491	985	0	17	13083	10539.32	474.41	5000
807	1338	2637	0	38	13083	9260.1	539.95	5000
977	1574	3163	0	26	13083	8379.19	596.72	5000
1568	1744	3660	0	49	13083	6923.65	722.16	5000
2107	1902	3936	4	75	13083	6494	769.99	5000
2163	2078	4253	1	47	13083	7494.28	667.18	5000
2566	2018	4118	1	58	13083	6283.92	795.68	5000
3009	2119	4310	1	77	13083	6070.07	823.71	5000
3202	2126	4293	2	87	13083	5963	838.52	5000
7587	2399	4760	7	217	13083	5469	914.19	5000

Table A.8: Evaluation of Q3 in Push Mode

Latency

Data Rate	Latency
1604	134
2172	778
2660	950
3200	1183
3801	1351
4209	1539
5718	1608

Table A.9: Latency of Q1 in Push Mode

Data Rate	Latency
1615	123362
2236	124400
2698	125014
3116	124991
3838	125935
3860	125351
6180	125649

Table A.10: Latency of Q3 in Push Mode

A.2.2 Non-Contiguous Pattern Matching

Data Rate	Avg Mem	Max Mem	Avg Queue	Max Queue	Result	Time (ms)	Throughput	Records
97	742	1139	0	5	3493	52431.23	95.36	5000
278	1180	2070	0	13	3493	19352.94	258.36	5000
458	1457	2755	0	11	3493	14386.26	347.55	5000
830	1829	3571	0	4	3493	8736.87	572.29	5000
1248	1949	3864	0	0	3493	7039.14	710.31	5000
1532	2187	4349	1	53	3493	6605.33	756.96	5000
1617	2028	4075	0	10	3493	7430.87	672.87	5000
1656	2097	4156	0	1	3493	6295.29	794.24	5000
2144	2184	4333	3	96	3493	5643.69	885.94	5000
2487	2218	4368	2	123	3493	5333.54	937.46	5000
2983	2187	4429	0	25	3493	4987.61	1002.48	5000
3132	2205	4483	0	25	3493	4983.15	1003.38	5000
3460	2242	4515	5	91	3493	4792.7	1043.25	5000
3870	2295	4606	2	128	3493	4986.5	1002.71	5000
6241	2339	4692	23	339	3493	4989.27	1002.15	5000

Table A.11: Evaluation of Q2 in Push Mode

Data Rate	Avg Mem	Max Mem	Avg Queue	Max Queue	Result	Time (ms)	Throughput	Records
98	753	1191	0	3	5180	51947.74	96.25	5000
420	1519	2998	0	4	5180	14647.47	341.36	5000
701	1760	3492	0	30	5180	10428.05	479.48	5000
1278	2036	3964	0	59	5180	7888.75	633.81	5000
1421	2063	4101	0	19	5180	8754.08	571.16	5000
2025	2189	4347	0	0	5180	6812.58	733.94	5000
2204	2225	4524	1	107	5180	6715	744.57	5000
2241	2153	4308	2	127	5180	6525.74	766.2	5000
2809	2210	4444	0	37	5180	6022.59	830.21	5000
3154	2250	4525	3	147	5180	5999.53	833.4	5000
3722	2286	4594	4	183	5180	5792	863.26	5000
7578	2356	4739	51	478	5180	6226	803.05	5000

Table A.12: Evaluation of Q4 in Push Mode

A.3 Adaptive Switch

A.3.1 U=0

Data Rate	Avg Mem	Max Mem	Avg Queue	Max Queue	Result	Time (ms)	Throughput	Records
97	4	15	3	137	7457	51685.17	96.74	5000
187	5	17	8	243	7457	26778.18	186.72	5000
349	6	16	23	376	7457	14369.79	347.95	5000
613	6	17	7	67	7457	8198.17	609.89	5000
840	7	18	31	145	7457	6000.82	833.22	5000
1125	7	16	1318	2641	7457	6219.45	803.93	5000
1522	7	16	887	1587	7457	4579.9	1091.73	5000
1810	7	17	1149	2345	7457	4309.48	1160.23	5000
2456	7	16	1389	2675	7457	3959	1262.93	5000
2885	7	16	1588	3214	7457	5587	894.91	5000
3163	7	16	1906	3842	7457	4385	1140.19	5000
3515	7	17	1838	3651	7457	4006	1248.13	5000
7229	7	16	2298	4602	7457	4205	1188.94	5000

