Changing flights in mid-air
A model for safely modifying continuous queries

Author(s):
Esmaili, Kyumars Sheykh; Sanamrad, Tahmineh; Fischer, Peter Michael; Tatbul, Nesime

Publication Date:
2011

Permanent Link:
https://doi.org/10.3929/ethz-a-006905523

Rights / License:
In Copyright - Non-Commercial Use Permitted
Technical Report Nr. 722
Systems Group, Department of Computer Science, ETH Zurich

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

by
Kyumars Sheykh Esmaili
Tahmineh Sanamrad
Peter M. Fischer
Nesime Tatbul

June 2011
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 report, 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.

In this report, 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.

The rest of this report is outlined as follows. After a summary of related work in Section 2, we introduce the basic framework and methodology in Section 3, on which our modification model is built. The description of the model itself follows in Section 4. In Section 5, 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 7, giving guidelines about when to use which method.

1 The Problem Statement

In Stream Processing Engines (SPE), 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.

While analyzing the problem, we found out that different use cases have different sets of requirements for query modification. In below, by the means of three use cases, we highlight his fact. Later in this report when we explain the possible options to modify a running continuous query, we come back to each of this use case and devise the variation that fits it best.

- **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 [1]): 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 4.3.3 and 4.3.4 show solutions).

- **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 4.3.1).

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.

- **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 [14]. 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 [2]). 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.

## 2 State-of-the-Art Solutions

The basic vision for dynamic modification of continuous queries (CQ) was first put forth by the Borealis project [2]. 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. [16], yet with different semantics. In that work, data streams are annotated with punctuations 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 [9] and plan migration [18, 17]. 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 [8] 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.

Application-driven CQ changes are proposed by Lindeberg et al. [12], 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 [10, 6, 7]. 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.

3 Our Solution: A Framework and Methodology to Model Query Lifecycle Operations

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 [2] for an approach that encounters timing issues). Instead, we focus on describing input and output streams and annotate them with punctuations [16] which we call Control Elements. These control elements cause changes in the query behavior and those changes depend on the type of control elements.

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, after introducing our running example in Section 3.1, we describe 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 4 will describe the complete execution of this methodology for query modification. Moreover, in Appendix C we model Query Pause-Resume, another query lifecycle operation with a different set of requirements.
3.1 Running Example

But we first introduce a running example that will be used throughout the report. 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.

3.2 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.

3.2.1 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 [13]), 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 report.

3.2.2 Continuous Queries

A continuous query is a query that is issued once but runs (potentially) forever. It takes a stream $X$ as input and produces a stream $Y$ as output. At any time point $t$, 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 [11]. 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.

### 3.2.3 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 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 $Q$:

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

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

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

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

We illustrate the query mapping functions in Figure 2 on query $Q_1$ 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 5, we will describe how mapping functions work for individual operators, and how, in turn, they can be composed for the mapping function of a complete query.

### 3.3 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. At a high level view, there are two classes of basic control elements: Start and Stop. They essentially reflect the boundary conditions of continuous queries.

#### 3.3.1 Start

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

- Fresh Start ($F_{\text{Start}}$): We denote this control element $x_{f_{\text{Start}}}$ (and short $F$ in figures where no details are needed), where $f_{\text{Start}}$ is its position in the stream. Upon receiving an $x_{f_{\text{Start}}}$, query $Q$ will be started, i.e., the input data elements having a greater position than $f_{\text{Start}}$ will contribute to the output. More formally:
Figure 2: Mapping Functions of Q1

Y_{fstart} = \{y_j \in Y \mid \forall x_i \in \text{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)). This property makes Fresh Start unique among Basic Control Elements, because all other Basic Control Elements only use the mapping functions and do not modify them. 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.

- **Cold Start (CStart):** depicted as x_{cstart}, where cstart (short C) denotes its index. Upon receiving x_{cstart} by the query, it will start producing outputs which exclusively depend on the input items arriving after x_{cstart}. In formal terms:

  \[ Y_{cstart} = \{y_j \in Y \mid \forall x_i \in \text{depends}(y_j), i > cstart \} \]

- **Warm Start (WStart):** depicted as x_{wstart}, where wstart (short W) denotes its index. Upon receiving x_{wstart} by the query, it will start producing outputs which fully or partly depend on the input items arriving after x_{wstart}. In formal terms:

  \[ Y_{wstart} = \{y_j \in Y \mid \exists x_i \in \text{depends}(y_j), i > wstart \} \]

Figure 4 illustrates the difference between the impacts of Cold Start and Warm Start Control Elements. Notice that both Cold Start and Warm Start Control Elements rely on the existing mapping functions and do not create new ones. In other words, they always reuses the mapping functions set up by their preceding Fresh Start.

\(^1\)This has the implication that they can not be the first in series of control elements.
3.3.2 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:

  \[
  Y_{istop} = \{ y_j \in Y \mid \forall x_i \in \text{depends}(y_j), i < istop \}
  \]

- **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:

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

Figure 5 illustrates \textit{Immediate Stop} and \textit{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 report, both stop elements not only provide relevant semantics, but are also efficiently implementable on existing data stream systems.
3.4 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 3.3. (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. We have also decided to exclude Cold Start and Warm Start Control Elements; primarily because such start methods are not necessary to describe query modification, since a modification will establish a new mapping function. In addition, it simplifies the interaction diagram.

Figure 6 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 $F_{Start}$ ($F$), the mapping function is set up and the query transitions into the Running state, in which output is produced, as seen on Figure 5 for Q1. An $I_{Stop}$ (I) 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 \( \epsilon \)-transition, but depends on data elements (which we call \( \lambda \)).

All these transitions can be seen in the handling of the values (3,4,7) in Figure 5 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 \( \lambda \) transition is similar to the notion of Timed Automata, 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.

### 3.5 A Methodology to Create 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., in case of query modification, we name it 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 3.2.2)).

