Dynamic Modification of Continuous Queries

Master Thesis

Tahmineh Sanamrad
Information Systems Group
Computer Science Department
ETH Zurich
sanamrat@student.ethz.ch

July 15, 2009 - January 15, 2010

Supervised by:
Dr. Peter M. Fischer
Kyumars Sheykh Esmaili
Abstract

Nowadays, sophisticated near-real time data stream queries run on massive and continuous amount of data, resulting from transactions and measurements. However, these Long-Running Data Stream Queries might not correspond to user requirements at some point and need to be modified. Considering the distributed, continuous and near-real time nature of data streams, modifying a query needs to be conducted through a reliable model. So far, there has not been any model developed by the streaming community to classify continuous query modification methodic. Hence, in this thesis we will introduce a new model to capture the problems rising when modifying a long-running query on a data stream whereas fulfilling the application’s correctness requirements. This model is based on a simple start-stop model, and gets more complicated resulting in several change models. Each of these change models tries to cover some correctness criteria. There are basically two groups of correctness criteria: Safety and Liveness. The former minimizes loss, duplicity and disorder among output elements and the latter guarantees progress of the new version of the query and termination of the old one.
I would like to dedicate this thesis to Michel, who made it possible for me to come to Zurich and study at ETH, and to my dear mother for her love and continuous faith in me.
Acknowledgments

Special thanks to Dr. Fischer for his continuous and passionate supervision.
Special thanks to Prof. Tatbul and Prof. Kossmann for their feedbacks and support.
Thanks to Kyumars for his supervision and also his sense of humor which was necessary for me to keep going.
Thanks to Martin Seiler for his brilliant ideas and inputs.
And finally very special thanks to Derivative Partners AG, the company I worked for, during my studies, for being extremely flexible and supportive toward my study plans.
## Contents

1 Introduction 10  
1.1 Motivation .................................................. 10  
1.2 Usecases .................................................... 11  
1.3 Problem Definition ........................................ 12  
1.4 Related Work ............................................... 12  
1.5 Outline ..................................................... 14

2 Basic Query Lifecycle Model 16  
2.1 Basic Definitions on Stream 17  
2.1.1 Continuous Query ...................................... 17  
2.1.2 Stream .................................................. 18  
2.1.3 Stream Element ........................................ 18  
2.1.4 Sequence Number Generator .......................... 19  
2.2 Mapping Functions ......................................... 20  
2.2.1 Basic Mapping Functions ............................... 20  
2.2.2 Stream Mapping Functions .............................. 21  
2.3 Basic Control Elements ................................. 23  
2.3.1 Start .................................................... 23  
2.3.2 Stop .................................................... 25  
2.3.3 Interactions of Basic Control Elements .............. 26

3 Query Modification Model 32  
3.1 Query Modification .......................................... 32  
3.2 Correctness Criteria ....................................... 33  
3.3 Safety Criteria ............................................. 33  
3.3.1 Loss ...................................................... 34  
3.3.2 Disorder ............................................... 36  
3.3.3 Duplicates .............................................. 37  
3.4 Liveness Criteria .......................................... 37  
3.4.1 Progress of the new version of the Query .......... 38  
3.4.2 Termination of the old version of the Query ....... 38  
3.5 Complex Control Elements ................................. 38  
3.5.1 Immediate Change ..................................... 38  
3.5.2 Drain Change ........................................... 41  
3.5.3 Delayed Drain Change ................................. 44  
3.5.4 Graceful Change ....................................... 48  
3.6 Summary of Correctness Criteria in Change Policies 51
4 Refinement and Application of Model

4.1 Operators .................................................. 52
   4.1.1 Stateless Operators .................................. 53
   4.1.2 Stateful Operators ................................. 55
4.2 Application of Model ..................................... 56

5 Implementation .............................................. 60

5.1 XQuery as Continuous Query Language ............. 60
   5.1.1 MXQuery as Data Stream Management System .... 60
5.2 DMCQ Component ........................................ 61
   5.2.1 DMCQ Architecture ................................. 61

6 Conclusion .................................................. 66

6.1 Future Work .............................................. 66
# List of Figures

2.1 The basic elements of the Stream Processing Model .......................... 19
2.2 Sequence Number Generator assigns to each input element a sequence number .......................... 20
2.3 \textit{sequenceAt}(x_i) \text{ returns the sequence number of a given element}
and \textit{elementAt}(i) \text{ returns the element corresponding to a sequence number} .......................... 21
2.4 \textit{depends}(y_j) \text{ function} ........................................ 22
2.5 \textit{contributes}(x_i) \text{ function} ....................................... 23
2.6 The difference between Immediate Start and Smooth Start ................. 24
2.7 The difference between Immediate Stop and Drain Stop ..................... 26
2.8 Query State Transitions .................................................. 28
2.9 By Reversing the Input and using the inverse of each control element, reverse of output stream can be created ........................................ 29
2.10 The big image of Basic control elements and their functionality .......... 30
3.1 Simple Loss in the Observed Stream and Reference Stream ............... 35
3.2 The Observed Stream, when an immediate change element comes along and the Reference Streams ................................. 36
3.3 The Immediate Change Control Element splits the Output to two completely non-overlapping Sets ........................................................ 39
3.4 The Drain Change Control Element implementations according to definition 3.5.2 .................................................. 42
3.5 Strict Delayed Drain Change in Action .................................. 45
3.6 Smooth Delayed Drain Change in Action ................................ 45
3.7 In both Delayed Drain Change policy loss can occur ....................... 47
3.8 Graceful Change in Action ................................................. 49
4.1 Interactions between operators in a query forms a DAG ....................... 53
4.2 Selection Operator in Action: Select all the input tuples, with a value bigger than 1 .................................................. 54
4.3 Projection Operator in Action .............................................. 55
4.4 Immediate Stop followed by Immediate Start according to our model ........................................................................ 57
4.5 Comparison of Model Output and practical output by a consecutive istop-istart .......................................................... 58
4.6 The Drain Stop policy does not allow any window being produced after Drain Stop element .................................................. 59
5.1 The DMCQ components of the Basic Query Lifecycle Elements ........ 61
5.2 Finite State Machine of a Gatekeeper . . . . . . . . . . . . . . . 62
5.3 The full architecture of DMCQ component. . . . . . . . . . . . . 64
2.1 Query State Transition Table
2.2 Summary of Basic Control Elements

3.1 Safety guarantees offered by Immediate Change
3.2 Liveness guarantees offered by Immediate Change
3.3 Safety guarantees offered by Drain Change
3.4 Liveness guarantees offered by Drain Change
3.5 Safety guarantees offered by Delayed Drain Change
3.6 Liveness guarantees offered by Delayed Drain Change
3.7 Safety guarantees offered by Graceful Change
3.8 Liveness guarantees offered by Graceful Change
3.9 Safety guarantees offered by each Change Policy
3.10 Liveness guarantees offered by each Change Policy
Chapter 1

Introduction

Nowadays, the amount of data generated from different sources is tremendous. Storing data, like what traditional databases do, appears not to be a solution to deal with applications like financial analytics or sensor networks. The data created by these sources are updated quickly, maybe in seconds or milliseconds, and the old data does not hold valuable information anymore since it is outdated. The idea of letting data flow has created a new field of research, named Data Streams. The concept of streaming data can be applied to lots of cases. For example, RSS feeds are continuous infinite data pulled every second to be by the subscriber. Another example is the idea of a paperless office. In most big organizations today, all the interactions between employees are getting more and more electronic based; these interactions can also be seen as an infinite data generation source. Thus, it is getting more important to deal with different dimensions of these systems and explore their properties, in order to propose and guarantee the correctness of methods manipulating data in a stream management system.

1.1 Motivation

As already discussed, data streams were a proper solution to manage large amount of data in a timely acceptable manner by running continuous queries on them, resulting in minimizing the storage requirements. This amount of data is produced due to dynamic and turbulent nature of their environments. For example common uses of data streams are in Sensor Networking and Financial Analysis. Both these environments are very dynamic and chaotic, meaning that no prediction can be done, and thus no static solution can be determined to manage these systems. These continuously changing environments should be able to reliably adapt to evolution of data streams (change of data characteristics on the stream), and evolution of query requirements at some point to maintain their goals, or in other words to maintain their steady state.

Now that the evolution of data characteristics and also query requirements is an inevitable fact of the environment that a data stream has been thought for, the ability to properly cope with the newly emerged requirements is of a
great importance. In order to maintain a steady state in a turbulent and dynamic environment, data stream systems should be flexible and ready to undergo changes, with minimum effort and damage to the system. In other words, the non-functional requirements of the system should be guaranteed. For example, in case of transactions, there are a set of properties named ACID (Atomicity, Consistency, Isolation, Durability) which guarantees that transaction has been processed reliably. In our case, we should also define a set of correctness criteria which guarantee a reliable change.

In this thesis, we introduce a model which enables a change being processed reliably and efficiently by a data stream system.

1.2 Usecases

In many stream processing applications, it is desirable to change certain attributes of the query at runtime[1]. Further on, we will name and explain some of the usecases where having a robust and reliable query modification model is necessary:

**Security and Compliance Monitoring:** Suppose all Automatic Teller Machines of a Bank, react to three failed logins by a single user in thirty minutes in that they block his card. In case of a card theft, it is even more important for these machines to be able to accurately notify the Bank center in order to lock the card and avoid further abuse. According to the policy mentioned earlier, our user manages to establish two failed login in 10 minutes. At this moment the bank policy changes, to detect three failed logins in one hour and then blocks the card. Here we have a user, who had 2 failed login in 10 minutes and a changed policy of bank. The important question here is: if the query changes at this moment, will all the previous states being discarded resulting in giving the possibility to have even more than 3 failed login in half an hour, or the states will be extended, or the change will be deferred until all uncompleted sates are done? Our model answers to all these questions in a neat way, by showing different possible models to solve this problem by guaranteeing a certain and expected behavior in case of change.

**Sensor Networking:** Consider a set of sensors, spread in a jungle. After a while receiving some reports about a possible fire spread around, the data controller decides to change the running query in order to gather up detailed and accurate data about the locality and cause of the fire. Since in this usecase the data should be received as fast as possible to avoid more spreading of the fire, what kind of change policies and guarantees are required? Certainly waiting for the uncompleted states from the current query can be dangerous and resource consuming in contrast to the security monitoring usecase.

**Data Mining:** Suppose there is a data mining algorithm running on a data stream, resulting in financial indices which describe certain aspects of data. After a while, a simple change of the ticker symbol can cause inconsistency in the
outcome of the data mining algorithm. The data mining algorithm had to collect a lot of data in order to run its model and deliver results. Now, the question is how could we change this query without discarding all the states gathered by the system so far? Immediately changing the query, will discard all of its states. Waiting for the query to finish its job, can take a very long time. Moreover, it may never end. So what policy should we choose to apply the change?

Load Management: Suppose we have a system which may have high loads sometimes and not be reachable therefore, in order to make this system accessible and responsive, we may want to replace a query running on it, with its less accurate version that uses less resources. Again in case of a light load on the system, the query can be replaced by its more accurate versions. This query may have several versions, tailored for many different kinds of loads. The policy to switch between these versions is dependent on the system load. In a military stream application from MITRE, they wish to switch to a cheaper query when the system is overloaded. In order to implement this, the operators composing the running query should be altered. Such manual substitutions impose high overhead and are slow to take effect as the new query network starts with an empty state. Our goal is to support low overhead, fast, and automatic modifications [1].

1.3 Problem Definition

In any type of application which uses a Data Stream Management System the need for a change in query may sooner or later appear. The depth of these changes can vary from simply alter the operator parameters (e.g., window size, filter predicate), whereas sometime it calls for altering the operators that compose the running query [1], which could end up with more complexity. What kind of strategies we can have, and what kind of guarantees each strategy can give the user is a field that has not yet been explored much. In this thesis, by introducing usecases the problem becomes clear. There are several approaches for dynamically modifying a long running query at run-time. Each approach guarantees certain correctness criteria and violates others. Depending on the application running on a stream there is not straightforward which policy to choose and what could be guaranteed. By categorizing the possible approaches when dynamically modifying a query we enter a new phase in making a more reasonable decision by knowing the pros and cons of each policy and willingly pay for the tradeoffs of our decision.

