Changing Flights in Mid-air: A Model for Safely Modifying Continuous Queries

Kyumars Sheykh Esmaili, Tahmineh Sanamrad, Peter M. Fischer, Nesime Tatbul
Systems Group, ETH Zurich, Switzerland
{kyumarss, sanamrat, petfisch, tatbul}@inf.ethz.ch

ABSTRACT

Continuous queries can run for unpredictably long periods of time. During their lifetime, these queries may need to be adapted either due to changes in application semantics (e.g., the implementation of a new alert detection policy), or due to changes in the system's behavior (e.g., adapting performance to a changing load). While in previous works query modification has been implicitly utilized to serve specific purposes (e.g., load management), to date no research has been done that defines a general-purpose, reliable, and efficiently implementable model for modifying continuous queries at run-time. In this paper, we introduce a punctuation-based framework that can formally express arbitrary lifecycle operations on the basis of input-output mappings and basic control elements such as start or stop of queries. On top of this foundation, we derive all possible query change methods, each providing different levels of correctness guarantees and performance. We further show how these models can be efficiently realized in a state-of-the-art stream processing engine; we also provide experimental results demonstrating the key performance tradeoffs of the change methods.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - Query Processing; H.2.4 [**Database Management**]: Systems - Transaction Processing

General Terms

Reliability, Verification

Keywords

Continuous Query, Stream Processing, Query Lifecycle, Query Modification

1. INTRODUCTION

With the proliferation of dynamically generated data and the need for its continuous monitoring to the end, we have witnessed the emergence of data stream processing as a new data management paradigm. Stream processing has proven to have a wide spectrum of practical applications with varying requirements such as real-time financial analysis, network traffic monitoring, sensor-based tracking. As a result, a significant number of academic (e.g., Borealis [6], STREAM [21], TelegraphCQ [12]) and commercial (e.g., StreamBase [4], Truviso [5], IBM InfoSphere Streams [1], MS StreamInsight [7]) stream processing engines (SPE) have been built to meet these needs.

In SPEs, continuous queries over infinitely long data streams can run for unpredictably long time periods. During their lifetime, these queries may need to be modified either due to changes in application semantics, or due to changes in the system's behavior, as we illustrate with a few examples in the next part.

1.1 Selected Use Cases

1. Security Monitoring:

Consider a bank that applies the following security policy for its ATM machines (adapted from real-world policies we observed in the compliance research project MASTER [2]): Block a customer card upon 3 failed logins at the same ATM location within a time window of 10 minutes. This policy can be implemented as a continuous count query on a 10-minute window over a stream of failed login events. Now suppose that, due to a change in regulations, the bank would like to change the window in this query to 15 minutes. If this change happens while a user has already tried 2 failed logins within 5 minutes, it is not obvious how the system should behave. A naive approach would be to replace the query with the new one immediately, discarding any existing state. In this case, the user would be able to try up to 3 more logins in the next 15 minutes in addition to the 2 failed ones in the past 5 minutes (leading to a total of 5 tries over 20 minutes, not matching any policy!). A more cautious approach would be to defer the query replacement until a time there is no incomplete query state left, but in more complex monitoring use cases such as stock trading, such times typically do not exist. (Sections 3.3.3 and 3.3.4 show solutions).

2. Sensor Networking: Consider a network of temperature and smoke sensors deployed over a forest in order to detect and monitor wildfires. Again, continuous queries can be defined over these sensor readings in order to signal unusual increases in sensor values. Whenever such activity is reported for a certain region of the forest, the firefighters want to replace the currently running query with a more specific query so that the possible fire can be located with a higher degree of certainty. However, in sensor-based applications, there is already an inherent level of uncertainty and loss, and it is also critical for the new query to take effect as fast as possible. Therefore, for this application a naive, but more responsive approach is better (leading to the solution in Section 3.3.1).

In the above two use cases, the application semantics necessitates query modification, albeit with different requirements. There are also cases where the modification is triggered by the system itself.

SIGMOD'11, June 12–16, 2011, Athens, Greece.
https://doi.org/10.1145/1989323.1989388.

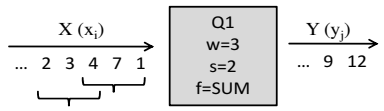


Figure 1: Running example - Q1: window size 3 slide 2 tuples.

3. Adaptive Load Management: SPEs need to deal with resource overload, e.g., caused by fluctuating arrival rates. The most common technique is load shedding, e.g., by inserting/removing load-reducing drop operators into/from selected parts of a running query plan [22]. This yields a "cheaper" version of the query, while the quality of the results becomes lower. A drawback of load shedding is that it takes the control away from the query writer and introduces non-deterministic behavior. In an alternative strategy, several versions of the same query are defined by the application in advance, each tailored for a different load level (see the military use case in [6]). As the system load changes due to fluctuations in input rates, the system is expected to switch adaptively between different query versions. It is crucial that the switch across different query versions happens seamlessly and efficiently, with as little additional system overhead as possible.

The above examples show the importance and diversity of query modification capability in SPEs. Each requires a different tradeoff between correctness parameters and performance, and provides different information to exploit. What is needed is a general-purpose, reliable, and efficiently implementable model for modifying continuous queries at run-time.

Since these use cases cover specific areas of the problem space, we now introduce a running example that will be used throughout the paper. $Q1$ uses a tuple-based sliding window of size 3 and slide 2, applying a sum operation on the window (Figure 1). Without limiting generality, we are using windows as a representative of operators whose semantics exhibit non-trivial behavior on change.

1.2 Our Contribution

In this paper, we introduce a punctuation-based framework for correctly modeling and efficiently implementing on-the-fly modifications over continuous queries. Our framework provides a set of basic control elements for starting and stopping queries, over which various advanced change variants can be defined, each providing different levels of correctness and performance. *Correctness* is defined through two types of criteria, *Safety* and *Liveness* [18]. Our model allows choosing different change variants for different use cases as well as defining new ones as needed. We further show how our model can be realized in a state-of-the-art stream processing engine ([3]) with few code changes and provide experimental results, demonstrating the key tradeoffs across different change variants.

The rest of this paper is outlined as follows. Section 2 introduces the basic framework and methodology on which our modification model is built. The description of the model itself follows in Section 3. In Section 4, we describe how our model can be refined onto real-life SPEs and provide an architecture and implementation on a state-of-the-art SPE in Section 5. Results of our performance study are provided in Section 6, giving guidelines about when to use which method. After a summary of related work in Section 7, we conclude the paper in Section 8 with directions for future work.

2. FRAMEWORK AND METHODOLOGY

Query modification can best be rooted in a general framework that tackles the boundary conditions (e.g., start, stop) of continuous queries. In order to achieve generality, precision, and deterministic behavior, this framework should not rely on semantics of specific operators, timing, or state, which are all notoriously hard to reason about (e.g., see [6] for an approach that encounters timing issues). Instead, we focus on describing input and output streams, annotating these with *punctuations* [24] which we call *Control Element*. With this approach, semantics and execution of queries are abstracted into their data dependencies. We define a minimal set of basic control elements (including their impact on output and their interaction) and derive complex control elements out of them. Within the scope of this work, we primarily focus on queries with a single input and output stream, a restriction which we plan to remove in future work. In this section, we will first introduce the foundations of our model, then the basic control elements, their interaction, and finally our methodology to model complex control elements in a generic manner. Section 3 will describe the complete execution of this methodology for *query modification*.

2.1 Foundations

To build our formal framework, we will first provide clear definitions for data streams and continuous queries. Moreover, we introduce a set of mapping functions, which provide the basis for all following definitions.

2.1.1 Streams

A stream S is an unbounded sequence of *stream elements*, where each stream element has a *tuple* part and a *position*. The *position* assures the total order among the stream elements.

Given our approach of inserting control elements into the stream, there are two types of stream elements:

- *Data Elements*, which carry regular data values.
- *Control Elements*. which carry control metadata.

Control Elements do not directly take part in query processing and therefore do not contribute to the query result. However, they convey important information to our framework regarding how the query should behave if encountering them. *Control Elements* are punctuated into the input stream either by the user or by the system itself, depending on the use case. The order among data elements can be relaxed to a partial order (for models that process elements in groups, such as STREAM [21]), only control elements need to have a total order among them and with data elements. For clarity of presentation, we assume total order among all *stream elements* in the rest of the paper.