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
   - (a) the types of basic control elements considered,
   - (b) the number of control elements allowed,
   - (c) their order, and
   - (d) 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. As our secondary example, we model *Query Pause/Resume* in Appendix C.

## 4 Modeling Query Modification Using Our Framework and Methodology

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 4.1). Secondly, we will define *Correctness* criteria for *Query Modification* (Section 4.2).

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 4.3. This is complemented by an analysis of interaction of complex and basic control elements in Section 4.4. Finally, we will conclude our discussion by presenting a set of correctness rules for query modification in Section 4.5.

### 4.1 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: $depends_{old}(y^{old}_j)$ and $contributes_{old}(x_i)$ for $Q^{old}$, and $depends_{new}(y^{new}_j)$ 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* $Q_1^{old}$ (introduced earlier in Figure 2) into $Q_1^{new}$, which is another continuous aggregation with a tuple-based sliding window of size 2 and slide 2, applying a sum over each window.
4.2 Correctness of Query Modification

Inspired by the approaches in system design and verification [10], we defined two general classes of Correctness criteria for Query Modification: Safety and Liveness.

4.2.1 Safety Criteria

A safety property expresses that “something bad will not happen” during a given execution [10]. 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 ($Y_{obs}$), 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 $Q^{new}$.

   We can formally express Loss as follows:

   $$Loss = [Y_{ref}] - [Y_{obs}]$$

   Here we use the $[ ]$ operator, which turns a stream of elements into a bag of tuples, thus creating an unordered collection of the data parts. Moreover, individual streams are defined as follows:

   $$Y_{ref}^{old} = \{y|y_j \in Y_{old}^{old} \land \exists x_i \in \text{depends}_{old}(y_j^{old}) \land i < chg\}$$

   $^{2}$ In other words, the reference stream would be modeled by using DStop + FStart.
$Y_{new} = \{ y | y_j \in Y^{new} \land \forall x_i \in \text{depends}_{new}(y^{new}_j) \land i > chg \}$

where $chg$ denotes the position of the change element.

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 $Q^{old}$ to $Q^{new}$ by enforcing an IStop on $Q^{old}$ and an FStart on $Q^{new}$ (i.e., $\text{Change} = \text{ISTop} + \text{FStart}$). As shown in Figure 7, this leads to a lossy Change since it produces one fewer output (11) than then the reference stream.

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 $Q^{new}$.

Formally, this is defined as follows:

$$\exists y^{chg}_j, y^{chg}_j' \in Y^{chg} :$$
$$\text{org}(y^{chg}_j) \in Y^{new} \land \text{org}(y^{chg}_j') \in Y^{old}$$
$$\land j < j'$$

where

$$\text{org}(y^{chg}_j) = \begin{cases} y^{old}_i & \text{if } y^{chg}_j \text{ is taken from } Y^{old} \\ y^{new}_k & \text{else} \end{cases}$$

(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. Formally, this is defined as follows:

$$\exists y^{chg}_j, y^{chg}_j' \in Y^{chg} :$$
$$\text{org}(y^{chg}_j) \in Y^{new} \land \text{org}(y^{chg}_j') \in Y^{old}$$
$$\land j > j' \land$$
$$\max \left( \text{indexOf} \left( \text{depends}_{new}(\text{org}(y^{chg}_j)) \right) \right) <$$
$$\max \left( \text{indexOf} \left( \text{depends}_{old}(\text{org}(y^{chg}_j')) \right) \right)$$

where

$$\text{indexOf}(X) = \{ i | x_i \in X \}$$

Figure 8 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 $Q^{old}$ 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.

---

The $\max$ function here can be considered as generalization of the ordering windows by their last items. In definition of Stream-Level Disorder, other types of statistical representatives can also be applied as well. Another common example can be $\min$ (order by first item of the windows).
3. Duplicates: In case of Change, it is possible that the same exact (infinite sequences of) 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.

This is formally defined as follows:

$y_{j}^{chg}$ is a duplicate of $y_{j}^{chg'}$ if and only if:

$$\text{depends}_{new}(\text{org}(y_{j}^{chg})) = \text{depends}_{old}(y_{j}^{chg'})$$

Note that $y_{j}^{chg}$ and $y_{j}^{chg'}$ can be duplicates even if they carry different values, as long as they depend on the same input stream elements $X$.

4.2.2 Liveness Criteria

A Liveness property expresses that “something good must eventually happen” during a given execution [10]. 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).

It should be noticed that termination of the old version does not necessarily imply the progress of the new version and vice versa.
4.3 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 (simplified) interaction diagram of a single query, as presented in Figure 6.

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 9, on which each of the states is labeled with the state for each of the queries (e.g., RS means \( Q_{\text{old}} \) is Running and \( Q_{\text{new}} \) is Stopped), and transitions are the combination of the individual query version’s transitions (e.g., IStop for \( Q_{\text{old}} \) on RS will lead to SS). As outlined before, we want to stop \( Q_{\text{old}} \) and start \( Q_{\text{new}} \), so our goal is to go from RS to SR.

Once the initial state and the final state have been identified, there are four parameters which influence the number of paths between them (notice that each distinct path defines a variation of Change Control Element):

1. Types of the Basic Control Elements: here we have one option for Start (FStart), and two options for Stop (IStop and DStop).
2. Number intermediary states: in case of Change, we restrict this to be at most two.
3. Order of Basic Control Elements: which dictates how Basic Control Elements follow/precede each other.
4. Distances between the Basic Control Element: this parameter considers the number of data elements between the Basic Control Elements. Although theoretically, distances of any arbitrary length are possible, in practice a small subset of these length are meaningful:
   (a) zero length: Basic Control Elements follow each other directly (no data element in between)
   (b) in accordance to the preceding Basic Control Element (the drain distance for DStop, and the first output for FStart)

As shown in the Tree in Figure 10 and Table 1, 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.