1.4 Related Work

The approach to change a query in our suggested system, has been influenced from the idea of putting punctuations in a stream in [13]. The similarity of the proposed model of change in this thesis to the punctuation model in [13], is that we also put symbols indicating start, stop or change of the query, in the
stream; thus, as the input stream is being processed and executed, the involving operators, which are mainly blocking operators, act properly to each special predefined punctuation symbol, which are referred in my thesis as control elements and implement the expected behavior. The punctuation proposed and used in [13] are mainly to deal with the inefficiency of stream processing systems when dealing with infinite data, mainly boosting up the performance of join and other blocking operators, by putting special symbols in the stream to indicate a state is finished and should not take up resources anymore because there is no data arriving which the blocking operator requires.

The other work that at the first sight seems related to our work is Stop and Restart style execution for Long running queries in [6]. However, in [6], the main concept is to store states of a long-running query in a way that minimizes the resource consumption by the query and simultaneously makes it possible to restart the query from where it had left off. They [6] refer to it as a checkpointing problem. Their solution is mainly to choose some tuples to store in order to make a query restartable and also minimize its resource consumption. In our suggested model, there is no concept of storing input tuples, however, using the methods in [6] could be an expansion to our model if we would further consider pause and restart issues. Dealing with pause and restart issues are considered as future work.

In the High Availability paper, [7], they are dealing also with correctness requirements and guarantees which quite resembles our model. However, in [7] they are suggesting high-availability algorithms for distributed stream processing and different recovery models, which in our case is of less concern. The similarity of our model and [7] is that we are somehow dealing with the same correctness concepts, but our model is for a single system and having no concerns about systems failures and recovery plans. Nevertheless, dealing with change in a distributed environment is also a future work we like to cover.

Finally, there is a quite similar topic in [14]. Dynamic plan migration is concerned with the on-the-fly transition from one continuous query plan to a semantically equivalent yet more efficient plan. Migration is important for stream monitoring systems where long-running queries may have to withstand fluctuations in stream workloads and data characteristics [14]. In another paper [8], algorithms are suggested to detect the above mentioned fluctuations, namely stream workloads and data characteristics. In [14] their main concern is to cope with the changes in stream workloads and data characteristics which causes a sub-optimal performance of currently running query, which leads to an optimization technique by rearranging the query plan. There are still a lot of similarities seen between their model and our model. We both have same correctness concerns dealing with losses and duplicates, and the strategies to replace a query plan with the query plan has some similarities. The only difference is that they are dealing with the same query all this time, but our model is dealing with a completely other query after replacement of the query plan. Thus, there are no assumptions we can make about the new coming query.
1.5 Outline

The outline of the thesis is as follows. In chapter 2 we introduce the basic query lifecycle model, which is composed of fundamental concepts and basic definitions. In chapter 3, we use the basic query lifecycle model as the building block to construct the query modification model upon it. Additionally, in chapter 3, we will introduce the correctness requirements and how our model reacts toward each correctness criteria with proofs. In chapter 4, we refine our model from query to operator level to check for composability and implementability of our model toward existing data stream management systems. As this model has been also implemented on MXQuery, we demonstrate the basic query lifecycle architecture and query modification architecture and the interactions between different APIs. In chapter 6, we conclude and list the future work.
Chapter 2

Basic Query Lifecycle Model

To build up a resilient model from scratch, the fundamental concepts has to be defined, and a solid ground has to be established to avoid the rest of the model trembling down. In this chapter we will first define, what all the keywords being frequently repeated in the first chapter mean to gain a common understanding. Then, we will introduce new concepts, which will serve as a linking bridge between well known basic concepts and our suggested model. In the end, we will begin with our model by introducing query lifecycle. According to our model, a query may encounter different events in its lifecycle and thus it has to know how to smoothly and robustly interact with them.

In order to influence a query, we have decided to put additional elements among the input stream tuples. Inspired by the idea of punctuated stream presented in [13], we also introduce the events happening in a query’s lifecycle as being punctuated in the stream, among the input tuples. In this case as the query processes input tuples to produce output results, it may also encounter specific event elements, known as control elements 2.1.3. In this case, instead of producing result and processing them as a normal data, the query changes its behavior depending on the action a certain control element triggers. To sum up, instead of explicitly telling query what to do, the query will find it out itself while processing the input stream.

In our case, the punctuated approach gives us a lot of accuracy and determinism, because we can exactly say, from which element we want the event to happen. Moreover, it makes possible to make very accurate event definitions, in other words we gain mathematical convenience, otherwise we had to use the concept of time, which could have made everything very complex, each item on the stream had to have a timestamp attached to it; thus one problem was which timestamp to take: application timestamp, original timestamp, system timestamp or our own-defined timestamp. Afterwards the real pain begins in the distributed environments where even more and more synchronizations were
required. After all, in case of an event being fired, timestamp of items should have been compared to the timestamp of event causing a lot of overhead, and even more complex than that is the unwanted disorder in the arriving tuples if they are time stamped by the source, in order to find out when a query should change its behavior. Dealing with temporal logic in the definitions following, would have caused lots of unnecessary complexity and finally exhaustion of the model making the model unpractical. Choosing a punctuated stream for our design, we could also use sequence numbers to provide total order. The sequence numbers introduce additional determinism into the model.

2.1 Basic Definitions on Stream

In this section we will define the terms being frequently repeated so far, in order to share our perspective with the reader.

2.1.1 Continuous Query

Due to the continuous nature of data streams, they are typically queried using long-running continuous queries rather than the traditional one-time queries [3]. One-time queries (a class that includes traditional DBMS queries) are queries that are evaluated once over a point-in-time snapshot of the data set, with the answer returned to the user. Continuous queries, on the other hand, are evaluated continuously as data streams continue to arrive[4].

Definition (Continuous Query) A continuous query is a query that is logically issued once but run forever. At any point of time, the answer to a continuous query reflects the elements of the input data streams seen so far, and the answer is updated as new stream elements arrive[3, 12].

Conventionally, in this thesis, continuous query will be referred to as $Q$.

2.1.1.1 Query Identity and Expression

Our model supports having multiple queries in the system. Thus each query is composed of several attributes:

$$Q = Id \times Expression$$

Each query is being identified by a Query Identifier. The identity of a Query is opaque, meaning that it is not value based, but rather refers to the owner of the query. Moreover, the identity of a query remains the same in the whole query lifecycle 2.3.3. On the other hand the expression of a query is the query logic.

Proposition 2.1.1. Query $Q$ and $Q'$ are the same, if and only if they have same identifier (id).

$$Q \neq Q' \Leftrightarrow Q_{id} \neq Q'_{id}$$
Corollary 2.1.2. Query Q having the same expression as Query Q’, does not necessarily imply that Query Q is the same as Query Q’.

\[ Q_{expression} = Q’_{expression} \iff Q = Q’ \]

2.1.2 Stream

Definition (Stream) Stream S is an unbounded sequence of Stream Elements (see section 2.1.3 for stream element definition). Unlike the definition in [2], which defines a stream as an unbounded bag (multiset) of pairs \(<s, \tau>\) where \(s\) is a tuple and \(\tau\) is the timestamp, we define a stream as a sequence of elements. Sequence by nature includes the concept of a total order between its elements; therefore there is no need to define a timestamp to induce order.

In this thesis, we frequently refer to Input Stream I and Output Stream O. Nevertheless, these two concepts are practically the same and have the same functional characteristics (see 2.1.2). The only difference, which matters to us is that Input Stream and Output Stream are mapped to each other using Stream Mapping Function 2.2.2. Following notations are crucial to remember throughout the thesis to be able to follow further definitions.

- **Stream** is referred to as \(S\).
- **Input Stream** is referred to I and it serves as the input for query \(Q\).
- **Output Stream** is referred to O and it serves as the output or in other words the result of query \(Q\).

2.1.3 Stream Element

Now that we know what a punctuated stream is, we will define Stream Elements. Stream Elements can be seen as building blocks of a stream, and each of them carries some information. We will first define a Stream Element from a structural point of view and then we will see what kind of stream elements exist in our model and what kind of information they carry along the stream.

Definition (Stream Element) A stream element, conventionally represented as \(x_i\), consists of a tuple part \(x\) and a sequence number \(i\), where \(i \in \mathbb{N}\). The sequence number maintains the total order of the stream elements in the stream.

Based on the type of information a stream element carries, there are two types of stream elements, which we are interested in: Data Elements and Control Elements. In order to illustrate these concepts see Figure 2.1.

Definition (Data Element) A Data Element is a Stream Element, \(x_i \in S\), carrying normal data which will be processed by the query, and contributes to the output or filtered out.
Definition (Control Element) A control element is a Stream Element, $x_i \in I$, carrying control information which causes the query to change its state. These elements do not contribute to the result, but they can indicate whether the result should be produced and even for how long it should be produced. Control Elements are events occurring in a query’s lifecycle. Operators composing the query Q, are aware of these control elements, and thus will take proper actions when encountering them. Moreover, control elements can only be seen in the Input Stream I, and they can never appear as a result of processing a query in the output stream. If a control element happens to be in the result, then the system is malfunctioning.

2.1.4 Sequence Number Generator

Each stream element needs to obtain a sequence number, which is strictly increasing to guarantee total order and uniqueness throughout the whole stream. Thus, there is a sequence number generator function, $f(x)$, which satisfies the strictly increasing property we are looking for. The sequence number is explicitly assigned in our model to enable us to have neat conceptual definitions. Figure 2.2, shows how the Sequence Number Generator, applied to the input stream.

Definition (Sequence Number Generator) A Sequence Number Generator assigns a sequence number to each input element. All input elements, regardless of their type, whether they are data element or control element will have a unique sequence number in an ascending order.
2.2 Mapping Functions

Now that we have defined the basics of our stream model, we will define some mediatory mapping functions, which help us to link the basic concepts to our model. These mapping functions are described under two separate sections. Basic Mapping Functions, are simple functions, having single input and single output. Stream Mapping Functions work on sequence of input and output elements. Both type of functions are introduced and illustrated in the coming sections.

2.2.1 Basic Mapping Functions

A stream element has some properties. In order to extract these properties, we define Basic Mapping Functions. One of these properties is that a stream element has a sequence number, in order to extract a sequence number from a stream element we define the $sequenceAt(x_i)$. Figure 2.3 shows how these basic functions work.

Definition ($sequenceAt(x_i)$) The $sequenceAt(x_i)$ function takes a stream element, $x_i$, as an input parameter and returns its sequence number, $i$. This function is injective, meaning that the sequence number of each element is unique in the whole stream.

$$sequenceAt(x_i) : S \rightarrow \mathbb{N}$$

$$sequenceAt(x_i) = i$$

Inversely, we may want to see what element is at a given position, then we introduce the $elementAt(i)$. 

![Figure 2.2: Sequence Number Generator assigns to each input element a sequence number.](image-url)
Basic Query Lifecycle Model - Mapping Functions

Figure 2.3: sequenceAt(x_i) returns the sequence number of a given element and elementAt(i) returns the element corresponding to a sequence number.

Definition (elementAt(i)) The elementAt(i) function takes an index number as input parameter and returns the corresponding element, x_i, at this index position. The elementAt(i) is the inverse of sequenceAt(x_i).

\text{elementAt}(i) : \mathbb{N} \mapsto S
\text{elementAt}(i) = x_i
\text{elementAt}(i) = \text{sequenceAt}(x_i)^{-1}

2.2.2 Stream Mapping Functions

This section is crucial to deeply understand in order to proceed. The following definitions will be repeated throughout the thesis very often, and are of fundamental importance. These functions map input elements to output elements and vice versa. They will be used in the definition of control elements and stream events. We will introduce them by detail now.

conventionally from now on, wherever x_i is used, an input stream element is meant, and wherever y_j is used, an output stream element is meant.