2.1.2 Continuous Queries

A continuous query is a query that is issued once but runs forever. It takes a stream X as input and produces a stream Y as output. At any point in time t, the answer to a continuous query Q is based on the elements of its input stream X seen up to t, and this answer is updated as new stream elements continue to arrive on X, following the monotonicity definition in [19]. In our model, each continuous query Q is defined by a unique *query identifier* and a *query expression*. As a convention, we use the notation of x_i to indicate input stream elements and y_j to indicate output stream elements, where x and y correspond to the tuple parts, and i and j correspond to the positions ($i, j \in \mathbb{N}$), respectively.

2.1.3 Query Mapping Functions

We use the notion of *mapping functions* as an abstraction of the details of the query expression. Mapping functions are defined on of a pair of streams (input and output stream) and establish a relationship between a single element in one stream to a set of elements in the other. We specify two mapping functions, which capture the data dependencies established by the query expression of a given continuous query Q:

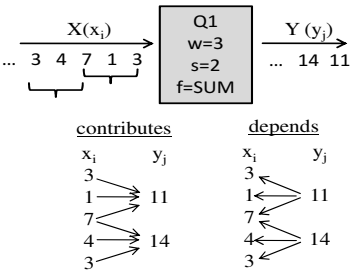


Figure 2: Mapping Functions of Q1

- *depends(y_j)*, $E \longmapsto \{E\}$: Given an output data element y_j, returns the set of all input data elements that y_j depends on.

  $depends(y_j) = \{x_i | x_i \in X \text{ where } Q(X) = y_j\}$

- *contributes(x_i)*, $E \longmapsto \{E\}$: Given an input data element x_i, returns the set of all output data elements which x_i has contributed to.

  $contributes(x_i) = \{y_j | x_i \in depends(y_j)\}$

We illustrate the query mapping functions in Figure 2, on query $Q1$ of our running example. Note that the mapping functions are not only driven by the query expression, but also the *starting position*, which is their reference point in the streams. For example, in Figure 2, a mapping starting at the input element 1 instead of 3 would not give the sequence (14,11) as a result, but one starting with 12. As we will show in the next section, this will play an important role when defining lifecycle operations. In Section 4 we will describee how mapping functions work for individual operators, and how, in turn, they can be composed for the mapping function of a complete query.

2.2 Basic Control Elements

We establish a minimal set of basic control elements, which define basic lifecycle behavior and serve as building blocks for complex control elements.

2.2.1 Start

Upon encountering a start control element, a query Q will start producing output, otherwise the arriving inputs will be ignored.

- *Fresh Start (FStart)*: We denote this control element x_{fstart} (and short **F** in figures where no details are needed), where $fstart$ is its position in the stream. Upon receiving an x_{fstart}, query Q will be started, i.e., the input data elements having a greater position than $fstart$ will contribute to the output. More formally:

  $Y_{fstart} = \{y_j \in Y \mid \forall x_i \in \textit{depends}(y_j),\ i > fstart\}$

  Y_{fstart} is the output after receiving the x_{fstart} control element.

  Figure 3 illustrates *Fresh Start* applied on $Q1$. It is important to note that a *Fresh Start* element restarts the *starting position* of the underlying query mapping functions (*depends(y_j)* and *contributes(x_i)*). As an example, applying *Fresh Start* after 1 instead of 3 in Figure 3 would shift the input sets of all windows by one position. For completeness, we also investigated start control elements that keep the mapping function as if the output had just been surpressed, e.g. for a pause/resume operation.

  We will not cover them in this paper for the following reasons: 1) Such start methods are not necessary to describe query modification, since a modification will establish a new mapping function. 2) Maintaining a mapping function incurs an overhead which can approach the cost of query execution (e.g., for a complex pattern query), making it infeasible for all but restricted cases. Details can be found in [15].

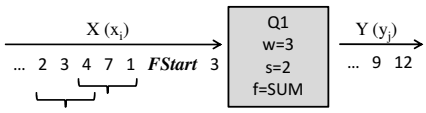


Figure 3: *Fresh Start* on Q1

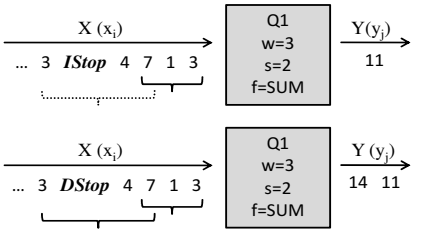


Figure 4: *Immediate Stop* vs. *Drain Stop* on Q1

2.2.2 Stop

Upon encountering a stop control element, a query Q will eventually stop producing output. We define two kinds of stop elements:

- *Immediate Stop (IStop*: We denote this control element with x_{istop} (short **I**)), where $istop$ is its position in the stream. Upon receiving an x_{istop}, a query Q will be immediately stopped, i.e., the input data elements having a position greater than $istop$ will no longer contribute to the output. More formally:

  $Y_{istop} = \{y_j \in Y \mid \forall x_i \in \textit{depends}(y_j),\ i < istop\}$

- *Drain Stop (DStop)*: We denote this control element with x_{dstop} (short **D**), where $dstop$ is its position in the stream. Upon receiving an x_{dstop}, a query Q will be gradually stopped, i.e., Q will continue to produce output which has dependencies on input data elements appearing before $dstop$ and completing its partially produced output elements. More formally:

  $Y_{dstop} = \{y_j \in Y \mid \exists x_i \in \textit{depends}(y_j),\ i < dstop\}$

Figure 4 illustrates *Immediate Stop* and *Drain Stop* applied on $Q1$, showing that IStop discards the uncompleted window, whereas DStop finishes uncompleted windows, but does not open new ones and stops when there aren't any windows left. As we will show later in the paper, both stop elements not only provide relevant semantics, but are also efficiently implementable on existing data stream systems.

2.3 Interaction of Basic Control Elements

An important aspect to completely define the semantics of query lifecycle is to cover the interaction of multiple control elements, both for the single query cases as well as for the interaction of multiple queries and expressions, which can be derived from the single query case. The interaction should maintain two design decisions established so far: (1) Control elements should become effective at the position they are specified, as defined in Section 2.2. (2) The order of control elements determines which control element is effective, superseding older ones.

We have chosen to use an interaction diagram, which can trivially be turned into an automaton by defining the start and end states for a specific operation. Figure 5 gives the states and the transitions, which correspond to the basic control elements. Initially, a query is in *Stopped* state, in which the *starting position* of the mapping function has not been set up. Using *FStart* (F), the mapping function is set up and the query transitions into the *Running* state, in which output is produced, as seen on Figure 3 for $Q1$. An *IStop* (I)

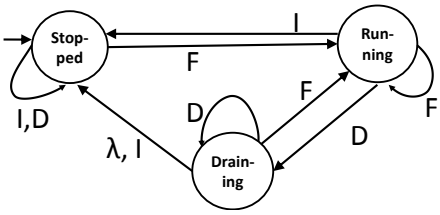


Figure 5: Lifecycle Interaction Diagram - Single Query

directly changes to the *Stopped* state, while an additional *Draining* state is needed for *DStop* (D) in which the pending output is produced. The transition from *Draining* to *Stopped* is not driven by any control element and is also no ϵ-transition, but depends on data elements (which we call λ). All these transitions can be seen in the handling of the values (3,4,7) in Figure 4: after the arrival of the *IStop* element, all following input can be ignored due to the immediate change to the stopped state. For *DStop*, the Draining state is kept until the arrival of 3, which closes the window and triggers the transition to *Stopped*. Conceptually, the λ transition is similar to the notion of Timed Automata [8], yet not based on time, but on the progress of the underlying mapping function. Applying *FStart* in any state will re-initialize the mapping function and lead to the *Running* state, while applying *IStop* when *Draining* immediately stops the query. Applying *DStop* while *Draining* has no effect, since neither extends nor restricts the output defined by the previous drain. To sum up, all this behavior is consistent with our design so far, and allows the specification of complex operations.

2.4 Creating Complex Control Elements

While the basic control elements can already support many use cases, the real benefit of this approach is that it provides a solid foundation to establish complex control and a modification model. In order to do so, we propose the following methodology:

1. Formally define the new operation (e.g., *Query Modification*) on top of our mapping functions. This leads to new complex control element(s) (e.g., *Change*), and some small extensions to our basic foundations (e.g., adding a *query version number* to the query definition that originally consists of an identifier and an expression (as defined in Section 2.1.2)).
2. Define the required/desirable behavior of the new operation, through what we call *Correctness Criteria* (e.g., safety, liveness).
3. Derive the possible options for this complex control element by the combination of the interaction diagrams of the individual queries, yielding a new interaction diagram with the powerset of the states. Choose start and end states, and sequences of transitions that connect them. Parameters influencing the number of options are (1) the types of basic control elements considered, (2) the number of control elements allowed, (3) their order, and (4) the distance between two basic control elements (e.g., directly following, data-driven, time-driven).
4. Evaluate the behavior of variants against the *Correctness Criteria*.