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

4.3.1 Immediate Change (IChange)

In some applications, a Change should be performed as early as possible, disregarding any partial results of \( Q_{\text{old}} \).

Formally speaking, this is expressed with an IChange control element, depicted as \( x_{\text{change}} \), composed of an IStop for \( Q_{\text{old}} \), followed directly by an FStart for \( Q_{\text{new}} \), or vice versa. Thus, the output stream is defined as:
Figure 9: Lifecycle Interaction Diagram - Two Queries

<table>
<thead>
<tr>
<th>Stop Type</th>
<th>Order</th>
<th>Distance</th>
<th>Change Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>IStop</td>
<td>I F</td>
<td>direct</td>
<td>IChange</td>
</tr>
<tr>
<td>IStop</td>
<td>I F</td>
<td>waiting</td>
<td>n/a</td>
</tr>
<tr>
<td>IStop</td>
<td>F I</td>
<td>direct</td>
<td>IChange</td>
</tr>
<tr>
<td>IStop</td>
<td>F I</td>
<td>at first result</td>
<td>GICchange</td>
</tr>
<tr>
<td>DStop</td>
<td>D F</td>
<td>direct</td>
<td>DChange</td>
</tr>
<tr>
<td>DStop</td>
<td>D F</td>
<td>drain completion</td>
<td>DDChange</td>
</tr>
<tr>
<td>DStop</td>
<td>F D</td>
<td>direct</td>
<td>DChange</td>
</tr>
<tr>
<td>DStop</td>
<td>F D</td>
<td>at first result</td>
<td>GDChange</td>
</tr>
</tbody>
</table>

Table 1: Change Variation Derivation

\[
Y^{\text{ichange}} = Y^{\text{new}}_{\text{fstart}} || Y^{\text{old}}_{\text{istop}}
\]

\[= \{ y^{\text{new}}_j \in Y^{\text{new}} \mid \forall x_i \in \text{depends}_{\text{new}}(y^{\text{new}}_j), \ i > \text{ichange} \}\]

\[\ || \{ y^{\text{old}}_j \in Y^{\text{old}} \mid \forall x_i \in \text{depends}_{\text{old}}(y^{\text{old}}_j), \ i < \text{ichange} \}\]

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

Figure 11 shows an example of using IChange control element.

In terms of safety, IChange can cause loss, since the partial results of \(Q^{\text{old}}\) 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 12(a)). Moreover, in terms of liveness, IChange guarantees both the termination of \(Q^{\text{old}}\) as well as the progress of \(Q^{\text{new}}\) (see Figure 12(b)). Proofs for these claims can be found in
Section A

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.
4.3.2 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, 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:

$$Y^{ddchge} = Y^{new}_{fstart} || Y^{old}_{dstop}$$

$$= \{ y^{new}_j \in Y^{new} | \forall x_i \in \text{depends}_{new}(y^{new}_j), i > \}$$

$$|| \{ y^{old}_j \in Y^{old} | \exists x_i \in \text{depends}_{old}(y^{old}_j), i < ddchange \}$$

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 13 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. However, DDchange does not offer neither termination nor progress guarantees.

4.3.3 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 14 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:

\[
Y_{qdchg} = Y_{fstart}^{new} \parallel Y_{dstop}^{old}
\]

\[
= \{ y_{j}^{new} \in Y_{fstart}^{new} | \forall x_i \in \text{depends}_{\text{new}}(y_{j}^{new}), i > dchange \} \parallel \{ y_{j}^{old} \in Y_{dstop}^{old} | \exists x_i \in \text{depends}_{\text{old}}(y_{j}^{old}), i < dchange \}
\]

while the output stream in SDChange is defined as:

\[
Y_{ddchg} = Y_{fstart}^{new} \parallel Y_{dstop}^{old}
\]

\[
= \{ y_{j}^{new} \in Y_{fstart}^{new} | \forall x_i \in \text{depends}_{\text{new}}(y_{j}^{new}), i > dchange \} \parallel \{ y_{j}^{old} \in Y_{dstop}^{old} | \exists x_i \in \text{depends}_{\text{old}}(y_{j}^{old}), i < dchange \}
\]

in which \( \parallel \) 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. Therefore, it may result in query-level disorder.
Going back to Figure 8, 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 be discussed further in Section 4.5, 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 case, since they avoid Loss and end the execution of $Q_{new}$ by draining.

4.3.4 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 7, GChange can indeed outperform other Change approaches in terms of responsiveness.

Graceful Immediate Change is also a candidate for the Security Monitoring use case, because it avoids Loss of the $Q_{old}$ until $Q_{new}$ produces its first output and provides better responsiveness.

4.4 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 9) 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:

---

4 We use the Stream-ordered variation of it.
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 \( Q^{\text{old}} \) and an FStart for \( Q^{\text{new}} \). This ensures a start of \( Q^{\text{new}} \) with correct starting index, albeit with possible Loss on \( Q^{\text{old}} \), 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 \( Q^{\text{old}} \) 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 4.3. In the case of 3 query versions, \( Q^{\text{new}} \) of the first change is \( Q^{\text{old}} \) of the second. We then translate the actions that are applied to \( Q^{\text{old}} \) (on the two-version case) onto the first two queries. The correctness analysis for multiple overlapping changes is analogous.

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.

### 4.5 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. Proofs are provided in Section A.

- **LDD rule**: In the general case, a Change policy can ensure at most two out of three of the following guarantees: No Loss, No Stream-level Disorder, No Query-level Disorder.

- **LD rule**: When \( Q^{\text{new}} \) has a larger depends-set than \( Q^{\text{old}} \), Disorder at stream level cannot occur. In this case, No Loss and No Disorder can both be guaranteed.

- **LT rule**: No Lossless Change policy can guarantee Termination.

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 12 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.

### 5 Concretization of the Model

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.