Definition (depends(y_j)) The depends(y_j) function takes an output element, y_j, as an input parameter and returns a sequence of input elements that y_j depends on. In other words, depends(y_j) maps a single output element to a sequence of contributing input elements. The size of the returned sequence is a finite natural number.

\text{depends}(y_j) : O \mapsto I
\text{Size of depends}(y_j) : |\text{depends}(y_j)| = a, where a \in \mathbb{N} and finite

Note that, control elements cannot be a part of depends(y_j). depends(y_j) can be seen as the provenance of a single output element, and control elements
are not considered to play any role in the provenance of an output element. In figure 2.4 you will see an illustration of $\text{depends}(y_j)$. $\text{depends}(y_j)$ maps one and only one output element into one or more contributing elements in a sequence of input elements, and control elements are not considered as a member of $\text{depends}(y_j)$.

In the following we will define a mirrored function of $\text{depends}(y_j)$, named $\text{contributes}(x_i)$. In figure 2.5 you will see an illustration of $\text{contributes}(x_i)$.

**Definition** ($\text{contributes}(x_i)$) The $\text{contributes}(x_i)$ function takes an input element, $x_i$, as an input parameter and returns a sequence of output elements that $x_i$ contributes to. In other words, $\text{contributes}(x_i)$ maps a single input element to a sequence of depending output elements. The size of the returned sequence is a finite natural number.

$$\text{contributes}(x_i) : I \rightarrow O$$

$$\text{Size of } \text{contributes}(x_i) : |\text{contributes}(x_i)| = b, \text{ where } b \in \mathbb{N} \text{ and finite}$$

$\text{contributes}(x_i)$ maps one and only one input element into one or more dependent elements in a sequence of output elements. In figure 2.5 this function has been illustrated.

A critical question here is, where does this $\text{depends}(y_j)$ and $\text{contributes}(x_i)$ come from? In section 2.1.1.1, we defined a query, as having an Identity and an Expression. The Expression of the query describes the logic of the query. Expression of the query tells how an output is produced, and what are the dependencies. Therefore, $\text{depends}(y_j)$ and $\text{contributes}(x_i)$ functions are being built on the expression of the query. Meaning that, Expression of the query results in its $\text{depends}(y_j)$ and $\text{contributes}(x_i)$, and thus every query, having different Expression has also
2.3 Basic Control Elements

The concept of change will be implemented through basic control elements. How to compose these control elements in order to implement change will be answered in next chapter, but first we need to thoroughly understand these basic control element. Implementing change by smaller building blocks, called basic control elements, similar to any other component-based development, gives us flexibility, reuse and makes it easier to understand. The basic control elements are defined using the Stream Mapping functions 2.2.2. More important than knowing how to change, in knowing how to start or stop a query. Start and Stop are considered as control elements. Each of which has two different implementations, that will be described in the following chapters.

2.3.1 Start

Start generally means that input is contributing to the process of output generation. After receiving a start control element, the query \( Q \) will begin producing output. In our model, there are two kinds of starts: Immediate Start and Smooth Start. Further on, we will define these two kinds of control elements, which realize start in two different ways. Note that Immediate Start control element will be shortened to \( \text{istart} \) and Smooth Start control element will be shortened to \( \text{sstart} \), overall in this writing.

**Definition (Immediate Start)** \( \text{IStart}(Q) \) where \( Q \) is the identity of the query to be immediately started, is one of the basic control elements depicted as \( x_{\text{istart}} \).
where \( istart \) is the sequence number of the Immediate Start control element in the input stream. After receiving the Immediate Start control element, \( x_{istart} \), by the query, data elements in the input having a sequence number bigger than \( istart \) will only contribute to the output. In other words, after receiving the Immediate Start control element, \( x_{istart} \), by the query, the outputs generated are only dependent on input elements appearing after Immediate Start control element. Thus, the following statement holds:

\[
O = \{ y_j | \forall x_i \in \text{depends}(y_j), i > istart \}
\]

**Definition (Smooth Start)** \( SStart(Q) \) where \( Q \) is the identity of the query to be smoothly started, is one of the basic control elements depicted as \( x_{sstart} \), where \( sstart \) is the sequence number of the start control element. After receiving the Smooth Start control element, \( x_{sstart} \), by the query, data elements in the input having a sequence number bigger than \( sstart \) will contribute to the output. In other words, after receiving the Smooth Start control element, \( x_{sstart} \), by the query, the outputs generated are dependent on input elements appearing after Smooth Start control element. Thus, the following statement holds:

\[
O = \{ y_j | \exists x_i \in \text{depends}(y_j), i > istart \}
\]

These two definitions might look similar, but they have an essential difference. Immediate Start, only lets input after it contribute to output. Nevertheless, Smooth Start, lets also outputs having dependencies on the elements before smooth start to contribute to output, in condition that they also have dependencies to elements after smooth start. Figure 2.6, illustrates clearly the difference between these two types of starts.

**Corollary 2.3.1.** Assume we apply an Immediate Start on the input once and the other time we apply a Smooth Start. The output produced by applying the Immediate Start is the subset of the output produced by the Smooth Start.

\[
O_{istart} = \{ y_j | \forall x_i \in \text{depends}(y_j), i > istart \}
\]

\[
O_{sstart} = \{ y_j | \exists x_i \in \text{depends}(y_j), i > sstart \}
\]

\[
O_{istart} \subseteq O_{sstart}
\]
Smooth Start control element is statically not computable and practically hard to implement, but it gives our basic model a symmetry which we will more describe later in this chapter and helps constructing new change methods as part of the modification model.

2.3.2 Stop

Symmetric to start, there are similar definitions for stop elements. *Stop* generally means that input will eventually stop contributing to the process of output generation. After receiving a stop control element, the query will eventually stop producing output. In our model, there are two kinds of stops: *Immediate Stop* and *Drain Stop*. Further on, we will define these two kinds of control elements, which implement stop in two different ways.

**Definition (Immediate Stop)** $\text{IStop}(Q)$ where $Q$ is the identity of the query to be stopped, is one of the basic control elements depicted as $x_{istop}$, where $istop$ is the sequence number of the *Immediate Stop* control element. After receiving the *Immediate Stop* control element, $x_{istop}$, by the query, data elements in the input having a sequence number bigger than $istop$ will no more contribute to the output. In other words, after receiving the *Immediate Stop* control element, $x_{istop}$, by the query, the outputs generated are dependent *only* on input elements appearing before *Immediate Stop* control element. Thus, the following statement holds:

$$O = \{ y_j | \forall x_i \in \text{depends}(y_j), i < istop \}$$

**Corollary 2.3.2.** There is no data element in output stream, having dependencies on inputs arriving after immediate stop control element $x_{istop}$.

$$\nexists y_j \in O, \text{where} \exists x_i \in \text{depends}(y_j) \land i > istop.$$  

**Definition (Drain Stop)** $\text{DStop}(Q)$ where $Q$ is the identity of the query to be gradually stopped, is one of the basic control elements depicted as $x_{dstop}$, where $dstop$ is the sequence number of the *Drain Stop* control element. After receiving the *Drain Stop* control element, $x_{dstop}$, by the query, only outputs will be produced that have dependency on input elements before $dstop$. Thus, the following statement holds:

$$O = \{ y_j | \exists x_i \in \text{depends}(y_j), i < dstop \}$$

**Corollary 2.3.3.** There is no output element in output, not having dependencies on inputs before drain stop control element $x_{dstop}$.

$$\nexists y_j \in O, \text{where} \nexists x_i \in \text{depends}(y_j) \land i < dstop.$$  

These two definitions might also look similar, but they have again an essential difference. *Immediate Stop*, only lets input before it contribute to output. In contrary, *Drain Stop*, lets also outputs having dependencies on the elements after *Drain Stop* contribute to output, in condition that they also have dependencies to elements before *Drain Stop*. Figure 2.7, illustrates clearly the difference between these two types of stops.
Corollary 2.3.4. Assume we apply an Immediate Stop on an already started input once and, then we apply a Drain Stop at the same position. Thus, the output, produced by applying the Immediate Stop is the subset of the output produced by the Drain Stop.

\[
O_{istop} = \{ y_j \mid \forall x_i \in \text{depends}(y_j), i < istop \} \\
O_{dstop} = \{ y_j \mid \exists x_i \in \text{depends}(y_j), i < dstop \} \\
O_{istop} \subseteq O_{dstop}
\]

2.3.3 Interactions of Basic Control Elements

The basic control elements are events triggering certain action in the behavior of query, or in other words changing its state. Thus their interactions can be seen as finite State Machine. Before we introduce the Query State Machine, we should define the query Lifecycle more concretely.

Definition (Query Lifecycle) Query Lifecycle is a series of states connected by predefined transitions. Each transition refers to a control element.

2.3.3.1 Query State Transition

A single query may have different states in its lifecycle. Our model of behavior is composed of finite number of states, and transitions between those states, where we can inspect the way in which the logic runs when certain control elements are encountered. Further on we will define all possible states clearly.

Definition (Stopped State) The Stopped State is a state where query consumes no more input to produce output. Every query is initially in the Stopped State. In other words, Stopped State is the start state of the query state transition.

Definition (Running State) The Running State is the state where query is consuming input to produce output.
Definition (Smoothing State) The Smoothing State is the state where the query might produce a limited output, having dependencies on elements before start, and then move to the Running State with an 𝜖-transition.

Definition (Draining State) The Draining State is the state where the query might produce a limited amount of output, having dependencies on elements after stop, and then move to the Stopped State with an 𝜖-transition.

Definition (Query State Transition Machine) We define the Query State Transition Machine, as an undeterministic finite state machine, having a quintuple \((\Sigma, S, s_0, \delta, F)\), where:

- \(\Sigma\) is the input elements. \(\Sigma = \{\text{istop}, \text{dstop}, \text{istart}, \text{sstart}, \text{data}\}\) where:
  - \(\text{istop}\) is short for Immediate Stop control element.
  - \(\text{dstop}\) is short for Drain Stop control element.
  - \(\text{istart}\) is short for Immediate Start control element.
  - \(\text{sstart}\) is short for Smooth Start control element.
  - \(\text{data}\) is generally for all elements except control elements.

- \(S\) is the finite set of states.
  \[ S = \{\text{Running}, \text{Stopped}, \text{Draining}, \text{Smoothing}\} \]

- \(s_0\) is the Start State, which is already described that every query is primarily in the Stopped State, \(q_{\text{stop}}\).

- \(\delta\) is the state-transition function:
  \[ \delta : S \times \Sigma \rightarrow S \]

- \(F\) is the set of final states. \(F\) is an empty set, because of the infinite nature of the stream.

  \[ F = \emptyset \]

In table 2.1, we see how a certain action, invokes certain state transitions. From table 2.1, we conclude that Data cannot change the state a query. However, implicitly it can change the query’s state from Draining to Stopped and from Smoothing to Running. The implicit changes between these states are demonstrated as 𝜖-transitions, emphasizing the statically not computable behavior of our model, when encountering a drain stop or smooth start. Fields having * mean that if a certain condition has been fulfilled they will change their state, otherwise they will stay in their current state. We see here that Stopped State and Running State are exactly the inverse of each other and also that Smoothing State and Draining State are exactly the inverse of each other.

In figure 2.8 you will see the Query State Transition. Moreover, in figure 2.8, the axis of symmetry shows that when you fold this image, each state will be covered up by its inverse state.
Table 2.1: Query State Transition Table.