Using this formalism, we can establish that all possible options within the control element framework are covered. The next section will show how we apply the above methodology to define *query modification* operation.

We chose to show *query modification* as it is the most challenging and the least explored lifecycle operation. It adds a new set of semantics to describe how results should behave in the transition phase, which cannot be easily described by looking at one query. *Query Migration*, *Query Pause-Resume*, or *Query Re-Optimization* are much more constrained in this respect, can be more easily supported, and have also received much more attention in previous work. We therefore discuss them in our technical report [15].

2.5 Discussion

We have strived to build our model on as few fundamental assumptions as possible: The mapping functions and two classes of basic control elements: *start* and *stop*. The mapping functions ensure that our model can be applied on any SPE, as long as its operations are monotonic (i.e., new input does not change old results). The specific semantics of operators, their composition and the execution model can all be catered for. Our choice of basic control elements and their variants reflects the fundamental operations one can perform on mapping functions. Intuitively, they provide all that is needed and we can exhaustively derive all variants of stop/start and complex elements from there. They also cover all the lifecycle operations we have studied so far. At this time, it appears premature to establish a formal proof of their completeness with respect to all possible lifecycle operations, since it is not even clear what all lifecycle operations might be - we consider this a topic for future work.

3. QUERY MODIFICATION

Having introduced our basic query management framework and methodology in the previous section, we now show how *Query Modification* can be modeled. In *Query Modification* there are two versions of a query Q, namely Q^{old} and Q^{new}, and we would like to switch from the former to the latter. Following the first step in our methodology, we will first express the *Query Modification* operation by a new complex control element, which we call *Change* (Section 3.1). Secondly, we will define *Correctness* criteria for *Query Modification* (Section 3.2). In our basic framework, *Change* can be translated into a combination of a *Stop Control Element*, which targets Q^{old}, and a *Start Control Element*, which targets Q^{new}. However, as we will explain, there can be different variations of this combination, which we derive by interaction diagram composition, and analyze them in Section 3.3. This is complemented by an analysis of interaction of complex and basic control elements in Section 3.4. Finally, we will conclude this section by showing that our approach is as good as possible in terms of *Correctness* rules (Section 3.5).

3.1 Definition of Query Modification

In *Query Modification*, there are two versions of a query Q, old (Q^{old}) and new (Q^{new}), which are applied on a single input stream. The change control element is formalized by extending the query definition to include query versions (*old* and *new*). Accordingly, we will also use two pairs of query mapping functions: $depends_{old}(y_j^{old})$ and $contributes_{old}(x_i)$ for Q^{old}, and $depends_{new}(y_j^{new})$ and $contributes_{new}(x_i)$ for Q^{new}, respectively. In case of *Change* we will have three output streams: Output stream of Q^{old}, output stream of Q^{new}, and *Change* output stream, denoted by Y^{old}, Y^{new}, Y^{chg} respectively. A change control element defines how the Y^{chg} is built from Y^{old} and Y^{new}.

As a running example throughout the rest of this section, we want to *Change* ${Q1}^{old}$ (introduced earlier in Figure 2) into ${Q1}^{new}$, which is another continuous aggregation with a tuple-based sliding window of size 2 and slide 2, applying a sum over each window.

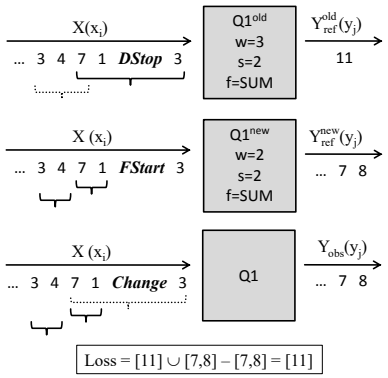


Figure 6: A lossy *Change* on Q1

3.2 Correctness of Query Modification

Inspired by the approaches in system design and verification [18], we defined two general classes of *Correctness* criteria for *Query Modification*: *Safety* and *Liveness*.

3.2.1 Safety Criteria

A safety property expresses that *"something bad will not happen"* during a given execution [18]. We identified *Loss*, *Disorder*, and *Duplicates* as possible safety problems.

1. **Loss**: A common undesirable consequence of changing a continuous query is losing some of the output elements. We formalize this concept as the set difference between *tuples* of the reference output stream (Y_{ref}) and those of the observed output stream (Y_{obs}):
   $Loss = [Y_{ref}] - [Y_{obs}]$
   where Y_{ref} is the *ideal* output stream generated by a given query Q consuming the change control element, and Y_{obs} is the output stream that Q actually generates. Intuitively, an ideal or lossless *Change* for a query Q should not lose any incomplete contributions from Q^{old}, and at the same time, it should include all contributions from Q^{new}. Note that in *Loss*, only the existence of *tuples* is considered. Their positions, which contain the order information, are captured as a separate safety issue later (hence, [ ] denoting a bag of tuples instead of a sequence).
   As an example, assume that we want to switch from $Q1^{old}$ to $Q1^{new}$ by enforcing an IStop on $Q1^{old}$ and an FStart on $Q1^{new}$ (i.e., Change = IStop + FStart). As shown in Figure 6, this leads to a lossy *Change* since it produces one fewer output (11) than then the reference stream. This reference stream would be modeled by a change using DStop + FStart.
2. **Disorder**: Order is a core property of data streams. Therefore, disorder is another critical threat to safety in continuous query execution. We identified two levels of order violation for *Change*:
   (a) **Query-Level Disorder**: In an execution that preserves query-level order, no output stream element from Q^{old} should appear after an output stream element of Q^{new}.
   (b) **Stream-Level Disorder**: In an execution that preserves stream-level order, output stream elements follow the order that is imposed by the query semantics and the structure of the input streams.

   Figure 7 depicts the difference between these two. Note that to be able to illustrate the difference better, just in this example, we used w=4 for $Q1^{old}$ instead of w=3. The upper part of the figure shows that the output appears in the same order as the windows are closing, and the lower part shows that the output appears according to the order of query versions, meaning that the output of the Q^{old} will be seen first and then the output of the Q^{new} will show up.
3. **Duplicates**: In case of *Change*, it is possible that the same exact input elements contribute to both the outputs of Q^{old} and Q^{new}. These outputs are considered duplicates, since they have the exact same dependency sets.

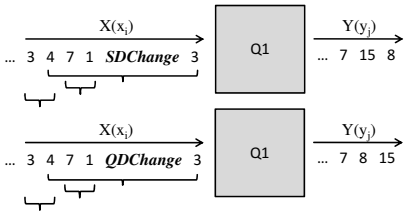


Figure 7: Query- vs. Stream-level Disorder on Q1

A complete formalization of the above can be found in technical report [15].

3.2.2 Liveness Criteria

A *Liveness* property expresses that *"something good must eventually happen"* during a given execution [18]. We identified two complementary *Liveness* criteria for *Query Modification*:

1. **Termination of the Old Query**: The system should guarantee that Q^{old} will eventually terminate, producing no more output and freeing up its occupied resources. As an example, *DStop* without any restrictions on the semantics of the query expression may lead to termination problems (e.g., closing condition of a semantic window never being satisfied).
2. **Progress of the New Query**: The system should guarantee that Q^{new} will eventually progress, starting to consume and process input. Note that *Progress* is not necessarily a property that is observable in terms of the query output, since certain query semantics may prohibit the generation of output (e.g., a selection query whose condition is never satisfied).

3.3 Change Control Elements

We will now derive the *Change* options by combining our basic control elements (Start and Stop) in different ways (i.e., Step 3 in our methodology). The key idea is to build a common interaction diagram for both query versions from the interaction diagram of a single query, as presented in Figure 5. On this common interaction diagram, we can determine and evaluate the possible options to perform the change. The resulting interaction diagram is shown in Figure 8, on which each of the states is labeled with the state for each of the queries (e.g., RS means Q^{old} is **R**unning and Q^{new} is **S**topped), and transitions are the combination of the individual query version's transitions (e.g., *IStop* for Q^{old} on RS will lead to SS). As outlined before, we want to stop Q^{old} and start Q^{new}, so our goal is to go from RS to SR. Table 1 shows the options to do so on the left-hand side. We also model the distance between executing these steps, in order to allow data operations during the change (e.g., waiting for completion of a drain). In total, there are 8 options, of which one can be discarded (IStop-FStart with waiting), since it does not provide any meaningful guarantees by adding a distance when no activity is ongoing. Some of these options are equivalent, since (1)they are executed next to each other with no data operations in between, (2) changing the order of two control elements does lead to the same target state, e.g., for both cases of *IChange*.