5.1 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 3.2) 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 $depends$ ($contributes$ can be expressed in an analogous way):

Given a query $Q$ with $dependsQ()$ and operators $OP1, OP2, \ldots$ with $dependsOP1(), \ldots$, 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

$$dependsQ(y_j) = dependsOP1(dependsOP2(y_j)),$$

since the transitive function composition holds.

Because 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 each output stream element $y_j$ depends on exactly one tuple:
\( \forall y_j \in Y, |\text{depends}(y_j)| = 1 \)

- 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 some (if not all) output stream elements \( y_j \) depend on more than one tuple:

\( \exists y_j \in Y, |\text{depends}(y_j)| > 1 \)

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.

### 5.2 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: how is each control element handled in a composition of operators? To answer this question we first formally define a simple two-operator query:

\[
Y = Q(X)
\]

\[
Q :: OP_1 || OP_2
\]

\[
Z = OP_1(X)
\]

\( Z \) is the intermediate stream

\[
Y = OP_2(Z)
\]

\[
\rightarrow Y = OP_2(OP_1(X))
\]

\[
\text{dep}_{OP_1} : Z \rightarrow X
\]

\[
\text{dep}_{OP_1} : Y \rightarrow Z
\]

\[
\text{dep}_Q : Y \rightarrow X
\]

Following statements hold for this query network (and linear compositions in general):

\[
Y_{FS} = Q(X_{FS})
\]

\[
Y_{FS} = OP_2(OP_1(X_{FS}))
\]

\[
Y_{CS} = Q(X_{CS})
\]

\[
Y_{CS} \neq OP_2(OP_1(X_{CS}))
\]
\[ Y_{WS} = Q(X_{WS}) \]
\[ Y_{WS} \neq \text{OP}2(\text{OP}1(X_{WS})) \]
\[ Y_{IS} = Q(X_{IS}) \]
\[ Y_{IS} = \text{OP}2(\text{OP}1(X_{IS})) \]
\[ Y_{DS} = Q(X_{DS}) \]
\[ Y_{DS} \neq \text{OP}2(\text{OP}1(X_{DS})) \]

In which \( X_{IS} \) denotes an input stream which contains an Immediate Stop Control Element.

In short, applying \( F\text{Start} \) and \( I\text{Stop} \) 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) \( F\text{Start} \) and \( I\text{Stop} \) affect the output immediately.

However, for \( D\text{Stop} \) (and \( C\text{Start} \) and \( W\text{Start} \) as well), 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 these claims can be found in Section B.

In such cases, 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 [15].

5.3 Our Framework on SPE models

The query modification framework that we have introduced in this report has been designed to be general, abstract, and conservative in terms of its assumptions, thus making it applicable in the 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 [2]) 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.
As proof of concept, in Section 6.2, we will show how our framework can be specialized for two very different SPEs: MXQuery as a pull-based, language-oriented implementation, and Borealis[2] as a push-based query network.

6 Implementation

Here, we first propose our general architecture for query lifecycle management in Section 6.1 and then show how to incorporate this architecture into existing stream processing engines.

6.1 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 which we provided an overview in Section 2.

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.

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 15. 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 MX-Query 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.

6.2 SPE-specific Implementation

We implemented our architecture on MXQuery [5] and also studied how an implementation on top of Borealis [2] 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
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, on the other hand, 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.

7 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 [5]: (1) A synthetic data/query set to perform a sensitivity analysis for stateful operators. (2) A Linear Road Benchmark [4] query to study the impact of change on complex queries.

7.1 Sensitivity Analysis for Stateful Operators

Our sensitivity analysis 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 [5]. 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 template query in our experiments is shown in Listing 1. 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.

```sql
1. declare variable $input external;
2. for sliding window $w in $input//seq
```
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.

### 7.1.1 Response Time

In the first experiment, we vary the window size of \( Q_{\text{new}} \) between 10 and 100 elements, while keeping that of the \( Q_{\text{old}} \) at 50. Both queries are using a slide of 1, providing a significant overlap amount the windows.

As Figure 16 shows, the different impact of window size on the response times (time between the last element of \( Q_{\text{old}} \) and the first element of \( Q_{\text{new}} \)) is quite profound for the various methods: For \( I\text{Change} \) and \( D\text{Change} \), 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 \( Q_{\text{new}} \) only starts when \( Q_{\text{old}} \) has ended, and the processing time is proportional to the number of input items in a window.

For \( Q\text{DChange} \), the response time is 0.2msec for window sizes of \( Q_{\text{new}} \) that are smaller than or equal to 50, since the output of \( Q_{\text{new}} \) would have been produced earlier, and needs to be held up until \( Q_{\text{old}} \) finishes. Once \( Q_{\text{new}} \) has window sizes bigger than that of \( Q_{\text{old}} \), the same trend as for \( I\text{Change} \) is visible, because now the size of the new window dominates. The additional cost of synchronization between \( Q_{\text{old}} \) and \( Q_{\text{new}} \) causes response times to increase somewhat faster. For \( S\text{DChange} \), smaller windows of \( Q_{\text{new}} \) mean that the output of \( Q_{\text{new}} \) needs to be produced before the output of \( Q_{\text{old}} \), yielding a negative response time for the smaller values, e.g. -7 msec for WS=10. As the window size of \( Q_{\text{new}} \) increases, the response time increases, showing values similar to \( Q\text{DChange} \) for WS greater than 50.

\( G\text{IChange} \) shows “perfect” response times (3 microseconds), since \( Q_{\text{old}} \) is kept producing until \( Q_{\text{new}} \) can produce output, then it is terminated immediately. \( G\text{DChange} \) uses the same approach, but drains \( Q_{\text{old}} \), thus showing a “negative” response time of around 10 msec, slightly more than cost of producing windows of \( Q_{\text{old}} \), as the two queries run in parallel and need to be coordinated.