<table>
<thead>
<tr>
<th>Current State → Event</th>
<th>Running</th>
<th>Smoothing</th>
<th>Draining</th>
<th>Stopped</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate Start</td>
<td>Running</td>
<td>Running</td>
<td>Running</td>
<td>Running</td>
</tr>
<tr>
<td>Smooth Start</td>
<td>Running</td>
<td>Smoothing</td>
<td>Smoothing</td>
<td>Smoothing</td>
</tr>
<tr>
<td>Drain Stop</td>
<td>Draining</td>
<td>Draining</td>
<td>Draining</td>
<td>Stopped</td>
</tr>
<tr>
<td>Immediate Stop</td>
<td>Stopped</td>
<td>Stopped</td>
<td>Stopped</td>
<td>Stopped</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Running</td>
<td>Running$^*$</td>
<td>Stopped$^*$</td>
<td>Stopped</td>
</tr>
<tr>
<td>Data</td>
<td>Running</td>
<td>Smoothing</td>
<td>Draining</td>
<td>Stopped</td>
</tr>
</tbody>
</table>

Figure 2.8: Query State Transitions
Figure 2.9: By Reversing the Input and using the inverse of each control element, reverse of output stream can be created.

Table 2.2: Summary of Basic Control Elements

<table>
<thead>
<tr>
<th>Basic Control Element</th>
<th>Abbreviation</th>
<th>Definition</th>
<th>Inverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate Start</td>
<td>istart</td>
<td>( O = { y_j</td>
<td>\forall x_i \in \text{depends}(y_j), i &gt; istart } )</td>
</tr>
<tr>
<td>Smooth Start</td>
<td>sstart</td>
<td>( O = { y_j</td>
<td>\exists x_i \in \text{depends}(y_j), i &gt; sstart } )</td>
</tr>
<tr>
<td>Drain Stop</td>
<td>dstop</td>
<td>( O = { y_j</td>
<td>\exists x_i \in \text{depends}(y_j), i &lt; dstop } )</td>
</tr>
<tr>
<td>Immediate Stop</td>
<td>istop</td>
<td>( O = { y_j</td>
<td>\forall x_i \in \text{depends}(y_j), i &lt; istop } )</td>
</tr>
</tbody>
</table>

2.3.3.2 Inverse of Basic Control Elements

One of the most important aspects of having a symmetric model is the ability to feed the stream backward using the inverse control element of each control element in the stream and producing the same output backward again. This aspect is clearly shown in figure 2.9.

This symmetry of our model, not only gives it a conceptual strength but more significantly it enables redo and undo of events; thus, stepping toward transactional behavior on the streams.

2.3.3.3 Short Specification of Basic Control Elements

We have defined four Basic control elements. In order to proceed to next chapter, where we will combine them and get new Complex control elements, we provided a summary in table 2.2 for these basic elements, and in figure 2.10, there will be a complete demonstration of how these basic control elements function.
Figure 2.10: The big image of Basic control elements and their functionality.
Chapter 3

Query Modification Model

So far it was all about fundamental aspects of the model. We have learned the Basic Control Elements, and how they interact as events in a query’s lifecycle. It was all about one query changing its states upon receiving a Basic Control Element. As already discussed in the previous chapter, we define Basic Control Elements to be the building blocks of Change Elements. In our model, we have used a punctuated stream having more than just data in it. As a matter of fact we wanted to extend this model, to be able to also modify queries. In this chapter we will first see what a Change means, based on the fundamental definitions of chapter 2. Then, we will take a look at the correctness issues that should be taken care of. In the end, we will suggest some of the possible query modification models, and discuss their correctness criteria.

3.1 Query Modification

In order to introduce different modification models we should first define what a modification means in our system.

As already mentioned in chapter 2, section 2.1.1.1, each query in the system has an identity and an expression.

\[ Q = Id \times Expression \]

However, this was the story before Query Modification. When we modify a query, the identity of the query should not change; otherwise the query could lose its owner. In order to make it more understandable, in a normalized database relation, each tuple has a primary key, and any modification or update happening to the tuple, the primary key should not change. Back to our model, when we change the query its Identity remains the same and its Expression changes. Still, there is something missing. If we change a query just by changing its Expression, the old version of the query will be lost forever. Here comes exactly a new concept into play, the query Version. By using an additional attribute to keep track of the Version of the query, we can not only easily switch between old and new version, but also switch between all versions of the modified query.
By versioning we could also form up a history for the query’s lifecycle, and make additional analysis on it. As discussed here, we introduce a slight change in our query model to proceed with our Modification Model. A query consists of:

\[ Q = Id \times Version \times Expression \]

Additionally, according to section 2.2.2, each Query Expression implies its own \( contributes(x_i) \) and \( depends(y_j) \). Knowing this preliminaries, we can neatly define Query Modification.

**Definition (Query Modification)** When the query Q having the following characteristics:

\[ Q = (id, version, expression) \]

has been modified, then we will have the same query Q, with the following characteristics:

\[ Q = (id, version', expression') \]

In other words, modifying a query, means changing its \( Expression \), and thus changing its \( Version \), without touching its \( Identity \).

Changing a query’s Expression, changes its \( depends(y_j) \) and \( contributes(x_i) \). We can claim that for each version of the query Q, we will have a different \( depends(y_j) \) and \( contributes(x_i) \). Since, in this thesis, in case of change, we only deal with at most two versions of a query concurrently, conventionally we refer to them as the old version of the query Q and the new version of the query Q. There are further conventions in representing their \( depends(y_j) \) and \( contributes(x_i) \).

The old version of the query Q, is the version we want to modify. Its \( depends(y_j) \) and \( contributes(x_i) \) functions are represented as: \( depends_o(y_j) \) and \( contributes_o(x_i) \), where the additional subscript, \( o \), stands for \( old \) version; and, the new version of the query Q, is the modified version. Its \( depends(y_j) \) and \( contributes(x_i) \) functions are represented as: \( depends_n(y_j) \) and \( contributes_n(x_i) \), where the additional subscript, \( n \), stands for \( new \) version.

### 3.2 Correctness Criteria

Now that we know what the exact definition of modifying a query is, we will see what kind of correctness criteria are required to make a resilient Modification Model. The correctness criteria are being categorized into Safety and Liveness criteria.

### 3.3 Safety Criteria

According to Lamport [9] a safety property expresses that \textit{something bad will not happen} during the execution. Based on this we have specified, what could be \textit{bad} when a change happens. In the following subsections of safety criteria,
we will define what bad is, and illustrate it with examples.

As already defined in the query state transitions (see 2.3.3.1), we have described that after successfully deploying a query, it will be in a stopped state, until it receives its first start element and then it begins consuming input and producing output. As a general assumption for the whole safety criteria we consider that a query is in the running state.

3.3.1 Loss

Upon changing a query while it is running, there might be some chance of losing the output data. In this section we will define loss, and different policies to identify its occurrences on a stream. In order to simplify the definition of Loss, we assume that query Q has not been modified, but only its state has been changed. Afterwards, we will define loss in case of Query Modification. There are some preliminaries to be defined, before proceeding with the definition of Loss. These concepts are later used to define Loss.

Definition (Observed Output Stream) An Observed Output Stream is an output stream, being generated by the Q, receiving control elements in an arbitrary order and amount. The Observed Output Stream can be considered as a set of output elements, because order does not play any role in the definition of loss (Order will be discussed in depth in section 3.3.2 later in this chapter). Conventionally, the Observed Output Stream will be represented throughout this thesis as: $O_{obs}$

Definition (Reference Output Stream) A Reference Output Stream is an output stream, being generated by the Q, without receiving any control element. In other words, the Reference Stream is the perfect world where no event has been practiced on the stream. The Reference Output Stream can be considered as a set of output elements, because order does not play any role in the definition of loss (Order will be discussed in depth in section 3.3.2 later in this chapter). Conventionally, the Observed Output Stream will be represented throughout this thesis as: $O_{ref}$

Definition (Loss) As $O_{ref}$ and $O_{obs}$ are both considered as sets, rather than sequences, Loss is the set-difference between the Reference Stream and the Observed Stream.

$$Loss = O_{ref} - O_{obs}$$

In figure 3.1 we illustrate a Loss, assuming we have an input Stream with an immediate stop followed by an immediate start, and the query simply returns values bigger than 1. As illustrated in Figure 3.1, if we consider each output element as a $(data,\text{depends}(y_j))$ pair, then the Output of the observed Stream will be $O_{obs} = \{(2,2),(3,9)\}$, whereas in the following picture where the reference stream is demonstrated, the output will be: $O_{ref} = \{(2,2),(3,6),(3,9)\}$. It is demonstrated here, that the reference stream does not behave corresponding
to the control elements it receives, but it simply ignores them to produce the most complete result. To construct \( O_{obs} \) and \( O_{ref} \) sets, we may also not need pair elements, but we can only include the provenance of each output element. Hence, in the above example, we will have \( O_{obs} = \{2,9\} \) and \( O_{ref} = \{2,6,9\} \), so Loss will be \( \text{Loss} = O_{ref} - O_{obs} = \{6\} \), which means that an output element is missing from the observed stream; hence forming a Loss.

After defining Loss for basic control elements, it is time to have Loss defined in case of a Query Modification. When a query is being modified, a single Reference Stream cannot help anymore, because we have two queries, the old query and the new query. We can define two reference streams for each query, which will run from the beginning, but the problem is we may have tens of queries deployed and we do not know beforehand to which version of the query we will switch, so that we can build its reference stream. Therefore, we define the following stream which minimizes Loss and will serve as our reference stream.

Definition (Cross-Reference Output Stream) A Cross-Reference Output Stream serves for the same goal, as for the reference stream but this time the complex control elements are allowed. In case of change we will build a cross-reference stream with the following characteristics. After receiving the change element \( C(o,n) \), where \( o \) is the old version of the query and \( n \) is the new version of the new query, old query has already a reference stream running for it, so we inject a drain stop to stop the old version of query from running, without discarding any state, to guarantee a lossless reference stream, and simultaneously inject an immediate start control element for the new query. A Cross-Reference Stream is defined as follows, assuming that \( c \) is the index of the change element:

\[
\begin{align*}
O_{ref,old} &= \{ y_j \mid \exists x_i \in \text{depends}_o(y_j) \text{ where } i < c \} \\
O_{ref,new} &= \{ y(j') \mid \forall x_i \in \text{depends}_n(y(j')) \text{ where } i > c \} \\
O_{ref} &= O_{ref,old} \cup O_{ref,new}
\end{align*}
\]

Or in other words, the reference stream of the old query, \( O_{ref,old} \), is the definition of a Drain Stop 2.3.2; and the reference stream of the new query, \( O_{ref,new} \), is the definition of an Immediate Start 2.3.1.
Figure 3.2: The Observed Stream, when an immediate change element comes along and the Reference Streams

After, properly defining the Cross-Reference Output Stream, we will proceed with the definition of Loss in case of change.

**Definition (Loss in presence of Change)** Loss in presence of Change elements is quite similar to the basic loss definition, only instead of having the Reference Output Stream, the Cross-Reference Output stream is used and the following holds:

\[
\text{Loss} = O_{\text{cref}} - O_{\text{obs}}
\]

In order to illustrate Loss in presence of change definition, consider we have a query \( Q \), which sums up each 3 consequent values with a sliding of 2. After a while an immediate change element 3.5.1 appears, thus the observed stream and our reference streams will look like figure 3.2.

As illustrated in Figure 3.2, if we consider each output element as a \((y_j, \text{depends}(y_j))\) pair, then the Output of the observed stream will be \( O_{\text{obs}} = \{(4,3-1),(7,8-5)\} \), whereas according to the Cross-Reference Stream ,the output will be: \( O_{\text{cref}} = O_{\text{ref:old}} \cup O_{\text{ref:new}} = \{(4,3-1),(4,6-3),(7,8-5)\} \).

According to this example the Loss will be \( \text{Loss} = O_{\text{cref}} - O_{\text{obs}} = \{(4,6-3)\} \), which means a window has been expected from the third to sixth element to participate in an output element generation, but it did not, and thus a loss has occurred.