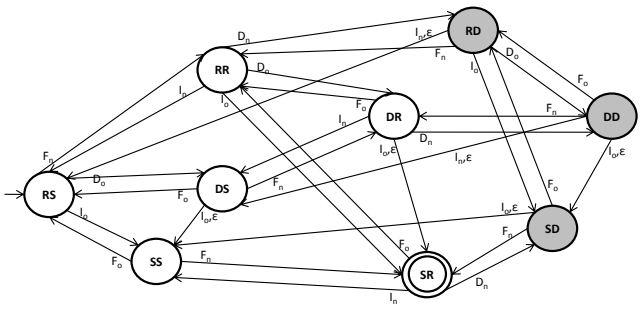


Figure 8: Lifecycle Interaction Diagram - Two Queries

Stop Type	Order	Distance	Change Option
IStop	I F	direct	IChange
IStop	I F	waiting	n/a
IStop	F I	direct	IChange
IStop	F I	at first result	GIChange
DStop	D F	direct	DChange
DStop	D F	drain completion	DDChange
DStop	F D	direct	DChange
DStop	F D	at first result	GDChange

Table 1: Change Variation Derivation

As a result, we get 5 variants of change, which we will now discuss in terms of their correctness, performance, and use cases.

3.3.1 Immediate Change (IChange)

In some applications, a *Change* should be performed as early as possible, disregarding any partial results of Q^{old}.

Formally speaking, this is expressed with an IChange control element, depicted as $x_{ichange}$, composed of an *IStop* for Q^{old}, followed directly by an *FStart* for Q^{new}, or vice versa. Thus, the output stream is defined as:

$$Y^{ichg} = Y^{new}_{fstart} \,||\, Y^{old}_{istop}$$
$$= \left\{ y^{new}_j \in Y^{new} | \, \forall x_i \in \mathit{depends}_{new}(y^{new}_j),\ i > \mathit{ichange} \right\}$$
$$|| \left\{ y^{old}_j \in Y^{old} | \, \forall x_i \in \mathit{depends}_{old}(y^{old}_j),\ i < \mathit{ichange} \right\}$$

Note that (1) || is the *append* operator, which concatenates two streams, and (2) each input element contributes exclusively to the output element of Q^{old} or Q^{new}.

Figure 9 shows an example of using IChange control element.

In terms of safety, IChange can cause *Loss*, since the partial results of Q^{old} are discarded. It produces the outputs in the correct order (both stream- and query-level), and does not generate any duplicate output elements (see Figure 10(a)). Moreover, in terms of liveness, IChange guarantees both the termination of Q^{old} as well as the progress of Q^{new} (see Figure 10(b)) [1]. With respect to the use cases introduced in Section 1, *Sensor Networking* is a candidate for IChange, since loss is tolerable while the immediate progress of the new query (in case of an emergency) and preserving the energy-saving requirements of the sensors need to be guaranteed. Given that IChange does not require *DStop* nor any other kind of complex change coordination, its behavior corresponds to what a typical SPE would do.

3.3.2 Delayed Drain Change (DDChange)

An alternative approach for *Change* is to ensure that Q^{old} is drained, and only then Q^{new} is started. We call the corresponding control element DDChange, denoted as $x_{ddchange}$. Formally

[1] Please see the technical report [15] for formal proof of these guarantees.

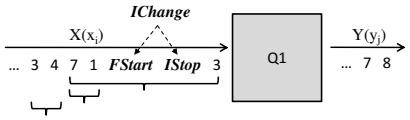


Figure 9: Immediate Change on Q1

speaking, it is composed of a *DStop* for Q^{old}, followed by an *FStart* for Q^{new} in a statically-incomputable distance . Thus, the output stream is defined as:

$$Y^{ddchg} = Y^{new}_{fstart} || \, Y^{old}_{dstop}$$
$$= \left\{ y^{new}_j \in Y^{new} | \, \forall x_i \in \mathit{depends}_{new}(y^{new}_j),\ i > ? \right\}$$
$$|| \left\{ y^{old}_j \in Y^{old} | \, \exists x_i \in \mathit{depends}_{old}(y^{old}_j),\ i < \mathit{ddchange} \right\}$$

where $'?'$ denotes the fact that the real position of the FStart element which will be inserted is not known in advance. Therefore the *starting position* of the mapping function for Q^{new} is different to that of all the other change cases (which are all initialized at the position of change), so that output after the change may also be different. Figure 11 shows an example of using DDChange control element.

In terms of safety, DDChange exhibits a behavior similar to that of IChange; it also holds the exclusive contribution property.

3.3.3 Drain Change (DChange)

Both *Change* variants presented above can induce *Loss*, which is not desirable in many streaming scenarios. This *Loss* is caused by the input elements that are no longer picked up by Q^{old} and have not yet considered by Q^{new}, since Q^{new} is only started when Q^{old} has been completed. Next, we introduce Drain Change, which is composed of a *DStop* for Q^{old} followed directly by an *FStart* for Q^{new}, or vice versa. Figure 12 shows an example of using a DChange control element.

In contrast to previous *Change* variations, merging the output streams of the old and the new queries is not straightforward in *Drain Change*, due to the fact that we have overlapping output elements. Hence, we distinguish between two variants of DChange, $x_{qdchange}$ and $x_{sdchange}$: QDChange, a Query-level order preserving *Drain Change* and SDChange, a Stream-level order preserving *Drain Change* . In QDChange is the output stream is defined as:

$$Y^{qdchg} = Y^{new}_{fstart} || \, Y^{old}_{dstop}$$
$$= \left\{ y^{new}_j \in Y^{new} | \, \forall x_i \in \mathit{depends}_{new}(y^{new}_j),\ i > \mathit{dchange} \right\}$$
$$|| \left\{ y^{old}_j \in Y^{old} | \, \exists x_i \in \mathit{depends}_{old}(y^{old}_j),\ i < \mathit{dchange} \right\}$$

while the output stream in SDChange is defined as:

$$Y^{ddchg} = Y^{new}_{fstart} \,|||\, Y^{old}_{dstop}$$
$$= \left\{ y^{new}_j \in Y^{new} | \, \forall x_i \in \mathit{depends}_{new}(y^{new}_j),\ i > \mathit{dchange} \right\}$$
$$||| \left\{ y^{old}_j \in Y^{old} | \, \exists x_i \in \mathit{depends}_{old}(y^{old}_j),\ i < \mathit{dchange} \right\}$$

in which ||| is the *interleave* operator. It interleaves the output elements from the new and the old queries based on their relative dependencies on the input elements.

Going back to Figure 7, the *Change* element at the top is behaving like a SDChange element, while the one at the bottom acts like an QDChange element.

In terms of safety, both DChange variants are lossless and free of duplicates, but neither of them can provide both order guarantees at the same time. As will bed discussed further in Section 3.5, this is not caused by the design of our *Query Modification* model, but is rather an inherent problem of *Query Modification*. In terms of liveness, both DChange variants guarantee the progress of Q^{new}, but not the termination of Q^{old}. The *DChange* variants provide a good match to the requirements of the *Security Monitoring* use

Change \ Safety	No Loss	No Query-Level Disorder	No Stream-Level Disorder	No Duplicates
IChange	✗	✓	✓	✓
DDChange	✗	✓	✓	✓
QDChange	✓	✓	✗	✓
SDChange	✓	✗	✓	✓
GIChange	✗	✓	✓	✓
GDChange	✓	✗	✓	✗

(a) Safety guarantees

Change \ Liveness	Progress	Termination
IChange	✓	✓
DDChange	✗	✗
QDChange	✓	✗
SDChange	✓	✗
GIChange	✓	✗
GDChange	✓	✗

(b) Liveness guarantees

Figure 10: *Correctness* guarantees of Change variants

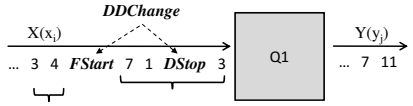


Figure 11: Delayed Drain Change on Q1

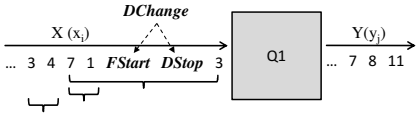


Figure 12: Drain Change on Q1 (QDChange)

case, since they avoid *Loss* and end the execution of Q^{new} by draining.