### 7.1.2 Correctness

We measure correctness by creating the reference streams according to the definition in Section 4.2.1 and compare the outputs against it. Figure 17(a) shows that there is constant loss (49 expected elements) for \( I\text{Change} \) and \( D\text{DChange} \), corresponding to the loss of a complete \( Q_{\text{old}} \) window until \( Q_{\text{new}} \) picks up. \( Q\text{DChange} \), \( S\text{DChange} \), and \( G\text{DChange} \) do not show any loss, since the draining of \( Q_{\text{old}} \) and the starting of \( Q_{\text{new}} \) are balanced to avoid this. \( G\text{IChange} \) has loss proportional to the size difference of the new window and old window, since \( Q_{\text{old}} \) receives an \( I\text{Stop} \) as soon as the first output of \( Q_{\text{new}} \) is available, discarding the last window of \( Q_{\text{old}} \).

For disorder (Figures 17(b) and(c)) we also see the expected results: \( I\text{Change} \), \( D\text{DChange} \), and \( G\text{IChange} \) never cause any disorder, since no overlapping results are produced. \( Q\text{DChange} \) produces results out of

```sql
4  start $first at $s when fn:true()
5  only end $last at $e when $e - $s = WindowSize
6  return <sum>{sum($w)}</sum>
```

Listing 1: The template query used in the sensitivity analysis experiments
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).

7.1.3 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 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 18, 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.

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.
7.2 Complex Queries

Furthermore, we tested the different change methods on the Linear Road Benchmark [4] workload, in particular on the Accident Segment query (Listing 2). 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.

```xml
declare variable $ReportedCarPositionsSeq external;
forseq $w in $ReportedCarPositionsSeq early sliding window
start curItem $s curr, prevItem $s prev when $s curr/@minute ne $s prev/@minute
end curItem $e curr, nextItem $e next when ($s e curr/@minute + 2) eq ($e next/@minute)
let $currMin := fn:ceiling ($e curr/@minute)
let $stopedCars :=
for $rep in $w
group $rep as $r
−
group by $rep/@VID as $vid s, $rep/@XWay as $xway s, $rep/@Seg as $seg s, $rep/@Dir as $dir s, $rep/@Lane as $lane s, $rep/@Pos as $pos s
where count($r − group) ge 4
return <stopped car VID="{$vid s}"
XWay="{$xway s}"
Seg="{$seg s}"
Dir="{$dir s}"
Lane="{$lane s}"
Pos="{$pos s}">
</stopped car>
let $accidents :=
for $car in $stopedCars
group $car as $c
−
group by $car/@XWay as $xway a, $car/@Seg as $seg a, $car/@Dir as $dir a, $car/@Lane as $lane a, $car/@Pos as $pos a
where count($c − group) ge 2
return <accident minute="{$currMin}"
XWay="{$xway a}"
Seg="{$seg a}"
Dir="{$dir a}"
Lane="{$lane a}"
Pos="{$pos a}">
</accident>
let $accidentsRes := if ( count($accidents) gt 0 ) then <accidents>
{ $accidents } </accidents>
else <accidents>
<accident minute="{$currMin}"
XWay=""−1"
Seg=""−1"
Dir=""−1">
</accident>
</accidents>
return $accidentsRes
```

Listing 2: The Accident Segments query from the LR Benchmark [4]

$Q^{old}$ of the accident query defines a sliding window with a size of 120s and a slide of 30s, producing output every 30 seconds, shown as grey bars in Figure 19. For $Q^{new}$, the window size was varied from 60 seconds to 360 seconds. There is an increasing response time for all methods not using the graceful approach, since the waiting time for result of $Q^{new}$ increases with increasing window size, while $Q^{old}$ terminates after a fixed time.

The graceful approaches ($GIChange$, $GDChange$) overcome this problem by keeping $Q^{old}$ producing until there is result for $Q^{new}$, thus yielding a stable response time. In particular, $IChange$ and $DDChange$ show the highest response time: $WS(Q^{new}) + 30$ seconds (e.g. $Q^{new}$ 150 seconds at WS 120), since $Q^{new}$ is only started when $Q^{old}$ has terminated. The additional 30 seconds are required to find the next start condition for a window.

$QDChange$ and $SDChange$ improve the responsiveness by starting $Q^{new}$ already at the change, thus yielding a response time of $WS(Q^{new}) - WS(Q^{old}) + 30$ sec (as to open the new window). Once this dif-
ference becomes smaller than 0, the two methods differ, since a particular order needs to be established: \(QOChange\) needs to delay the output of \(Q^{new}\) until the last output of \(Q^{old}\) has been produced, therefore emitting (almost) at the same time, and out of order to their window specifications. \(SDChange\) produces the first output of \(Q^{new}\) 30s before the last output of \(Q^{old}\), thus creating query-level disorder. \(GIChange\) stops \(Q^{old}\) once output for \(Q^{new}\) is available (as to avoid duplicates), so there is a constant 30s difference. \(GDChange\) drains \(Q^{old}\) when the first output of \(Q^{new}\) is available, as to avoid loss, leading to an additional output with WS(\(Q^{old}\)). If the window sizes do not have such clear relationships, there will different numerical values, but the overall trends stay the same.

7.3 Tradeoffs 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.
<table>
<thead>
<tr>
<th>Important</th>
<th>Irrelevant</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime Resources, Implementation Overhead</td>
<td>Loss, Response Time</td>
<td>IChange</td>
</tr>
<tr>
<td>Loss, Query Order</td>
<td>Response Time</td>
<td>QDChange</td>
</tr>
<tr>
<td>Loss, Stream Order</td>
<td>Response Time</td>
<td>SDChange</td>
</tr>
<tr>
<td>Response Time, Duplicates</td>
<td>Runtime Resources, Implementation Overhead, Loss</td>
<td>GIChange</td>
</tr>
<tr>
<td>Response Time, Loss</td>
<td>Runtime Resources, Implementation Overhead, Duplicates, Order</td>
<td>GDChange</td>
</tr>
</tbody>
</table>

Table 2: Change Method Decision Matrix

8 Summary

In this report, 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.