Table A.13: Evaluation of Q1 with Adaptive Switch

Data Rate	Avg Mem	Max Mem	Avg Queue	Max Queue	Result	Time (ms)	Throughput	Records
96	4	13	4	147	13083	52137.61	95.9	5000
271	5	14	2	42	13083	18511.46	270.1	5000
450	6	14	18	120	13083	11145.35	448.62	5000
855	6	13	730	1605	13083	7935.95	630.04	5000
1210	6	13	1440	2831	13083	7868.55	635.44	5000
1506	6	13	1508	3096	13083	7212.54	693.24	5000
1792	6	14	1744	3444	13083	8566.06	583.7	5000
2120	6	13	1783	3539	13083	6774.67	738.04	5000
2486	6	13	1848	3670	13083	6492.91	770.07	5000
2859	6	13	2025	4131	13083	6685.82	747.85	5000
3286	6	13	2059	4095	13083	6426.35	778.05	5000
3670	6	13	2118	4221	13083	6547.52	763.65	5000
6531	6	14	2352	4675	13083	6552.91	763.02	5000

Table A.14: Evaluation of Q3 with Adaptive Switch

Latency

Data Rate	Latency
1508	455
2055	1076
2477	1270
2815	1452
3480	1692
3725	1785
5298	2039

Table A.15: Latency of Q1 with Adaptive Switch

Data Rate	Latency
1526	123687
2095	124571
2500	125070
2863	125232
3527	125641
4021	125671
6825	125884

Table A.16: Latency of Q3 with Adaptive Switch

A.3.2 U=1

Data Rate	Avg Mem	Max Mem	Avg Queue	Max Queue	Result	Time (ms)	Throughput	Records
1480	34	83	662	1129	7457	4133.31	1209.68	5000
2107	35	83	1050	2152	7457	3852.67	1297.8	5000
3017	35	83	1569	3220	7457	3824.47	1307.37	5000
3696	34	81	1889	3697	7457	3703.46	1350.09	5000
6835	36	83	2186	4487	7457	3631	1376.97	5000

Table A.17: Evaluation of Q1 with Adaptive Switch, U=1

A.3.3 U=2

Data Rate	Avg Mem	Max Mem	Avg Queue	Max Queue	Result	Time (ms)	Throughput	Records
1480	34	83	662	1129	7457	4133.31	1209.68	5000
2107	35	83	1050	2152	7457	3852.67	1297.8	5000
3017	35	83	1569	3220	7457	3824.47	1307.37	5000
3696	34	81	1889	3697	7457	3703.46	1350.09	5000
6835	36	83	2186	4487	7457	3631	1376.97	5000

Table A.18: Evaluation of Q1 with Adaptive Switch, U=2

A.3.4 U=10

Data Rate	Avg Mem	Max Mem	Avg Queue	Max Queue	Result	Time (ms)	Throughput	Records
1480	34	83	662	1129	7457	4133.31	1209.68	5000
2107	35	83	1050	2152	7457	3852.67	1297.8	5000
3017	35	83	1569	3220	7457	3824.47	1307.37	5000
3696	34	81	1889	3697	7457	3703.46	1350.09	5000
6835	36	83	2186	4487	7457	3631	1376.97	5000

Table A.19: Evaluation of Q1 with Adaptive Switch, U=10

Bibliography

[1] Bayesian network. http://en.wikipedia.org/wiki/Bayesian_network/.

[2] Bézier curve. http://en.wikipedia.org/wiki/Bezier_curve/.

[3] Conditional independence. http://en.wikipedia.org/wiki/Conditional_independence/.

[4] MySQL. http://www.mysql.com/.

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

[6] POSIX. http://www.pasc.org/.

[7] S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR Conference*, Asilomar, CA, January 2003.

[8] W. M. Clement T. Yu. *Principles of database query processing for advanced applications*. Morgan Kaufmann, 1998.

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

[10] N. Dindar. Pattern Matching over Sequences of Rows in a Relational Database System, September 2008.

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

[12] D. Gyllstrom, E. Wu, H. Chae, Y. Diao, P. Stahlberg, and G. Anderson. SASE: Complex Event Processing over Streams (Demo). In *CIDR Conference*, Asilomar, CA, January 2007.

[13] P. Lau. A Router-based Implementation of the 'PARTITION BY'-clause in DejaVu, April 2009.

[14] S. Madden and M.Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *IEEE ICDE Conference*, 2002.

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