3.3.4 Graceful Change (GChange)

The approaches discussed so far did not cater for performance, in particular responsiveness, which measures the time elapsed between the last output of Q^{old} to the first output of Q^{new}. By keeping Q^{old} running (instead of draining or stopping it) until the first output of Q^{new} is produced, the responsiveness can be significantly improved. We call this change method *Graceful Change* (GChange).

We further distinguish between two variants of GChange: *Graceful Immediate Change* (GIChange) and *Graceful Drain Change* (GDChange). In GIChange, *Loss* can occur, but no *Duplicates* and *Disorder*; whereas in GDChange, *Duplicates* and query-level *Disorder* can occur, but no *Loss*. Q^{new} will make progress, since it starts immediately, but the termination Q^{old} is not guaranteed, since it depends on the existence of output of Q^{new}, which is not guaranteed. A typical good use of GChange is when Q^{new} has a low output rate (e.g., very large window size and very small window slide, very low selectivity, etc.). As shown in Section 6, GChange can indeed outperform other *Change* approaches in terms of responsiveness.

3.4 Interaction of Control Elements Revisited

So far we have defined the semantics of a *single* complex control element by a translation to several basic control elements and their interaction. In the next step, we need to cater for the interaction of *multiple* complex control elements or of a complex control element with a basic control element on the whole query. In particular, since changes (apart from IChange) do not complete immediately, such overlapping actions need to be properly defined. An examples of such a case would be an IStop on a query that currently performs a DDChange, as the user decides that results are no longer needed. In this scenario, one would expect no more output after an IStop, but the naive application of the DDChange translation means to perform a FStart after draining, which would start the query again.

Similar to the interaction of basic control elements, we therefore want to ensure that (1) control elements become effective at their specified position, and (2) the order of control elements determines that the latest control element is effective. The interaction diagram for two queries (Figure 8) does not provide a direct answer, since it only specifies the behavior of two query versions, no global operations. Yet it already contains the necessary foundations to define our extended semantics:

For a basic control element appearing during a change, we can translate IStop and DStop by applying them on both versions, thus achieving stop semantics. In turn, FStart can be translated into an IStop for Q^{old} and an FStart for Q^{new}. This ensures a start of Q^{new} with correct *starting index*, albeit with possible *Loss* on Q^{old}, since the change is turned into an IChange. A change applied after a basic control element will in any case lead to a running query, since our definition of change requires lifeness. If the query is already started or stopped, the implementation is obvious, for a draining query we can again rely on the interaction rules of basic control elements, since the stop of Q^{old} will not extend the drain period of the stop on the whole query.

For complex control elements following other complex control elements, we can build an interaction diagram for three (or more) query versions using the same rules as we built one for two query versions in Section 3.3. In the case of 3 query versions, Q^{new} of the first change is Q^{old} of the second. We then translate the actions that are applied to Q^{old} (on the two-version case) onto the first two queries. Due to space limitations and the general similarity to the two-version change, the complete correctness analysis for multiple overlapping changes is presented in our technical report [15].

As a result, we are getting a weak transactional model for change: ensuring that we always complete a change is only possible for IChange, while other change models do not provide this guarantee. In our opinion, this is actually a desirable behavior, as it allows more flexibility. In addition, stronger transactional models require giving up strict definitions of position impact and sequential order of Control Elements.

3.5 Correctness Rules for Change

In addition to covering the design space for change implementation, we investigated the correctness guarantee space. We have observed some common patterns, which we were able to compile into a set of rules. These rules help us determine that we indeed provide the strongest possible guarantees. For space reasons, we are present the actual rules in the paper, while their proofs can be found in the technical report [15].

- **LDD rule**: In the general case, a *Change* policy can ensure at most two out of three of the following guarantees: No **L**oss, No Stream-level **D**isorder, No Query-level **D**isorder.
- **LD rule**: When Q^{new} has a larger depends-set than Q^{old}, Disorder at stream level cannot occur. In this case, No **L**oss and No **D**isorder can both be guaranteed.
- **LT rule**: No **L**ossless *Change* policy can guarantee **T**ermination.

These rules show us that we have indeed covered all possible options for change when considering strong guarantees (at least two out of LDD and no duplicates, as well as termination where possible). This can be seen by comparing Table 10 with the set of all

possible combinations of correctness guarantees. Other, new options will only reduce guarantees, and these reductions are typically not useful (e.g., having no order at all, or one kind of disorder with loss). Thus we have shown that our methods to express change cover the relevant problem space and cannot be improved further for the general case.

4. REFINEMENT AND MODEL MAPPING

So far, we have investigated our model for lifecycle and change using a black-box mapping function that covers a single query with a single input and a single output. To make our model applicable in practice, we need to perform several steps: (1) Refine the mapping functions and control elements to the level of operators, and determine how control elements need to be implemented on operators and their compositions. In turn, this also allows us to work with the compositions of queries. (2) Investigate how real-life SPEs can be adapted to support the lifecycle and change model established in this work.

4.1 Operator Composition

Since operators provide the building blocks for complete queries, we now need to decrease our abstraction level to that of operators, analyze each operator and then compose them in order to achieve mapping functions and control element processing for complete queries. In a first step, we do this by generalizing our query mapping functions (Section 2.1) to operators instead of queries, and showing how to build a query mapping function from the operator mapping functions. In a second step, we determine how control elements need to be handled on this composition in order to achieve the same semantics as with a single black-box mapping function, thus completing the refinement.

Conceptually, mapping functions apply to operators in the same way as they apply to entire queries, defining on which input data item an output items depends and vice versa. We can thus complement our definition of mapping for an operators OP.

- *depends(y_j)*: Given an output data element y_j, returns the set of all input data elements that y_j depends on.

  $dependsOP(y_j) = \{x_i | x_i \in X \text{ where } OP(X) = y_j\}$

- *contributes(x_i)*: Given an input data element x_i, returns the set of all output data elements which x_i has contributed to.

  $contributesOP(x_i) = \{y_j | x_i \in dependsOP(y_j)\}$

Queries are composed of these operators, forming a query plan. In this composition, mapping functions are transitive, since the output items of one operator form the input items for the next. We can formalize this composition as follows, focusing on depends (cotributes can be expressed in an analogous way):

Given a query Q with $dependsQ()$ and operators $OP1, OP2, ...$ with $dependsOP1(), ...$, we prove: For $Q = OP1$, the single operator determines the results for the full query, thus $dependsQ() = dependsOP1()$,.

For a sequential composition, $Q = OP1||OP2$, we can see that $dependsQ(yj) = dependsOP1(dependsOP2(yj))$,
since the transitive function composition holds.

Since operators are the basic building parts of queries, their mapping functions can be derived from their formal definitions. Instead of analyzing each operator individually, we can categorize them in two classes, each with different results for our analysis:

- *Stateless* operators (e.g., selection, projection) perform their computation on one tuple at a time. More formally, for a stateless operator *op*, each output stream element y_j depends on exactly one tuple:
  $\forall y_j \in Y, |depends(y_j)| = 1$
- *Stateful* operators (e.g., window-based operators, pattern matching) perform their computation possibly on multiple tuples at a time. More formally, for a stateful operator *op*, some (if not all) output stream elements y_j depend on more than one tuple:
  $\exists y_j \in Y, |depends(y_j)| > 1$

Given this method to derive the mapping functions from formal operator specification and the composition rules for mapping functions, we now can determine the overall mapping functions of sequential query plans. More complex operators and query plans such as trees or DAGs follow the same approach, but need extensions on the definition of the mapping functions and the composition.