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 [10]. Safety criteria capture issues such as Loss, Duplicates, and Disorder, whereas Liveness criteria capture issues such as termination of the old version of the query and progress of its new counterpart. Our model allows choosing different change variants for different use cases as well as defining new ones as needed.

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.
A Guarantee Proofs

Here we provide proofs for a representative set of claims we have made in table 12(a) and table 12(b). In order to the redundancy we present the proofs in the column-major order.

Safety Guarantees

Loss

Here we first recall the definition of the Reference Stream which we defined the Loss based on.

\[ Y_{old}^{ref} = \{ y_j \in Y_{old} \land \exists x_i \in \text{depends}_{old}(y_j) \land i < chg \} \]

\[ Y_{new}^{ref} = \{ y_j \in Y_{new} \land \forall x_i \in \text{depends}_{new}(y_j) \land i > chg \} \]

\[ Y_{ref} = Y_{ref}^{old} \| Y_{ref}^{new} \]

Proposition A.1 Drain Change is lossless.

Proof It is easily noticeable that reference stream is defined in the same way that we define output stream for Drain Change. Hence Drain Change (both variation, SDChange and QDChange) is always lossless. It should be noted that although the Interleaving operator (\(\|\)) differs from the Append operator (\(||\)) in terms of ordering the output items, but they are identical with regard to the number of output tuples.

Proposition A.2 Immediate Change can be lossy.

Proof According to the definition of Immediate Change, the output stream is as follows

\[ Y_{ichg} = Y_{fstart}^{new} \| Y_{istop}^{old} \]

Since we have

\[ [Y_{istop}^{old}] \subseteq [Y_{dstop}^{old}] \Rightarrow [Y_{ichg}] \subseteq [Y_{ref}] \]

which means Immediate Change can cause loss.

Proposition A.3 Delayed Drain Change cannot avoid Loss.

Proof definition of output stream of Delayed Drain Change is as follows:

\[ Y_{ddchg} = Y_{fstart}^{new} \| Y_{dstop}^{old} \]

\[ = \{ y_j^{new} \in Y_{fstart}^{new} \mid \forall x_i \in \text{depends}_{new}(y_j^{new}), i > ? \} \]

\[ \| \{ y_j^{old} \in Y_{dstop}^{old} \mid \exists x_i \in \text{depends}_{old}(y_j^{old}), i < ddchange \} \]
where ’?’ denotes the fact that the real position of the FStart element which will be inserted is not known in advance.

The fact that application of FStart element is delayed can cause Loss. More concretely, all those output items from the new query which would been produced by input tuples between the initial position of the DDChange and the effective position of FStart are lost.

It should be noted that in contrast to all other Change variations, Delayed Drain Change uses a different pair of mapping functions. This is due to the fact that in this case, the effective position of FStart id determined by old query’s mapping functions.

Disorder

Here we prove a general proposition and refer to it in the relevant proofs.

Proposition A.4  The append operator (||) always enforces the No Query-Level Disorder Guarantee.

Proof  The property of output of the || operator:

\[ \forall y^chg, y'^chg \in Y^{chg} : \]
\[ \text{org}(y^chg) \in Y^{new} \land \text{org}(y'^chg) \in Y^{old} \]
\[ \rightarrow j > j' \]

inherently contrasts with possibility of Query-Level Disorder:

\[ \exists y^chg, y'^chg \in Y^{chg} : \]
\[ \text{org}(y^chg) \in Y^{new} \land \text{org}(y'^chg) \in Y^{old} \]
\[ \land j < j' \]

therefore, Immediate, Delayed Drain, and Query-ordered Drain Change guarantee the No Query Level Disorder property, since the all use this operator.

Proofs for other parts can be deduced from the LDD Rule’s proof.

Duplicates

Here we first define two properties which will later be used in the relevant proofs.

- **Disjointness**: An output stream of change policy has the Disjointness property, if its elements are exclusively dependent on input elements before \( x_{\text{change}} \) or elements after it. Thus, following statement holds:

\[ \exists y^chg \in Y^{chg} : \]
\[ \exists x_i, x_{iv} \in X : \]
\[ (x_i, x_{iv} \in \text{depends}_{\text{new}}(\text{org}(y^chg))) \]
\[ \lor x_i, x_{iv} \in \text{depends}_{\text{old}}(\text{org}(y^chg)) \]
\[ \land i < \text{change} < iv \]
• **Exclusive Contribution**: An input stream has this property, if it has an particular element -say \(x_e\) for which all of its elements arriving before \(x_e\) only contribute to output elements taken from the old query’s output and also those arriving after \(x_e\) only contribute to output elements taken from the new query’s output.

\[
\exists x_i \in X : \exists y_{j_i}^{ch}, y_{j_i}^{ch} \in Y^{ch} : \\
\text{org}(y_{j_i}^{ch}) \in Y^{old} \land \text{org}(y_{j_i}^{ch}) \in Y^{new} \\
\land \text{org}(y_{j_i}^{ch}) \in \text{contributes}_{old}(x_i) \\
\land \text{org}(y_{j_i}^{ch}) \in \text{contributes}_{new}(x_i)
\]

Notice that each of these properties implies that no duplicates are caused by the corresponding change policy.

**Proposition A.5** *Immediate Change guarantees No-Duplicate.*

**Proof** Since it implies the Disjointness property on the output stream and it is guarantees no-duplicate behavior.

**Proposition A.6** *Delayed Drain Change guarantees No-Duplicate.*

**Proof** Since it assures the Exclusive Contribution usage of input stream, hence it is free of duplicates. Notice that in Delayed Drain change, the values of \(e\) (in \(x_e\)) is same as the index of \(x_{fstart}\).