### 3.3.2 Disorder

Disorder occurs when the order of the output elements violate order rules. There are two kind of order rules defined: Query-Level Disorder and Window-Level Disorder.
**Definition (Query-Level Disorder)** In case of having a change control element, there might be conditions where the output of the old version of the query appears after the output of the new version of the query. Here, we say there has been a Query-Level Disorder occurred. In other words, if there is an output element belonging to the old version of the query, but having dependencies on input elements arriving after change, and if there is an output element belonging to the new version of the query, then the output of new version of the query has appeared before output of the old version of the query on the output sequence. This definition is formalized as following:

\[
(\exists y_j \in O \land y_j \in \text{contributes}_o(x_i), \text{ where } i > c) \\
\land (\exists y_{j'} \in O \land y_{j'} \in \text{contributes}_n(x_{i'}), \text{ where } i' > c) \\
\land j > j'
\]

**Definition (Window-Level Disorder)** Window-Level Disorder is more complex than the former one. The general rule of having order between windows is that the window which finishes first should deliver its output first. If two window close on a single item, then the output of the window that has started first should appear in the output first, and if both windows have the same start and end element then they are considered duplicates. For more information on duplicates see 3.3.3. Window-Level Disorder can be formalized as following:

\[
\text{MAX}(\text{sequenceAt}(\text{depends}_o(y_j))) > \text{MAX}(\text{sequenceAt}(\text{depends}_n(y_{j'}))) \\
\land j' > j
\]

Generally speaking, Query-Level Disorder can be considered as a coarse-grained disorder compared to Window-Level Disorder which can be considered as a fine-grained disorder.

### 3.3.3 Duplicates

In case of change, there might be that the same exact input contributes to two different outputs. These outputs are considered duplicates because they have the exact same \(\text{depends}(y_j)\). The transition logic of the query applied to the each output’s \(\text{depends}(y_j)\) is not considered in the definition of duplicates, meaning that even two output elements having completely different data but same \(\text{depends}(y_j)\) members are considered duplicates.

**Definition (Duplicates)** According to the description above, an output having the same input as another output is considered a duplicate.

\[
y_{j'} \text{ is a duplicate of } y_j \iff \text{depends}_n(y_{j'}) = \text{depends}_o(y_j)
\]

### 3.4 Liveness Criteria

According to Lamport in [9], a liveness property expresses that eventually *something good must happen* during an execution. Normally progress properties are under-categorized as Liveness criteria.
3.4.1 Progress of the new version of the Query

Definition (Progress of the new version of the Query) In case of a change, the system should guarantee that the new query will progress at some point. Since it may be possible that the old version of the query refuses to quit, the system has to explicitly enforce the progress of the new version of the query.

3.4.2 Termination of the old version of the Query

Definition (Termination of the old version of the Query) In case of change, the system should guarantee that the old version of the query will terminate at some point and free up its occupied resources. Since it might be possible that the old version never reaches the condition of its termination and continues to use up resources, thus there must be a termination policy explicitly enforced to guarantee the old version’s termination and hence the new version’s progress.

We should take note that the termination of the old version does not implicitly guarantee the progress of the new version and vice versa.

3.5 Complex Control Elements

Now that we know what can go wrong, and what kind of correctness criteria we have, we can begin moving the basic control elements from chapter 2 around and constructing complex control elements to implement a change policy which meets our requirements. In this section we will introduce some of these query modification models, for which we have found plausible usecases. These change elements are constructed by combining basic elements and shifting their point of action on the stream. A change element, brings about a new version of a query and therefore a new expression of the query, as already described in section 3.1; hence, the \( \text{depends}(y_j) \) and \( \text{contributes}(x_i) \) mapping functions of a query will change.

3.5.1 Immediate Change

In some applications the user needs to be able to instantly change the query. Immediate Change, makes it possible to instantly generate new results from the new version and the user will not receive any output corresponding to the old one. In immediate change each input element exclusively contributes to the output element of the old or the new version.

Definition (Exclusive Contribution) In case of change, if an input element, \( x_i \), solely contributes to \( \text{depends}_n(y_{j'}) \) or \( \text{depends}_o(y_j) \) then it is said to have an exclusive contribution.

\[
\forall x_i \in I, x_i \in (\text{depends}_n(y_{j'}) \oplus \text{depends}_o(y_j))
\]
Figure 3.3: The Immediate Change Control Element splits the Output to two completely non-overlapping Sets.

Note that, ⊕ is an EXCLUSIVE-OR operator.

**Definition (Immediate Change)** $\text{IChange}(Q, V)$, where $Q$ is the Id of the query to be modified and $V$ is the desired version of $Q$ to be switched to, is one of the complex control elements. $\text{IChange}(Q, V)$ element, $x_{\text{ichange}}$, is basically an Immediate Stop element, $x_{\text{istop}}$, followed immediately by an immediate start element, $x_{\text{istart}}$. Thus, following statement holds:

$$O = O_{\text{istop}} \cup O_{\text{istart}}$$

$$O = \{y_j \mid \forall x_i \in \text{depends}_o(y_j), i < \text{ichange}(\text{istop}) \}$$

$$\cup \{y_j' \mid \forall x_i \in \text{depends}_n(y_j'), i > \text{ichange}(\text{istart}) \}$$

**Corollary 3.5.1. Disjointness** Output elements are exclusively dependent on input elements before $x_{\text{ic}}$ or elements after it. Thus, following statement holds:

$$\exists y_j, y_j' \in O,$$ \hspace{1em} where \hspace{1em} $\text{depends}_n(y_j') \cap \text{depends}_o(y_j) \neq \emptyset$

**Corollary 3.5.2. Exclusive Contribution** Input elements arriving before $x_{\text{ichange}}$ only contribute to $\text{depends}_o(y_j)$ and input elements arriving after $x_{\text{ic}}$ only contribute to $\text{depends}_n(y_j')$.

$$(\exists x_i \text{ where } i > \text{ichange} \land x_i \in \text{depends}_o(y_j))$$

$$\land (\exists x_i \text{ where } i < \text{ichange} \land x_i \in \text{depends}_n(y_j'))$$

In figure 3.3, we will see how the Immediate Change Control Element splits the output to two disjoint sets.

In terms of safety correctness guarantees, Immediate Change behaves as follows:

Here, we provide proofs according to statements made in table 3.1.

**Proposition 3.5.3.** Immediate Change cannot avoid Loss.
Proof. A Lossless Stream is already defined in 3.3.1 as a cross-reference stream, as follows:

\[ O_{\text{cref}} = O_{\text{dstop}} \cup O_{\text{istart}} \]
\[ O = \{ y_j \mid \exists x_i \in \text{depends}_o(y_j), i < \text{dstop} \} \]
\[ \quad \cup \{ y_{j'} \mid \forall x_i \in \text{depends}_n(y_{j'}), i > \text{istart} \} \]

and the definition of Immediate Change is as follows:

\[ O_{\text{ichange}} = O_{\text{istop}} \cup O_{\text{istart}} \]
\[ O = \{ y_j \mid \forall x_i \in \text{depends}_o(y_j), i < \text{ichange}(\text{istop}) \} \]
\[ \quad \cup \{ y_{j'} \mid \forall x_i \in \text{depends}_n(y_{j'}), i > \text{ichange}(\text{istart}) \} \]

Thus we will have:

\[ O_{\text{istop}} \subseteq O_{\text{dstop}} \land O_{\text{istart}} = O_{\text{istart}} \Rightarrow O_{\text{ichange}} \subseteq O_{\text{cref}} \]

Proposition 3.5.4. Immediate Change avoids Query-Level Disorders.

Proof. The definition of query-level disorder is:

\[ (\exists y_j \in O \land y_j \in \text{contributes}_o(x_i), \text{where } i > c) \]
\[ \land (\exists y_{j'} \in O \land y_{j'} \in \text{contributes}_n(x_{j'})) \land j > j' \]

According to definition of Immediate Change:

\[ \not\exists y_j \in O \land y_j \in \text{contributes}_o(x_i), \text{where } i > \text{ichange} \]

Thus, in an immediate change there is no input element after ichange which still uses the \( \text{contributes}_n(y_{j'}) \). This implies that no output of the old version of the query is seen after immediate change has been applied.

Proposition 3.5.5. Immediate Change avoids Window-Level Disorder.

Proof. Window-Level Disorder happens when:

\[ \text{MAX} (\text{sequenceAt}(\text{depends}_o(y_j))) > \text{MAX} (\text{sequenceAt}(\text{depends}_n(y_{j'}))) \]
\[ \land j' > j \]

In case of an Immediate Change this could never occur, because:

\[ \text{MAX} (\text{sequenceAt}(\text{depends}_o(y_j))) < \text{ichange} \]
\[ \land \not\exists y_j' \in O, \text{where } \text{MIN} (\text{sequenceAt}(\text{depends}_n(y_{j'}))) < \text{ichange} \]

There is no output element whose minimum sequence number of elements in \( \text{depends}_n(y_{j'}) \) is smaller than maximum sequence number of elements in \( \text{depends}_o(y_j) \).
Table 3.2: Liveness guarantees offered by Immediate Change

<table>
<thead>
<tr>
<th>Change Policy</th>
<th>Liveness Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate Change</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Proposition 3.5.6.** Immediate Change guarantees no duplicates.

*Proof.* Referring to the Disjointness property of Immediate Change in 3.5.1, this proposition is already proved. □

Moreover, in terms of liveness correctness guarantees, Immediate Change behaves as table 3.2:

**Proposition 3.5.7.** Immediate Change guarantees the progress of the new version of the query.

*Proof.* According to immediate change the old version will be stopped, and the new version will start running, putting the new version of the query in a running state, and this is what we desired. □

**Proposition 3.5.8.** Immediate Change guarantees the termination of the old version of the query.

*Proof.* According to immediate change the old version will be stopped, and the new version will start running, putting the old version of the query in a stopped state, and this is what we desired. □

### 3.5.2 Drain Change

In Drain Change, in contrast to Immediate Change 3.5.1, after Drain Change query Q will consume input until there is no output that have dependencies on data elements before drain change control element to produce output of the old query. Drain Change is defined as follows:

**Definition (Drain Change)** $D\text{Change}(Q,V)$, where $Q$ is the continuous query to be changed and $V$ is the desired version of $Q$ to be switched to, is one of the complex control elements. $D\text{Change}(Q,V)$ element, $x_{d\text{change}}$, is basically a Drain Stop element, $x_{d\text{stop}}$, followed immediately by an Immediate Start element, $x_{i\text{start}}$. Thus, following statement holds:

\[
O = O_{d\text{stop}} \cup O_{i\text{start}}
\]
\[
O = \{ y_j \mid \exists x_i \in \text{depends}_o(y_j), \ i < d\text{change}(d\text{stop}) \} \\
\cup \{ y_{j'} \mid \forall x_i \in \text{depends}_n(y_{j'}), \ i > d\text{change}(i\text{start}) \}
\]

The figure 3.4 shows the definition 3.5.2 of drain change.

In terms of safety correctness guarantees, Drain Change behaves as follows:

Here, we provide proofs according to statements made in table 3.3.
Figure 3.4: The Drain Change Control Element implementations according to definition 3.5.2

Table 3.3: Safety guarantees offered by Drain Change

<table>
<thead>
<tr>
<th>Safety Criteria</th>
<th>Change Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Loss</td>
<td>✓</td>
</tr>
<tr>
<td>No Query-Level Disorder</td>
<td>✗</td>
</tr>
<tr>
<td>No Window-Level Disorder</td>
<td>✓</td>
</tr>
<tr>
<td>No Duplicates</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Proposition 3.5.9.** Drain Change avoids Loss.