4.2 Control Elements on Composition

In the previous section, we have defined how query mapping functions can be composed out of operator mapping functions, thus defining how output is computed. To complete this composition, we need to determine the semantics in the presence of control elements. More specifically, the following need to be investigated, showing how each control element needs to be applied on a composition to achieve the same semantics as on the black-box query mapping function.

$$Y_{IS} = Q(X_{IS})$$
$$Y_{IS} = OP2(OP1(X_{IS}))$$
$$Y_{FS} = Q(X_{FS})$$
$$Y_{FS} = OP2(OP1(X_{FS}))$$
$$Y_{DS} = Q(X_{DS})$$
$$Y_{DS} \neq OP2(OP1(X_{DS}))$$

Applying *FStart* and *IStop* on the first operator of the composition achieves the desired semantics, since (1) the control element is defined on the input stream, (2) the first operator determines the contributions to the following operators and (3) *FStart* and *IStop* affect the output immediately. Due to the lack of space, we are not showing the full proof here. For *DStop*, this useful property does not generally hold, since the output elements which are drained cannot be determined by considering the first operator alone, unless all operators but the first are stateless. As example for such a problematic drain consider nested windows, in which draining on the first window operator will not permit the second to produce meaningful results any more. The proof for this can be found in [15]. Instead, we need a more complex coordination among operators, since the first and intermediate operators do not have enough knowledge to handle the control elements, and therefore need to delegate this task to their following operators. This delegation ends when a *dominant* operator is reached, which will determine the drain output elements. On linear plans, the last stateful operator dominates the query output, and previous stateful operators (we call them subordinate) are to be kept producing output until the dominant operator has determined all input for draining.

If there is a dominant function, the precise mapping functions of the subordinate operators are not needed any more. Therefore, we can also support user-defined functions without knowing the detailed mapping function, as long as it is monotonic. For more general query plans, a dominant operator can be constructed using techniques similar to those proposed in the literature for load-shedding on multiple aggregates [23].

4.3 Our Framework on SPE models

The query modification framework that we have introduced in this paper has been designed to be general, abstract, and conservative in terms of its assumptions, thus making it applicable in the

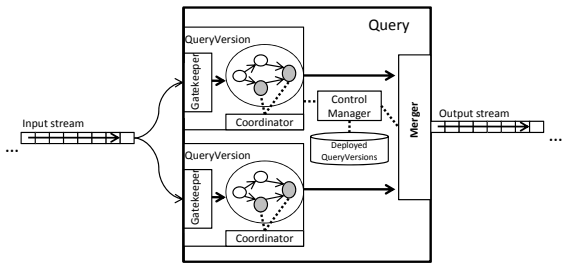


Figure 13: Query Modification architecture

context of a broad range of SPEs and their query models. In practice, individual SPEs often provide more restricted models, and therefore, our framework can be specialized for the SPE at hand. By doing so, stronger correctness guarantees and more efficient implementations can be achieved. For example, systems providing only count- and time-based windows (e.g., Borealis [6]) do, by definition, always fulfill Termination and Progress criteria. Similarly, for certain SPEs, the query mapping functions stay fixed over a Stop/Start cycle, since time-based windows are opened based on a pre-defined time domain, and are not influenced by the index position of the Start control element. Thus no issues deriving from initializing a mapping functions come up.

Our approach reaches its limitations when (1) non-monotonic operators are present, and (2) the output data elements are computed in a non-deterministic way (e.g. affected by system time). It can still be implemented, but the guarantees it can provide are inherently weaker.

5. IMPLEMENTATION

5.1 General Architecture

Up to this point, we have discussed query lifecycle, in particular query modification on the level of formal models. Since our goal for this work has been to provide a complete picture of query lifecycle, we studied ways to implement our proposed semantics. Many approaches have been studied for efficient specific lifecycle operation in databases and SPEs, of whic we will provide an overview in Section 7. Since these options typically solve a specific problem and make several assumptions in order to achieve good performance, we instead chose to develop generic architectural extensions that require only minimal changes to the SPE's data and query models, while allowing the implementation of the necessary control and change logic on top of its already existing architecture. We then show how to incorporate this architecture into existing stream processing engines. At a later stage, more optimizations can be incorporated into this architecture.

For an SPE architecture to support query modification, we must ensure that the system can keep track of multiple versions of a given query, and execute them in a coordinated way during the change period, while taking the chosen change policy and its correctness guarantees into account.To achieve this, we propose the architectural extensions shown in Figure 13. Let us now describe each of the new components in this figure.

Query Versions: An SPE will keep track of each individual *Query*, which in turn consists of a set of *Query Versions*. *Query Versions* are stored in a *Query Version Repository*, possibly in an already validated/compiled/optimized form, so as to avoid potential errors and minimize overhead during the actual query modification period. These versions can be added or removed from the repository when not required. Each *Query Version* uses its own *Gatekeeper* and *Coordinator*, and the whole repository shares a common *ControlManager* and a *Merger*.

Gatekeeper: An important aspect of our framework is the implementation of the basic control elements (i.e., FStart, IStop, DStop), since complex control elements can then be built on top of these. As shown in the previous section, *FStart* and *IStop* can be implemented by just affecting the first operator in the plan. Instead of modifying each operator to support these semantics, we place a special operator with an identity mapping function and the control logic in front of the plan, which we call *Gatekeeper*. Therefore, we do not need to change any operator, greatly simplifying the integration. Since we need to control each *Query Version* independently, there is a *Gatekeeper* for each.

Drainable Operators: *DStop*, on the other hand, requires a slightly more invasive approach for stateful operators: Stateful operators in the query plan (e.g., windowing, pattern matching, joins) need to be extended with the ability to perform draining (i.e., completing the processing of the already started windows, but not initiating new ones), yet this facility needs only to be enabled on the dominant operator. For a given windowing operator implementation, only minimal extensions are necessary, since it is already computing the contributing elements when building the windows. For example, for our MXQuery implementation, the draining extension required only about 10 LOC to be added to the windowing operator.

Coordinator: Instead of extending all operators to propagate the information necessary for a *DStop* to the dominant operator, we externalize this logic into a separate component, called *Coordinator*. It interacts with the dominant operator and the gatekeeper, passing on the relevant information and controlling the execution flow.

Control Manager: Control Manager is responsible for interpreting the control elements. If the queries are known in advance, optimizations can be performed by this component, such as identifying common subexpressions, minimizing the state to be changed.

Merger: Merger combines the output of both query versions into a single stream. The key task of the Merger is to establish the correct delivery order over the two streams. For IChange, DDChange, and QDChange, this is straight-forward, since all output of Q^{old} will be produced before that of Q^{new}. For SDChange and GChange, additional order-related metadata (e.g., starting index) needs to be known for each stream element. This component can be seen as the implementation of the *Append* ($||$), and *Interleave* ($|||$) operators.

5.2 SPE-specific Implementation

We implemented our architecture on MXQuery [10] and also studied how an implementation on top of Borealis [6] could be done. Given the differences in the data model, operator semantics and execution strategies, this should provide a good coverage of the SPEs space. *MXQuery* is an implementation of XQuery 3.0, which has few implicit assumptions on the data model (sequences of semi-structured items), expressive predicate-based windows and a set of fully composable, Turing-complete expressions. It uses a classical DBMS-style pull model, which requires explicit threading for parallel query execution, yet simplifies output control and merging, since the output is always explicitly requested, and the end of available output is explicitly indicated. Determining the relative *depends* set for two items out of different versions (as needed by the $|||$ (Interleave) operation, and thus the Merger) is conceptually difficult, given the flexible data model, the data creation operations and the large number of operators. Due to the lazy execution strategy of MXQuery (which ensures that only required data is read from the input), observing the Gatekeeper before requesting the next element gives this information in a very lightweight way, thus not requiring to change the data model and instrument the operators.

Borealis uses relational tuples in combination with a small number of streaming operators, most prominent count- and time-based

windows. Push-based operators are connected by queues and manually composed to form a query network. A scheduler can decide how to prioritize certain operators. This form of coupling simplifies parallel execution, but makes it harder to determine when all output for a given input has been produced, so that a switch can be performed. This limitation can be overcome by instructing the scheduler to prefer operators which are connected to closed or draining Gatekeeper. When operators cannot process any more, the scheduler can detect this, and indicate it to the Merger. Computing the *depends* set for the ||| operation is simple, since aggregates on windows will assign the maximum contributing timestamp to the produced tuple, which correspond to the order-by-end semantics of the window model.

6. EXPERIMENTS

Our experiments were performed on a system with an Intel Core2 Duo, 2.66 Ghz, 4 GB RAM, running Windows 7, Java 6 (both 32 bit). We ran two sets of experiments on top of MXQuery [10]: (1) A synthetic data/query set to perform a sensitivity analysis for stateful operators. (2) A Linear Road Benchmark [9] implementation to study the impact of change on complex queries under strict time constraints.