**Proposition A.7** *Drain Change guarantees No-Duplicate.*

**Proof** We show that the difference between dependency set of any arbitrary element of \(Y^{dch}\) taken from the new query’s output and any other arbitrary element of \(Y^{dch}\) taken from the old query’s output is always non-empty.

\[
\forall y_{j_i}^{dch}, y_{j_i}^{dch} \in Y^{dch} : \\
\text{org}(y_{j_i}^{dch}) \in Y^{old} \land \text{org}(y_{j_i}^{dch}) \in Y^{new} \\
\exists x_i \in \text{depends}_{old}(\text{org}(y_{j_i}^{dch})) : i < dch \\
\land \\
\forall x_i \in \text{depends}_{new}(\text{org}(y_{j_i}^{dch})) : i > dch \\
\Rightarrow \text{depends}_{old}(\text{org}(y_{j_i}^{dch})) - \text{depends}_{new}(\text{org}(y_{j_i}^{dch})) \neq \emptyset
\]

This clearly shows that there are no two items in the output of Drain Change with the exact same dependency set. Hence, there would be no duplicate introduced by the Drain Change in the output stream.

**Liveness Guarantees**

**Termination**

Termination of the old query is driven by two parameters: 1) the type of stop control element used to implement change, and 2) fixity of its effective kick-in position. Immediate Stop terminates a query immediately whereas Drain Stop lingers the termination while waiting for the “right data”(e.g. closing condition
for a window operator) to arrive, and thus it is data dependent and statically incomputable. To generalize, change policies (e.g. Immediate Change) using immediate stop to stop the old query guarantee termination whereas others (e.g. DDChange, SDChange, QDChange) cannot guarantee termination.

Progress

Progress of the new query depends on whether the start position of the new query is statically determined or not. In case of Immediate and Stream-ordered Drain Changes, start element is statically determined to be at the position of the change control element itself, whereas for Delayed Drain Change the new query’s progress is dependent on the old query’s termination, which is not always guaranteed.

LDD Rule

Proposition A.8 The LDD rule states that in the general case, a Change policy can ensure at most two out of three of the following guarantees: No Loss, No Stream-level Disorder, No Query-level Disorder.

Proof In order to prove this rule, we split it into three separate cases; however, without losing the generality, throughout this set of proofs, we focus on the following example (formal depiction of Figure 8, it basically means the last window of the old query fully encompasses one of the new query’s windows):

\[
\exists y_{j}^{chg}, y_{j}^{chg} \in Y^{chg} : \\
\text{org}(y_{j}^{chg}) \in Y^{new}, \text{org}(y_{j}^{chg}) \in Y^{old} \\
\wedge \\
\min(\text{indexOf}(\text{depends}_{new}(\text{org}(y_{j}^{chg})))) < \\
\min(\text{indexOf}(\text{depends}_{old}(\text{org}(y_{j}^{chg})))) \\
\wedge \\
\max(\text{indexOf}(\text{depends}_{new}(\text{org}(y_{j}^{chg})))) > \\
\max(\text{indexOf}(\text{depends}_{old}(\text{org}(y_{j}^{chg}))))
\]

- Loss and Query-Level Disorder are guaranteed: by doing so, \( y_{j}^{chg} \) appears prior to \( y_{j}^{chg} \) and that means, Stream Level Disorder is violated
- Loss and Stream-Level Disorder are guaranteed: analogous to the previous one
- Query and Stream-Level Disorder are guaranteed: this means either \( y_{j}^{chg} \) or \( y_{j}^{chg} \) should be excluded from the output stream, resulting in Loss in both ways.

LT Rule

Proposition A.9 The LT rule indicates that No Lossless Change policy can guarantee Termination.

Proof This proof is straightforward: In order for a Change policy to be Lossless, it has to use the Drain Stop control element, and this type of Stop element does not guarantee the termination by its very definition.
B  Operation Composition Proofs

Propositions

\[ Y_{FS} = Q(X_{FS}) \]
\[ Y_{FS} = OP2(OP1(X_{FS})) \]
\[ Y_{CS} = Q(X_{CS}) \]
\[ Y_{CS} \neq OP2(OP1(X_{CS})) \]
\[ Y_{WS} = Q(X_{WS}) \]
\[ Y_{WS} \neq OP2(OP1(X_{WS})) \]
\[ Y_{IS} = Q(X_{IS}) \]
\[ Y_{IS} = OP2(OP1(X_{IS})) \]
\[ Y_{DS} = Q(X_{DS}) \]
\[ Y_{DS} \neq OP2(OP1(X_{DS})) \]

Here we provide the proofs for representative cases: IStop and DStop. Other proofs are analogous. Notice that in the following proofs we take advantage of the following equivalence:

\[ \text{dep}(\{y_j, y'_j\}) = \text{dep}(y_j) \cup \text{dep}(y'_j) \]

Operator Composition Behavior on Immediate Stop

\[ Y_{IS} = Q(X_{IS}) \]
\[ Y_{IS} = OP2(OP1(X_{IS})) \]