*Proof.* A Lossless Stream is already defined in 3.3.1 as a cross-reference stream, as follows:

\[
O_{\text{cref}} = O_{\text{stop}} \cup O_{\text{start}}
\]

\[
O = \{ y_j \mid \exists x_i \in \text{depends}_o(y_j), \ i < dstop \}
\]

\[
\cup \{ y_{j'} \mid \forall x_i \in \text{depends}_n(y_{j'}), \ i > istart \}
\]

And the definition of Drain Change is as follows:

\[
O_{\text{dchange}} = O_{\text{stop}} \cup O_{\text{start}}
\]

\[
O = \{ y_j \mid \exists x_i \in \text{depends}_o(y_j), \ i < \text{dchange}(dstop) \}
\]

\[
\cup \{ y_{j'} \mid \forall x_i \in \text{depends}_n(y_{j'}), \ i > \text{dchange}(istart) \}
\]

Thus we will have:

\[
O_{\text{dstop}} = O_{\text{stop}} \land O_{\text{start}} = O_{\text{start}} \Rightarrow O_{\text{dchange}} = O_{\text{cref}}
\]

**Proposition 3.5.10.** Drain Change cannot avoid Query-Level Disorders.

*Proof.* The definition of query-level disorder is:

\[
(\exists y_j \in O \land y_j \in \text{contributes}_o(x_i), \ \text{where} \ i > c) \land (\exists y_{j'} \in O \land y_{j'} \in \text{contributes}_n(x_{i'}), \ \text{where} i' > c) \land j > j'
\]

We will use the proof by contradiction method. We assume, Drain Change preserves Query-Level disorder. In figure 3.4 we can see that it contradicts our
Table 3.4: Liveness guarantees offered by Drain Change

<table>
<thead>
<tr>
<th>Change Policy</th>
<th>Liveness Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drain Change</td>
<td>✓</td>
</tr>
</tbody>
</table>

assumption and thus the proposition 3.5.10 holds. In figure 3.4, we see that two green lines showing the \( \text{depends}(y_j) \) sequence of new version of query, close sooner than a depends sequence of the old version, and thus will be seen in the output before an old output element, meaning that an old output element will be seen after some new output element after change.

Proposition 3.5.11. **Drain Change avoids Window-Level Disorder.**

**Proof.** Window-Level Disorder happens when:

\[
\text{MAX} (\text{sequenceAt}(\text{depends}_o(y_j))) > \text{MAX} (\text{sequenceAt}(\text{depends}_n(y_{j'})))
\]

\( \land j' > j \)

In case of a Drain Change this could never occur, because we have strictly defined, that output should be produced, guaranteeing the window-level order.

Proposition 3.5.12. **Drain Change guarantees no duplicates.**

**Proof.** According to definition of drain change 3.5.2:

\[
O_{\text{change}} = O_{\text{dstop}} \cup O_{\text{istart}}
\]

\[
O = \{ y_j \mid \exists x_i \in \text{depends}_o(y_j), i < \text{dchange}(\text{dstop}) \}
\]

\[
\cup \{ y_{j'} \mid \forall x_i \in \text{depends}_n(y_{j'}), i > \text{dchange}(\text{istart}) \}
\]

There are two kind of outputs, \( O_{\text{dstop}} \), which should have at least one dependency on input before dchange element and, \( O_{\text{istart}} \), which has solely dependencies on inputs after dchange element. Thus it is impossible to have:

\[
y_{j'} \in O_{\text{istart}} \land y_j \in O_{\text{dstop}} \Rightarrow \text{depends}_o(y_{j'}) = \text{depends}_n(y_j) \land o \neq n
\]

Which is the definition of a duplicate 3.3.3

Moreover, in terms of liveness correctness guarantees, Drain Change behaves as table 3.4.

Proposition 3.5.13. **Drain Change guarantees the progress of the new version of the query.**

**Proof.** According to drain change the old version will go to the draining state, and the new version will start immediately running. Putting the new version of the query in a running state implies that new query will progress, and this is what we desired.

Proposition 3.5.14. **Drain Change does not guarantee termination of the old version of the query.**
Proof. According to drain change the old version will go to the draining state, and the new version will start running. As the old query goes to a draining state, when it changes its state to stopped is not statically computable, and will be determined by data distribution. Thus, there is nothing we can say about termination of the old version.

3.5.3 Delayed Drain Change

In Drain Change 3.5.2, there will be a time period which outputs can belong to the old or to the new query, until the old query has stopped draining. In delayed drain change, the output of the new query will only be seen, when the output of the previous query has finished. It differs to Drain Change 3.5.2 in that, the second query will not start immediately, but it waits until the first query has finished draining. Since, the point where the first query stops draining is not statically computable and will be determined by the distribution data, the formal definition of this Change Element is different than the former two. In the definition of former two Change Elements, we already knew the sequence number of the istart, istop, or dstop element, but here although we know the sequence number of Drain Stop Element, we cannot say anything about the sequence number of the Start Element, unless we run the stream once. In the implementation we do not need to know this sequence number, since the query processing engine can notify us when the draining has been finished, but in mathematical representation, we cannot make any assumption about this, and we have to leave it. There are two approaches to implement a Delayed Drain Change.

Definition (Strict Delayed Drain Change) \(STDDChange(Q, V)\), where \(Q\) is the continuous query to be changed and \(V\) is the desired version of \(Q\) to be switched to, is one of the complex control elements. \(STDDChange(Q, V)\) element, \(x_{stddchange}\), is basically a drain stop element, \(x_{dstop}\), followed in a statically not computable distance by an immediate start element, \(x_{istart}\).

\[
O = O_{dstop} \cup O_{istart}
\]

\[
O = \{ y_j \mid \exists x_i \in \text{depends}_o(y_j), i < dstop(stddchange) \} 
\]

\[
\cup \{ y_j' \mid \forall x_i \in \text{depends}_n(y_j'), i > istart(\text{statically incomputable}) \}
\]

Corollary 3.5.15. Exclusive Contribution Input elements arriving before \(x_{stddchange}\) only contribute to \(\text{depends}_o(y_j)\) and input elements arriving after \(x_{stddchange}\) are exclusively in \(\text{depends}_n(y_j')\) or \(\text{depends}_o(y_j)\).

\[
(\exists x_i \text{ where } i > stddchange \land x_i \in (\text{depends}_o(y_j) \cap \text{depends}_n(y_j'))) 
\]

\[
\land (\exists x_i \text{ where } i < stddchange \land x_i \in \text{depends}_n(y_j'))
\]

Another approach to implement Delayed Drain Change is by using a Smooth Start instead of using an Immediate Start. However, the statically incomputable behavior still remains the same. In this approach there is no exclusive contribution. Nevertheless, Some Output share same input but after old output elements comes only new output elements and there will be no mixed output.
Definition (Smooth Delayed Drain Change) \( \text{SMDDChange}(Q, V) \), where \( Q \) is the continuous query to be changed and \( V \) is the desired version of \( Q \) to be switched to, is one of the complex control elements. \( \text{SMDDChange}(Q, V) \) element, \( x_{\text{smdch}} \), is basically a drain stop element, \( x_{\text{dstop}} \), followed in a not statically computable distance by a smooth start element, \( x_{\text{sstart}} \).

\[
O = O_{\text{dstop}} \cup O_{\text{sstart}}
\]
\[
O = \{ y_j | \exists x_i \in \text{depends}_o(y_j), i < \text{dstop}(\text{smdch}) \} \\
\cup \{ y_{j'} | \exists x_i \in \text{depends}_n(y_{j'}), i > \text{sstart}(\text{statically incomputable}) \}
\]

In figure 3.5 the Strict Delayed Drain Change is demonstrated and in figure 3.6 the Smooth Delayed Drain Change can be seen.

In terms of safety correctness guarantees, both Delayed Drain Change Policies behave the same, so we can summarize them in the same table 3.5. Here, we provide proofs according to statements made in table 3.5.

**Proposition 3.5.16.** Delayed Drain Change cannot avoid Loss.

**Proof.** A Lossless Stream is already defined in 3.3.1 as a cross-reference stream,
Table 3.5: Safety guarantees offered by Delayed Drain Change

<table>
<thead>
<tr>
<th>Change Policy</th>
<th>No Loss</th>
<th>No Query-Level Disorder</th>
<th>No Window-Level Disorder</th>
<th>No Duplicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delayed Drain Change</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

as follows:

\[
O_{cref} = O_{dstop} \cup O_{istart}
\]

\[
O = \{ y_j \mid \exists x_i \in \text{depends}_o(y_j), i < dstop \} \\
\cup \{ y_{j'} \mid \forall x_i \in \text{depends}_n(y_{j'}), i > istart \}
\]

and the definition of Delayed Drain Change is as follows:

\[
O_{dchange} = O_{dstop} \cup O_{istart}
\]

\[
O = \{ y_j \mid \exists x_i \in \text{depends}_o(y_j), i < dchange(dstop) \} \\
\cup \{ y_{j'} \mid \forall x_i \in \text{depends}_n(y_{j'}), i > istart(\text{statically incomputable}) \}
\]

The statically not computable position of the istart or sstart element of the Delayed Drain Change can cause Loss; because, all the elements between the dstop component and istart or sstart component that would have produced an output using \(\text{depends}_n(y_{j'})\), did not do so. In figure 3.7, it is shown how a loss can occur in a Delayed Drain Change. The Black lines are \(\text{depends}_n(y_{j'})\), that had to produce \(y_{j'}\), but did not. However, in smooth start, there are less loss, but still loss cannot be avoided.

**Proposition 3.5.17.** Delayed Drain Change avoids Query-Level Disorders.

**Proof.** Trivially, by definition 3.5.3 Delayed Drain Change implies that until the old version did not finished draining, the new query cannot start; Thus, it is impossible that any new output being produced before the old version of query completely stops; and thus, preserving the Query-Level Order.

**Proposition 3.5.18.** Delayed Drain Change avoids Window-Level Disorder.

**Proof.** Window-Level Disorder happens when:

\[
\text{MAX}(\text{sequenceAt}(\text{depends}_o(y_j))) > \text{MAX}(\text{sequenceAt}(\text{depends}_n(y_{j'}))) \\
\land j' > j
\]

In case of a Delayed Drain Change this could never occur, because by definition of Strict Delayed Drain Change 3.5.3, there will not be any window construction allowed by the new version, before draining of the old version finishes, and by definition of Smooth Delayed Drain Change all the windows being constructed after sddchange will be discarded if they finish before draining has finished. Of course if we buffer these windows and let them out after the old version finished draining, we will have Window-Level Disorder, but in our definition this is not the case; and there is no Window-Level Disorder; there will be of course loss...

**Proposition 3.5.19.** Delayed Drain Change guarantees no duplicates.
Figure 3.7: In both Delayed Drain Change policy loss can occur.
Table 3.6: Liveness guarantees offered by Delayed Drain Change.

<table>
<thead>
<tr>
<th>Change Policy</th>
<th>Liveness Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delayed Drain Change</td>
<td>✗</td>
</tr>
</tbody>
</table>

Proof. According to definition of strict delayed drain change 3.5.3, we have Exclusive Contribution according to Corollary 3.5.15. Trivially, we do not have any duplicates. Also in case of a smooth delayed change, though we do not have Exclusive Contribution, after the draining has been finished, the new version can only produce output by using inputs of after the draining of the old version has been finished, indirectly implying that there will not be any complete overlap, between \( \text{depends}_n(y'_j) \) and \( \text{depends}_o(y_j) \). ☐

Moreover, in terms of liveness correctness guarantees, Delayed Drain Change behaves as table 3.6.

**Proposition 3.5.20.** Delayed Drain Change cannot guarantee the progress of the new version of the query.

Proof. According to delayed drain change the old version will go to the draining state, and the new version will start when the old version goes to the stopped state. As the old query goes to a draining state, when it changes its state to stop is statically not computable, and will be determined by data distribution. Thus, there is nothing we can say about termination of the old version, and consequently, there is nothing we can say about the progress or begin of the new version. ☐

**Proposition 3.5.21.** Delayed Drain Change cannot guarantee termination of the old version of the query.

Proof. According to drain change the old version will go to the draining state. As the old query goes to a draining state, when it changes its state to stop is not statically computable, and will be determined by data distribution. Thus, there is nothing we can say about termination of the old version. ☐

### 3.5.4 Graceful Change

In graceful change the old query will continue producing output, until the new query generates its first output. In the case of change, this model is the most responsive approach. The old query, needs to be stopped as soon as the new query generates its first output. Again, like the Delayed Drain Stop, this model also has a not statically computable starting point, and the start point will be indicated by the distribution of data, which is completely unpredictable. Figure 3.8 shows Graceful Change.