6.1 Sensitivity Analysis for Stateful Operators

Our sensitivity analyis focuses on the behavior of stateful operators, since stateless operators will have a negligible effect on change performance. MXQuery uses a predicate-based window operator which can conveniently be used to express complex windows constructs, including count- and application time-based windows [10]. We study the impact of *window size*, and *window slide* on *performance*, *correctness criteria*, and *cost*. Both versions of our query compute a sum over count-based windows, since this provides a clear way to define the workload and the expected results. The input data consists of a sequence of 2000 XML elements containing an integer payload, which are fed to the system as fast as it could consume it. The change control element is inserted after 1000 data items, ensuring that the system has reached a steady state in terms of open windows and also has enough input to complete the change. All measurements were repeated 100 times. For performance we took the averages, for correctness we checked across all these runs that we always saw the same results. Since standard deviation on all results was small, we do not report it explicitly.

6.1.1 Response Time

In the first experiment, we vary the window size of Q^{new} between 10 and 100 elements, while keeping that of the Q^{old} at 50. Both queries are using a slide of 1, providing a significant overlap amount the windows. As Figure 14 shows, the different impact of window size on the response times (time between the last element of Q^{old} and the first element of Q^{new}) is quite profound for the various methods: For *IChange* and *DChange*, the response time is linear to the size of the new window (from 1.7 msec at WS=10 to 22.6 msec at WS=100), as processing of Q^{new} only starts when Q^{old} has ended, and the processing time is proportional to the number of input items in a window. For *QDChange*, the response time is 0.2msec for window sizes of Q^{new} that are smaller than or equal to 50, since the output of Q^{new} would have been produced earlier, and needs to be held up until Q^{old} finishes. Once Q^{new} has window sizes bigger than that of Q^{old}, the same trend as for IChange is visible, because now the size of the new window dominates. The additional cost of synchronization between Q^{old} and Q^{new} causes response times to increase somewhat faster. For *SDChange*, smaller windows of Q^{new} mean that the output of Q^{new} needs to

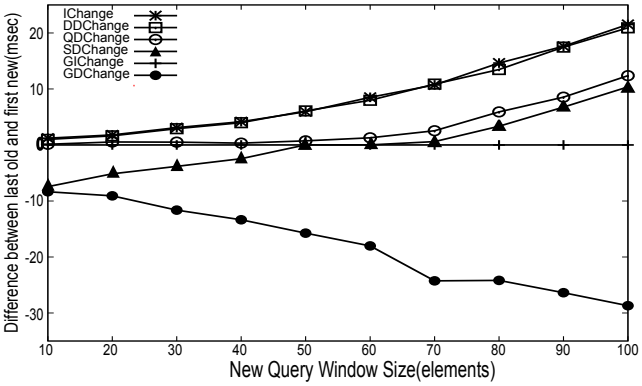


Figure 14: Responsiveness on Window Size, Slide=1

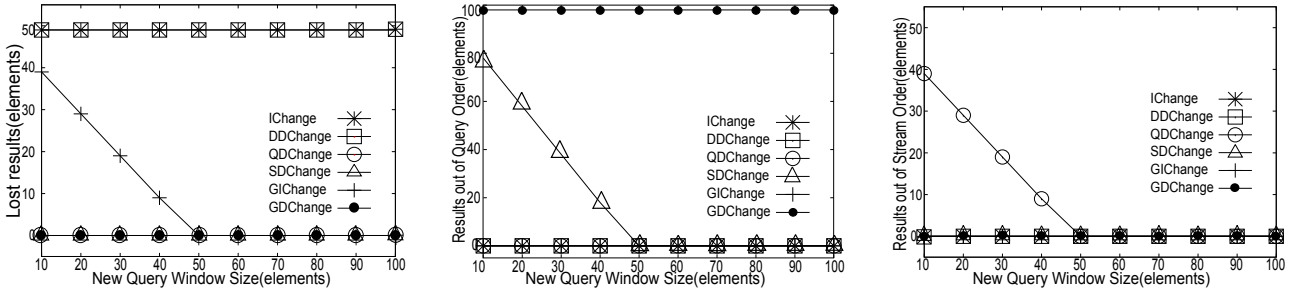
be produced before the output of Q^{old}, yielding a negative response time for the smaller values, e.g. -7 msec for WS=10. As the window size of Q^{new} increases, the response time increases, showing values similar to *QDChange* for WS greater than 50. *GIChange* shows "perfect" response times (3 microseconds), since Q^{old} is kept producing until Q^{new} can produce output, then it is terminated immediately. *GDChange* uses the same approach, but drains Q^{old}, thus showing a "negative" response time of around 10 msec, slightly more than cost of producing windows of Q^{old}, as the two queries run in parallel and need to be coordinated.

6.1.2 Correctness

We measure correctness by creating the reference streams according to the definition in Section 3.2.1 and compare the outputs against it. Figure 15(a) shows that there is constant loss (49 expected elements) for *IChange* and *DDChange*, corresponding to the loss of a complete Q^{old} window until Q^{new} picks up. *QDChange*, *SDChange*, and *GDChange* do not show any loss, since the draining of Q^{old} and the starting of Q^{new} are balanced to avoid this. *GIChange* has loss proportional to the size difference of the new window and old window, since Q^{old} receives an *IStop* as soon as the first output of Q^{new} is available, discarding the last window of Q^{old}. For disorder (Figures 15(b) and(c)) we also see the expected results: *IChange*, *DDChange*, and *GIChange* never cause any disorder, since no overlapping results are produced. *QDChange* produces results out of stream order if the window size of Q^{new} is smaller. The number of errors is proportional to the difference in window size (e.g., 39 at WS=10), since as many "smaller" windows are produced (due to the slide of 1) before the completion of Q^{old} and need to be delayed to maintain query order. In turn, *SDChange* shows the same behavior in respect to items out of query order, while *GDChange* has has a number proportional to the window size of Q^{old}, as it drains it after the start of Q^{new}. We only see duplicates (i.e, exact same input to a pair of results from both versions) when the window of size of Q^{new} is the same as that of Q^{old} and the change method is *GDChange*. Nonetheless it should be noted that both *GIChange* and *GDChange* produce additional results (with different inputs), both by the overlap of mapping functions and the output produced by Q^{old} while "waiting" for output from Q^{new} (which is not part of the reference stream).

6.1.3 Cost

The different change methods also have a different runtime overhead. In our measurements, we focused on the CPU cost, since the actual memory overhead depends on how an SPE supports the sharing of items, queues, etc. For all methods, we measured the CPU time of the main thread over the whole experiment execution as well as the use of any helper threads required to perform parallel



(a) Loss (b) Query Order (c) Stream Order

Figure 15: Correctness results for different changes

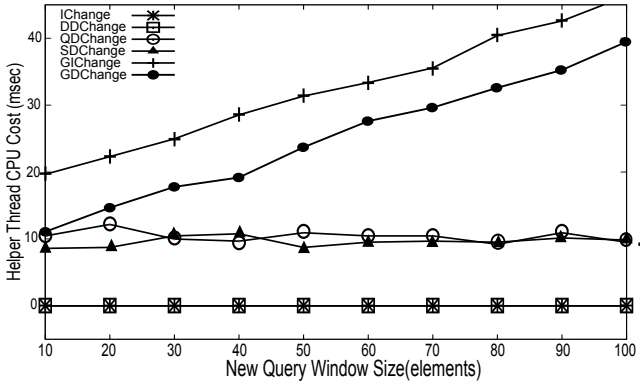


Figure 16: CPU cost on Window Size, Slide=1

query execution. The cost of the main thread is almost the same for all methods, thus we just show the results for the helper thread in Figure 16, measured in milliseconds of CPU time. Since *IChange* and *DDChange* do not execute both versions in parallel, there is obviously no cost for them. *SDChange* and *QDChange* always produce the reference output, so the relative cost stays the same. For *GIChange* and *GDChange*, additional output of Q^{old} is produced while waiting for output of Q^{new}, therefore we see a higher cost in general and an increase with the window size of Q^{new}. In our implementation, *GIChange* is slightly more expensive, since it computes output that might be discarded, while *GDChange* avoids that.

6.2 Other Experiment Results

We also performed tests with different slides and different relative positions of the change elements. The results showed the expected results. The bigger the slide and thus the smaller the overlap, the fewer correctness problems occur. If there are only tumbling windows, placing the change at the window change resulted in error-free results for all methods.

Multiple changes performed in decreasing distance did not influence each other as long as draining areas of earlier query versions did not overlap with the start of later versions. As soon as this overlap started, we would observe both additional overhead (due to a higher number of version executed in parallel) as well as duplicates and disorder.

Finally, we also tested the different change methods on the Linear Road Benchmark [9] workload, in particular on the Accident Segment query. There are several differences compared to the synthetic workload: (1) The query is significantly more complex, using multiple nested windows with predicates, grouping inside windows and parallel aggregation. (2) Data arrives using a specific timing (3) Results are expected within 5 seconds. Yet the observed results closely mirrored what we had seen on the synthetic data, with a slightly bigger impact on the arrival timing and window slide on the delays. In addition, our architecture extensions did not create a significant overhead when no change was happening.

Important	Irrelevant	Method
Runtime Resources, Implementation Overhead	Loss, Response Time	IChange
Loss, Query Order	Response Time	QDChange
Loss, Stream Order	Response Time	SDChange
Response Time, Duplicates	Runtime Resources, Implementation Overhead, Loss	GIChange
Response Time, Loss	Runtime Resources, Implementation Overhead, Duplicates, Order	GDChange

Table 2: Change Method Decision Matrix

6.3 Summary and Guidelines

The results of the conceptual as well as the experimental analysis give a fairly clear answer to when to use which change method, given that there cannot be a single winner which fulfills all criteria. We have summarized the tradeoffs in Table 2. Generally speaking, achieving zero loss and low response time incur additional implementation complexity and runtime overhead. So if neither of them is required, using *IChange* is viable in environments like sensor networks (due to limited resources) or complex query processors (due to the implementation effort). When loss must be avoided, but response time is less critical, *QDChange* and *SDChange* are the most suitable. The order that is expected during the change then determines which of them to use. Finally, Graceful Changes address Response Time, trading it off with higher resource usage. Among them, *GIChange* should be chosen if loss is tolerable, while *GDChange* should be preferred if it is not tolerable. *DDChange* is suitable only in very rare circumstances, since it does not provide stronger guarantees than IChange and requires drain support, In addition, it does not always guarantee the same results as the other change methods on Q^{new}, since the *start position* of the new mapping function is set at the end of the drain area, and not at the change element position as in all other approaches.

7. RELATED WORK

The basic vision for dynamic modification of continuous queries (CQ) was first put forth by the Borealis project [6]. The Borealis approach, however, is much more restricted: It focuses on specific operators (such as windows) with specific changes (slide or size) instead of allowing arbitrary changes on queries. No formal semantics of change are given, and the architecture ties its strategies to system time and execution speed. To our knowledge, this approach has never been implemented.

The use of control elements has been inspired by the punctuation-based stream processing work of Tucker et al. [24], yet with different semantics. In that work, data streams are annotated with punc-

tuations to mark the end of a subset of data in the stream, which are then exploited for optimizations.

Query modification shares a lot of challenges and solutions with other lifecycle problems in CQ, namely failure handling [16] and plan migration [26, 25]. The guarantees defined by these approaches are a subset of ours, e.g. Hwang et al. [16] omit order and liveness. A fundamental difference is that all of these approaches try to maintain a semantically unchanged query over changes of the infrastructure or execution plan, and change is driven by the system. In our context, change can also be triggered by the application, and therefore, we do not make assumptions about the timing and semantics of the new query.

In a similar spirit, the extensive work on adaptive query processing [14] targets a subset of the problem we are solving: No matter how the actual query execution is modified, the semantics of the query stay the same, and no errors are allowed to occur during the adaptation. Our formal framework can describe the behavior of such a system quite well, e.g. using stop and start to mark the boundaries of an adaptation. In terms of implementation, many approaches have been been studied on how to optimize this adaptation. In particular, reductions of resource consumption and improvements on transition time were studied. Many of these ideas might be relevant for our work and could be incorporated in our implementation. This will further improve performance on top of our conservative approach, which avoids re-engineering of existing SPEs.

Application-driven CQ changes are proposed by Lindeberg et al. [20], who investigate changing window sizes in order to improve results of a health monitoring use case. In contrast to our model, this model is very restricted in terms of the allowed query modifications (size change for tumbling windows) and use cases (heart attack prediction), and provides no formal correctness guarantees.

Finally, our work also relates to stopping and restarting of long-running in data warehouses [17, 11, 13]. In this case, some queries are intentionally terminated and later restarted to deal with resource contention. The restart should reuse some of the old state for efficiency reasons. Stopping and restarting the same query constitutes a special case in our more general framework. Furthermore, streaming has different semantic requirements than traditional warehouses, e.g., ordered data delivery.

8. CONCLUSION

In this work, we have presented a punctuation-based framework for correct and efficient modification of continuous queries over data streams. By representing query semantics with dependency functions, we were able to establish a general model where complex CQ lifecycle operations can be created out of the basic operations like start and stop. Moreover, our work builds on a powerful methodology that allows us to easily extend our framework even further to implement other query lifecycle operations beyond modification. We have also shown that an implementation of this framework is possible on typical SPEs, without requiring much effort or fundamental changes on the existing implementation.

Benchmark results on our prototype implementation clearly reveal the practical aspects and significant performance/correctness tradeoffs among our query modification techniques.

For future work, we foresee three directions: 1) Extending our modification model to queries with operators that accept multiple input streams (i.e., join and union); 2) Optimizing change performance by exploiting query knowledge (e.g., by identifying common subexpressions across versions to minimize change cost); 3) Utilizing our methodology for other query lifecycle operations and settings, such as query migration.

Acknowledgements.

The work published in this article was partially funded by the MASTER project (FP7-216917) under the EU 7th Research Framework Programme Information and Communication Technologies Objective . We would like to thank to Donald Kossmann for the discussions on early versions of this work. In addition, we want to thank Simonetta Zysset and Eva Ruiz for their invaluable services of proofreading.

9. REFERENCES

[1] IBM InfoSphere Streams. `http://www-01.ibm.com/software/data/infosphere/streams/`.

[2] Master: Managing Assurance, Security and Trust for sERvices. `http://www.master-fp7.eu/`.

[3] MXQuery: A Light-weight, Full-featured XQuery Engine. `http://mxquery.org/?page_id=2`.

[4] StreamBase Systems, Inc. `http://www.streambase.com/`.

[5] Truviso, Inc. `http://www.truviso.com/`.

[6] D. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR Conference*, 2005.

[7] M. H. Ali et al. Microsoft CEP Server and Online Behavioral Targeting. *PVLDB*, 2(2), 2009.

[8] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2), 1994.

[9] A. Arasu et al. Linear Road: A Stream Data Management Benchmark. In *VLDB Conference*, 2004.

[10] I. Botan et al. Extending XQuery with Window Functions. In *VLDB Conference*, 2007.

[11] B. Chandramouli et al. Query Suspend and Resume. In *ACM SIGMOD Conference*, 2007.

[12] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR Conference*, 2003.

[13] S. Chaudhuri et al. Stop-and-Restart Style Execution for Long Running Decision Support Queries. In *VLDB Conference*, 2007.

[14] A. Deshpande, Z. Ives, and V. Raman. Adaptive Query Processing. *Found. Trends databases*, 1:1–140, January 2007.

[15] K. S. Esmaili, T. Sanamrad, P. M. Fischer, and N. Tatbul. Changing Flights in Mid-air: A Model for Safely Modifying Continuous Queries. Technical Report 722, ETH Zurich, 2011. ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/7xx/722.pdf.

[16] J.-H. Hwang et al. High-Availability Algorithms for Distributed Stream Processing. In *IEEE ICDE Conference*, 2005.

[17] W. J. Labio et al. Efficient Resumption of Interrupted Warehouse Loads. In *ACM SIGMOD Conference*, 2000.

[18] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE TSE Journal*, 3(2), 1977.

[19] Y. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *VLDB*, 2004.

[20] M. Lindeberg et al. Adaptive-sized Windows to Improve Real-time Health Monitoring: A Case Study on Heart Attack Prediction. In *ACM MIR Conference*, 2010.

[21] R. Motwani et al. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR Conference*, 2003.

[22] N. Tatbul et al. Load Shedding in a Data Stream Manager. In *VLDB Conference*, 2003.

[23] N. Tatbul and S. Zdonik. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *VLDB Conference*, 2006.

[24] P. A. Tucker et al. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE TKDE Journal*, 15(3), 2003.

[25] Y. Yang et al. HybMig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries. *IEEE TKDE Journal*, 19(3), 2007.

[26] Y. Zhu et al. Dynamic Plan Migration for Continuous Queries over Data Streams. In *ACM SIGMOD Conference*, 2004.