**Definition (Graceful Change)** \( G\text{Change}(Q, V) \), where \( Q \) is identity of the continuous query to be changed and \( V \) is the desired version of \( Q \) to be switched.
Figure 3.8: Graceful Change in Action

Table 3.7: Safety guarantees offered by Graceful Change

<table>
<thead>
<tr>
<th>Safety Criteria</th>
<th>Change Policy</th>
<th>No Loss</th>
<th>No Query-Level Disorder</th>
<th>No Window-Level Disorder</th>
<th>No Duplicates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Graceful Change</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Graceful Change is one of the complex control elements. $G\text{Change}(Q, V)$ element, $x_{\text{gchange}}$, is basically an immediate start element, $x_{\text{istart}}$, followed by an immediate stop element, $x_{\text{istop}}$. As the $Q$ receives the $x_{\text{gchange}}$ element it will start running the new query and as soon as the first output of the new query has been produced, the old query will be immediately stopped.

$$O = O_{\text{istart}} \cup O_{\text{istop}}$$
$$O = \{ y_j \mid \forall x_i \in \text{depends}_o(y_j), i < \text{istop}(\text{statically incomputable}) \}$$
$$\cup \{ y_{j'} \mid \forall x_i \in \text{depends}_n(y_{j'}), i > \text{istart}(gchange) \}$$

There is a boundary condition that should be taken care of. If the first output for new query being produced has the same closing condition as an output of the cold version, then discard the old one, and immediately stop the old version.

$$\text{if } \text{MAX}(\text{sequenceAt}(\text{depends}_o(y_j))) = \text{MAX}(\text{sequenceAt}(\text{depends}_n(y_{j'}))) \text{ THEN Discard}(y_j)$$

In terms of safety correctness guarantees, see table 3.7:

Here, we provide proofs according to statements made in table 3.7.

**Proposition 3.5.22.** Graceful Change cannot avoid Loss.

**Proof.** A Lossless Stream is already defined in 3.3.1 as a cross-reference stream, as follows:

$$O_{\text{cref}} = O_{\text{dstop}} \cup O_{\text{istart}}$$
$$O = \{ y_j \mid \exists x_i \in \text{depends}_o(y_j), i < \text{dstop} \}$$
$$\cup \{ y_{j'} \mid \forall x_i \in \text{depends}_n(y_{j'}), i > \text{istart} \}$$
Graceful Change as defined in 3.5.4, will stop the old version, as soon as the first output of the new version has been produced, meaning that the old query will be immediately stopped, and therefore the uncompleted states of the old version being discarded.

**Proposition 3.5.23.** Graceful Change avoids Query-Level Disorders.

**Proof.** Trivially, by definition 3.5.4 Graceful Change implies that as soon as an output of the new version is seen, stop producing any output of the old version; and thus, preserving the Query-Level Order.

**Proposition 3.5.24.** Graceful Change avoids Window-Level Disorder.

**Proof.** Window-Level Disorder happens when:

\[
\max \left( \text{sequenceAt}\left(\text{depends}_{o}(y_j)\right) \right) > \max \left( \text{sequenceAt}\left(\text{depends}_{n}(y_{j'})\right) \right) \land j' > j
\]

In case of a Graceful Change this could never occur, because by definition of Graceful Drain Change 3.5.4, there will not be any window construction allowed by the old version, after the new version produced its first output, and all the windows of the old version at this point will be discarded(istop).

**Proposition 3.5.25.** Graceful Change guarantees no duplicates.

**Proof.** Trivially, by definition 3.5.4 Graceful Change implies that as soon as an output of the new version is seen, stop producing any output of the old version. Referring to the boundary condition of Graceful Change, we will see that in only place that a duplicate could have occurred, it is being explicitly prevented.

\[
\text{if} \quad \max(\text{sequenceAt}(\text{depends}_{o}(y_j))) = \max(\text{sequenceAt}(\text{depends}_{n}(y_{j'}))) \quad \text{then} \quad \text{Discard}(y_j)
\]

For the elements arriving after stopping the old version, duplicate cannot occur for sure.

Moreover, in terms of liveness correctness guarantees, Graceful Change behaves as follows:

**Proposition 3.5.26.** Graceful Change guarantees the progress of the new version of the query.
Table 3.9: Safety guarantees offered by each Change Policy

<table>
<thead>
<tr>
<th>Change Policies</th>
<th>Safety Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Loss</td>
</tr>
<tr>
<td>Immediate Change</td>
<td>☒</td>
</tr>
<tr>
<td>Drain Change</td>
<td>✓</td>
</tr>
<tr>
<td>Delayed Drain Change</td>
<td>☒</td>
</tr>
<tr>
<td>Graceful Change</td>
<td>☒</td>
</tr>
</tbody>
</table>

Table 3.10: Liveness guarantees offered by each Change Policy

<table>
<thead>
<tr>
<th>Change Policies</th>
<th>Liveness Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Progress</td>
</tr>
<tr>
<td>Immediate Change</td>
<td>✓</td>
</tr>
<tr>
<td>Drain Change</td>
<td>✓</td>
</tr>
<tr>
<td>Delayed Drain Change</td>
<td>☒</td>
</tr>
<tr>
<td>Graceful Change</td>
<td>✓</td>
</tr>
</tbody>
</table>

Proof. According to graceful change the old version will still be running, and the new version will also go into a running state by receiving a graceful change control element. This fact guarantees the progress of the new version of the query.

Proposition 3.5.27. Graceful Change cannot guarantee termination of the old version of the query.

Proof. According to graceful change the old version will continue running, until the new version produces its first output, which is statically not computable and dependent on the distribution of the data.

3.6 Summary of Correctness Criteria in Change Policies

There are a lot of combinations for Complex Control Elements possible by only moving the basic control elements back and forth, and adjust their distance. Here we tried to suggest some of them, which could have a usecase in real life. In table 3.9 and table 3.10, the correctness guarantees offered by each change policy are illustrated.
Chapter 4

Refinement and Application of Model

So far, we have suggested a model to perform a continuous query modification at run-time. In first chapter, we expected the query to know about the control elements, and perform a proper behavior while encountering them. Our model implied that query Q, is a black box. In this chapter we want to break a query down, in order to make further refinements on the operator level. In this chapter we will see, it is not the query that triggers a certain action, but it the operators of a query which determine what to do. We will continue defining operators in our model. In other words, we do the mapping between our model and existing known operators in the streaming world. We must, of course, not underestimate the differences between existing data stream management systems, and the way their operators function. In this chapter, we will first break down a query to its operators, define all of the operators according to our model. Afterwards, we take a look at mismatches between our model and the application of the model to data stream management systems.

4.1 Operators

Operators are building blocks of a continuous query. In figure 4.1, a continuous query on the stream has been expanded to its building blocks. The small yellow circles inside a query box are operators; and thus, we can see that Query Q is a DAG(Directed Acyclic Graph) of operators.

According to our model, we have assumed that each operator alone should be able to also work as a whole (a query). However, we should keep in mind that not all operators have the same characteristics. These different characteristics may cause inconsistencies in their behavior while applying our model. Thus, in order to further refine the operators and extract their properties and behavior towards our model, we begin to categorize them. According to [10], there are two kinds of operators:
4.1.1 Stateless Operators

Stateless operators perform their computation on one tuple at a time without holding any state between tuples[10]. There are two stateless operators which we would like to consider for our model: Selection and Projection.

We will define these operators according to our model again. Conventionally, in this text we will use $OP$ as short for an operator.

Definition Stateless Operator: $OP$ is a Stateless Operator, if and only if all $\text{depends}(y_j)$ of its outputs have exactly one member.

$OP$ is stateless $\iff \forall y_j \in O_{op}, |\text{depends}(y_j)| = 1$

4.1.1.1 Selection

Selection operator in our model is the equivalent of a relational selection operator, meaning that it filters out input tuples which satisfy some predicates.

Definition Selection: $\sigma_\Phi$ Selection Operator is depicted as, $\sigma_\Phi$, where $\sigma$ is a unary operator, and $\Phi$ is the selection condition to be fulfilled; thus, $\sigma_\Phi(I)$ is when the selection operator has been applied on the Input Stream, $I$; and it can be seen as a function which maps exactly one Input Element to one or no output element.

$\sigma_\Phi : I \rightarrow O$
\[ Q = \sigma_{\phi} = \text{values} > 1(I) \]

Every output element resulting from applying the selection operator, \( \sigma_{\phi} \), on the input, has a \( \text{depends}(y_j) \) set of size exactly one. In other words, \( \sigma_{\phi} \), is a stateless operator.

\[ \forall y_j \in \sigma_{\phi}(I), |\text{depends}(y_j)| = 1 \]

And for all input elements, the result of applying this operator is the element itself, or nothing.

\[ \forall x_i \in I, \sigma_{\phi}(x_i) = \begin{cases} x_i & \text{if } \Phi \text{ holds for } x_i \\ \emptyset & \text{otherwise} \end{cases} \]

\( \sigma_{\phi}(x_i) = \text{contributes}(x_i) \)

In figure 4.2, the selection operator has been applied on an input stream, and the corresponding result is depicted.

### 4.1.1.2 Projection

Projection operator in our model is the equivalent of a relational projection operator.

**Definition** Projection: \( \pi_A \) Projection Operator is depicted as, \( \pi_A \), where \( \pi \) is a unary operator, and \( A \) is a set of attributes to be transformed by the projection operator in the output; thus, \( \pi_A(I) \) is when the projection operator has been applied on the Input Stream, \( I \); and it can be seen as a function which maps exactly one input element to exactly one output element.

\[ \pi_A : I \mapsto O \]

Every output element resulting from applying the projection operator, \( \pi_A \), on the input, has a \( \text{depends}(y_j) \) set of size exactly one. In other words, \( \pi_A \), is a stateless operator.

\[ \forall y_j \in \pi_A(I), |\text{depends}(y_j)| = 1 \]
And for all input elements, the result of applying this operator is the transformation of the element itself:

$$\pi_A(x_i) = \text{contributes}(x_i)$$

In figure 4.3, the projection operator has been applied on an input stream, and the corresponding result is depicted. The projection operator in this example 4.3, returns a value to the output which is twice as big as its original input value.

### 4.1.2 Stateful Operators

Rather than processing tuples in isolation, stateful operators perform computations over groups of input tuples[10]. There are several stateful operators[10], but in our model we present only Aggregation. Presenting other stateful operators is listed as future work.

**Definition Stateful Operator** $OP$ is a Stateful Operator, if and only if there exists at least one output of this operator where, the size of its $\text{depends}(y_j)$ is bigger than 1.

$$OP \text{ is stateful } \iff \exists y_j \in O_{op}, |\text{depends}(y_j)| > 1$$

#### 4.1.2.1 Aggregation

Aggregation applies one or more aggregate functions to windows over its input stream[10]. We consider aggregation as an aggregate function, such as average, sum, minimum and maximum being applied on a window. Before applying the function, the aggregate operator can optionally partition the input stream using the values of one or more other attributes (e.g., produce the average temperature for each room)[10]. In this thesis, we will not consider partition operator being applied before windowing, in order to simplify the definition of aggregate operator.

The relational version of aggregation is typically blocking; the operator may have to wait to read all its input data before producing a result. This approach is not suitable for unbounded input streams. Instead, stream processing aggregates...
perform their computations over windows of data that move with time (e.g., produce the average temperature every minute)[10].

4.1.2.2 Window

Though the window operator is part of the aggregate operator, we dedicate a separate section to it to indicate its importance. Windows are mechanisms for extracting a finite relation from an infinite stream. The concept of windows solved the blocking operator problem. In [4], the concept of sliding windows is called a technique for producing an approximate answer to a data stream query by evaluating the query not over the entire past history of the data streams, but rather only over sliding windows of recent data from the streams.

There are different aspects to categorize windows:

- Direction of movements of the endpoints: fixed window, sliding window, landmark window
- Generation Model: Time-based vs. Tuple-based vs. Predicate-based

The window generation model we will choose to define for our model is predicate-based windows. The opening and end of a predicate-based window will be determined by an opening and closing predicate. The reason why we have chosen the predicate-based window model is that it is a generalization of all other type of windows. We also do not allow landmark windows, since they are considered as duplicates in our set of correctness definitions 3.3.3.

Definition Window Operator: \( \omega_{\Phi_{\text{open}}, \Phi_{\text{close}}} \)

The Window Operator, \( \omega_{\Phi_{\text{open}}, \Phi_{\text{close}}} \), when applied on the input stream, divides the input stream into overlapping and non-overlapping windows, where the beginning of each window fulfills the \( \Phi_{\text{open}} \) condition, in other words, the opening condition, and the end of each window fulfills the \( \Phi_{\text{close}} \) condition, in other words, the closing condition; and, within the window no other item fulfills the closing condition!

4.2 Application of Model

In a query, the stateful operators are the ones that are of our interest, because their state can overlap over control elements, creating interesting and complex combinations of control elements and data elements. In case that, all operators in a query are stateless, no problem will occur, when encountering a control element, since there is no state expanding over control elements and data elements. When the state expands over one or more control elements then we could encounter some problems. Further on we will describe two of the common problems.

**IStart Problem:** To demonstrate this problem, we assume an istop is followed by an istart of the same query. An istop followed by an istart element should
not change the $depends(y_j)$ of a query, but in practice it may occur. The problem here is we cannot guarantee that running the same query twice will produce the exact same output. We cannot even guarantee that running the same query twice will produce a subset of a reference stream 3.3.1, because each time we have a new start the beginning of the new window can be shifted as thus even changing the $depends(y_j)$ which violates our model. In our model, $depends(y_j)$ and $contributes(x_i)$ functions can only be changed when a change control element comes along, and basic control elements, should no be able to change these functions.

In order to know all the $depends(y_j)$ and $contributes(x_i)$ functions, our model has an infinite knowledge about the whole stream. This is similar to the case where we have to run the query on the stream once to extract all the $depends(y_j)$ and $contributes(x_i)$ functions, which is unpractical. In figure 4.4, we see that we know beforehand about all the $depends(y_j)$ functions on the stream. According to our model, stopping and starting the same query, should not change its $depends(y_j)$ and $contributes(x_i)$ functions.

In practice the following will happen, because we do not know all the $depends(y_j)$ and $contributes(x_i)$ functions of a query before we see the whole input. In figure 4.5 we see the comparison of the output of a query, summing up 3 values, creating tumbling windows out of each consecutive 3 input values. Black output elements are getting discarded, because their window overlaps an istop, and according the istop definition, non-completed windows in case of an istop will get discarded.

In order to make our model practical we have to either live with the fact that we cannot guarantee repeatable output or add an additional condition to our model in order to have a repeatable out. The condition of having repeatable output is repeating also all the control elements as they appeared in the first iteration of the lifecycle. Repeatable output is important when having transactions, in order to redo. Another solution to this problem is using a Smooth Start instead of an Immediate Stop, but there are also some complexities and
troubles emerging for using Smooth Start. The next example will point them out.

**Smooth Start Problem:** Though the idea of Smooth Start, to inverse the functionality of Drain Stop is helpful, Smooth Start can be hardly implemented in practice. If we have a lot of queries in the system, we cannot forecast which one will receive a smooth start and will get activated, so that we can begin producing its windows without sending it to the output. In order to implement it, either we should create windows for all queries and even all versions of a single query, which is extremely inefficient, or we should somehow be able to notify that a certain query will receive a smooth start and therefore we should begin producing windows for it.

**Nested Window Problem:** Having more than one window operator can get quite complicated. The following example shows the problem it can cause when our model is being applied. Our model implies that after receiving the dstop element, no new window is allowed to be constructed. If two windows are dependent on each other, like in nested windows, the smaller one is not allowed to produce any result; thus, either blocking the bigger window from closing up, or causing the bigger window to deliver incomplete results. A possible solution for the nested windows problem is to traverse through the query plan, and find the top-most window. The draining behavior should only be considered for the top-most window, and all other windows, which lay beneath this window, can keep opening and closing windows. Although this solution violates our model definition, but in practice this cases should be correctly considered. Figure 4.6, shows how this happens.
Figure 4.6: The Drain Stop policy does not allow any window being produced after Drain Stop element.
Chapter 5

Implementation

As we have implemented our Query Modification Model. This chapter is dedicated to the explanation of the model’s implementation. First, we will introduce the query language and stream management system we have used. Then, the Basic Query Lifecycle Architecture is being demonstrated and explained. Parallel to the model, by using the basic query lifecycle architecture as a building block, we have constructed the query modification model. In this chapter, components being used or altered to make the model implementable are being explained and the interactions between them are being discussed.

5.1 XQuery as Continuous Query Language

XQuery is basically an XML query language, which is applicable across many types of XML data sources. In [5], XQuery has been extended to also allow windows operations, similar to what other continuous query languages are performing. Thus, we have selected XQuery, as the first continuous query language to implement our model on it. However, this does not mean that we are not looking for compatibility between our model and other continuous query languages such as CQL, or the one used in Borealis.

5.1.1 MXQuery as Data Stream Management System

Using XQuery as our continuous query language, we required an XQuery engine. We have implemented our model on MXQuery. MXQuery is a low-footprint, extensible open-source XQuery engine implemented in Java. Besides a high level of compliance with XQuery 1.0, it provides a wide coverage of upcoming W3C standards proposals (Update, Fulltext, Scripting), support for a wide range of Java Platforms (including mobile/embedded devices) and support for data stream processing/CEP[11].
5.2 DMCQ Component

We have named the query modification component as DMCQ, which is short for Dynamic Modification of Continuous Queries.

5.2.1 DMCQ Architecture

5.2.1.1 Query Lifecycle

First, we will describe the Query Lifecycle Architecture of the DMCQ. Query Lifecycle refers to chapter 2. As in the modeling approach, also in the implementation we tried to first implement a single query’s basic lifecycle commands. Note that due to complexities, smooth start is not implemented, thus everywhere start is mentioned, the Immediate Start is meant.

Figure 5.1 is an illustration of the DMCQ for the Basic Query Lifecycle Architecture. The basic components are as follows:

- Gatekeeper
- Drainable Operators
- Coordinator

Each query has its own gatekeeper. The main responsibility of the Gatekeeper is to read the input tokens, and when encountering control elements, perform the proper actions.

The Gatekeeper has three different states: STOPPED, DRAINING, RUNNING.
• **STOPPED**: In this state, gatekeeper keeps reading input in search for a start element, otherwise it discards the token just being read.

• **RUNNING**: In this state, gatekeeper keeps reading input and forwarding it further to the query operators in order to generate output.

• **DRAINING**: In this state, gatekeeper should work closely with the coordinator. Each new token being read by the gatekeeper, it asks coordinator whether all the open windows are closed or not. As soon as all open windows are closed, the gatekeeper changes its state to **STOPPED**.

As the Gatekeeper reads the input tokens, its changes its state as depicted in figure 5.2. In this picture, we see that gatekeeper does not change its state when receiving data elements. Finite State Machine of a Gatekeeper is quite similar to Query State Transition Figure 2.8

Drainable Operators, are stateful operators (see 4.1.2). In order to drain it is important to identify these operators in the query plan of a query. Drainable operators have also two states: NORMAL and DRAINING.

• **NORMAL**: In this state the operator functions as it should. Opening, evaluating and closing windows as normal.

• **DRAINING**: In this state the operator knows that it is in a draining condition. Meaning that it should proceed with already opened windows (the windows already opened while operator was in a NORMAL state), and avoid opening new windows, even if the opening condition of the window operator 4.1.2.2 has been met by the newly read token.
Each query has its own coordinator. The main responsibility of a Coordinator is to keep track of the drainable operators in a query. Initially the coordinator traverses the query plan, and registers all the query’s drainable operators. Coordinator has also three states: NORMAL, DRAINING, and STOPGATEKEEPER

- **NORMAL**: In this state, the coordinator is idle.

- **DRAINING**: In this state, the coordinator know that gatekeeper is draining, thus each time coordinator will be asked about its drainables’ open windows. If there are still drainables with open windows, the draining state of coordinator will persist, and gatekeeper should continue draining.

- **STOPGATEKEEPER**: In this state, all the drainables have closed their windows, and thus the gatekeeper should be notified to change its state from draining to stopped.

The Coordinator works similar to a 2-Phase commit, it waits for all the drainables to finish their windows, and then notifies the gatekeeper that it can change its state to **STOPPED**.

### 5.2.1.2 Query Modification

Query Modification component works in scale of basic elements exactly the same as the Query Lifecycle model. In case of complex control elements, meaning change control elements, it has some additional components which will are as following:

- Change Manager
- Merger
- Deployed Queries

There are different versions of a single query 3.1. Thus, deployed queries are all kept ready to be switched to, whenever a relevant change control element comes along.

Figure 5.3 depicts the Query Modification Architecture.

When a change control element comes along, first it will be recognized by the gatekeeper of the current query. Then the gatekeeper of the current query calls ChangeManager. Change Manager is only one instance for the whole system. According to the arrived change element, the change manager will know which change policy to take. A change element also carries the version of the new query. Therefore, Change manager will know to which version it should switch.

The merger is responsible to deliver the output. If the change policy implies that one query is running at a time, like Delayed Drain Change policy or immediate change policy the merger easily returns the output of the currently running query. However, in some cases the change policy implies that two queries run
in parallel, like drain change, thus the merger pulls an item from each query’s output queue and compares their provenance, in order to guarantee Window-level Order 3.3.2, an item that is dependant to input elements having a smaller sequence number will be seen in the common output first. The Merger is also one instance in the whole system.
Chapter 6

Conclusion

The Query Modification Model introduced in this thesis, takes a similar approach as the concept of punctuated stream in [13]. Making the stream carry more than just data, had a lot of advantages in simplifying the conceptual definitions of the modification models and introducing determinism into the system. Basic query lifecycle control elements, like stop and start has been introduced. Based upon basic query lifecycle elements, there has been a modification model defined, which had enabled a query to have different versions available, and it can be easily switched between these versions. Moreover, modifying a query should be done reliably. There are correctness criteria being defined to perform a reliable change. Based upon different change usecases, there are some change policies being defined, and their correctness guarantees had been proven. In order to make the model closer to its application, we refined query into its operators and found some pitfalls of the model, while being applied. However, in the implementation phase, though the model was not completely statically computable, it was implementable. The basic query lifecycle can be used also as building blocks for other models like transactions and migration. This fact shows the reusability of the model’s building blocks.

6.1 Future Work

Nevertheless, there are some works left for the future. The performance benchmarking has to be done. Measuring the mean time to change, by applying different kinds of change policies and queries is a valuable performance indicator for the model. Having proper performance indicators we may also want to apply further optimization to the model, for example by reusing some parts of the query plan. Of course there should be performance measurements done on the resource consumption, such as CPU and Memory consumption. Moreover, there could be some further refinements introduced onto different stream processing engines, like Borealis. Additionally, in order to extend the model we can consider multiple input streams, and thus be able to conclude join operator in the model. The problems introduced by using a distributed system, has not been considered in this thesis, which could also be valuable and influence
the performance and correctness of the model. As already mentioned the basic query lifecycle model can be used as building blocks for several models on the streams, like transaction model, or migration models; thus, building other models than dynamic query modification model on top of the basic query lifecycle model can also be seen as an interesting future work.
Bibliography