Proof

\[ Y_{SS} = \{y_j| \forall x_i \in \text{dep}_Q(y_j) \ i < istop\} \]
\[ = \{y_j| \forall x_i \in \text{dep}_{OP1}(\text{dep}_{OP2}(y_j)) \ i < istop\} \]

assuming \[ \text{dep}_{OP2}(y_j) = \{z_{k_1}, \ldots, z_{k_n}\} \]
\[ = \{y_j| \forall x_i \in \text{dep}_{OP1}(\{z_{k_1}, \ldots, z_{k_n}\}) \ i < istop\} \]
\[ = \{y_j| \forall x_i \in \text{dep}_{OP1}(z_{k_1}) \cup \ldots \cup \text{dep}_{OP1}(z_{k_n}) \ i < istop\} \]
\[ = \{y_j| (\forall x_i \in \text{dep}_{OP1}(z_{k_1})) \land \ldots \land (\forall x_i \in \text{dep}_{OP1}(z_{k_n})) \ i < istop\} \]
\[ = \{OP2(\{z_{k_1}, \ldots, z_{k_n}\})| \forall x_i \in \text{dep}_{OP1}(z_{k_1}) \land \ldots \land (\forall x_i \in \text{dep}_{OP1}(z_{k_n})) \]
\[ \ i < istop\} \]
\[ = \{OP2(\{z_{k_1}, \ldots, z_{k_n}\})| z_{k_1} \in Z_{IS} \land \ldots \land z_{k_n} \in Z_{IS} \ i < istop\} \]
\[ = OP2(Z_{IS}) \]
Operator Composition Behavior on Drain Stop

\[ Y_{DS} = Q(X_{DS}) \]
\[ Y_{DS} \neq OP2(OP1(X_{DS})) \]

Proof

\[ Y_{DS} = \{ y_j | \exists x_i \in \text{dep}_{Q}(y_j) \text{ and } i < dstop \} \]
\[ = \{ y_j | \exists x_i \in \text{dep}_{OP1}(\text{dep}_{OP2}(y_j)) \text{ and } i < dstop \} \]

assuming \( \text{dep}_{OP2}(y_j) = \{ z_{k_1}, ..., z_{k_n} \} \)

\[ = \{ y_j | \exists x_i \in \text{dep}_{OP1}(\{ z_{k_1}, ..., z_{k_n} \}) \text{ and } i < dstop \} \]
\[ = \{ y_j | \exists x_i \in \text{dep}_{OP1}(z_{k_1}) \cup ... \cup \text{dep}_{OP1}(z_{k_n}) \text{ and } i < dstop \} \]
\[ = \{ y_j | (\exists x_i \in \text{dep}_{OP1}(z_{k_1})) \lor ... \lor (\exists x_i \in \text{dep}_{OP1}(z_{k_n})) \text{ and } i < dstop \} \]
\[ = \{ \text{OP2}(\{ z_{k_1}, ..., z_{k_n} \}) | (\exists x_i \in \text{dep}_{OP1}(z_{k_1})) \lor ... \lor (\exists x_i \in \text{dep}_{OP1}(z_{k_n})) \text{ and } i < dstop \} \]

Notice that here we can possibly have \( z_{k_m} \) for which, \( \forall x_i \in \text{dep}_{OP1}(z_{k_m}) \text{ and } i > dstop \)

\[ = \{ \text{OP2}(\{ z_{k_1}, ..., z_{k_n} \}) | (z_{k_1} \in (Z_{IS} \cup \Delta) \land ... \land z_{k_n} \in (Z_{IS} \cup \Delta)) \text{ and } i < dstop \} \]
\[ = \text{OP2}(Z_{IS} \cup \Delta) \]

Special Cases

If \( \text{OP2} \) is stateless (meaning \( |\text{dep}_{OP2}(y_j)| = 1 \)) then

\[ Y_{DS} = Q(X_{DS}) \]
\[ Y_{DS} = \text{OP2}(\text{OP1}(X_{DS})) \]

generalizable to as many as stateless operators following the last statefull operator in the query network

C Modeling Pause-Resume Using Our Framework and Methodology

Another important operation on continuous queries, is to Pause-Resume a query. Resource shortage is one of the most common reasons that forces the system to temporarily pause (suspend) execution of one or more of its running continuous queries. In such cases, as soon as the shortage is resolved, the paused queries should be resumed.

Here, following our methodology, we first precisely define what a Pause-Resume operation is, then after introducing the Correctness criteria for this operation, we enumerate the possible variations of its execution and compare them in terms of guarantees they provide.
Formal Definition of Pause-Resume

In *Pause-Resume*, there is a query \( Q \) which is applied to a single input stream \( X \) and produces an output stream \( Y \). The same input stream \( X \) with a *Pause-Resume* control element, would produce another output stream which we refer to as \( Y^{pr} \) in which \( pr \) is short for *Pause-Resume*.

Intuitively, a *Pause-Resume* control element can be translated into a combination of *Stop* directly followed by a *Start* targeting the same query. Notice that although there is no positional difference between the *Start* and *Stop* control elements, the time difference between applying them can be arbitrary.

Correctness of Pause-Resume

Similar to the Correctness criteria of *Query Modification*, one can define Safety and Liveness criteria for *Pause-Resume*.

C.0.1 Safety Guarantees

In case of *Pause-Resume*, the Reference Output Stream, is the original output stream. More formally

\[ Y^{ref} = Y \]

Having defined the \( Y^{ref} \), here comes the brief definition of Safety guarantees for *Pause-Resume*

- *Loss*: It follows the same formulas in 4.2 but with the new Reference Output Stream we defined above.
- *Duplicate*: Same definition is valid here too.
- *Disorder*: Is defined if the following statement does not hold

\[ \forall j \in \mathbb{N}, y_j \in Y, y_j^{pr} \in Y^{pr} : y = y^{pr} \]

C.0.2 Liveness Guarantees

Similarly, the *Termination* guarantee implies that the paused query will eventually stop producing outputs. Notice that in case of *Pause-Resume* the Progress of the query is always guaranteed.

Pause-Resume Control Elements

Here are the output definitions for all meaningful variations of the *Pause-Resume* control element:

- Pause Resume of type I-W: in which an Immediate Stop is directly followed by a Warm Start. More formally,

\[ Y^{iwpr} = Y^{istop}\|Y^{wstart} \]

this variation can guarantee all desired properties; however, since it uses Warm Start, its actual implementation incurs persisting the unfinished output items’ states.

---

5 Although here, duplicates will have the identical *tuple* values too.
• Pause Resume of type I-C: in which an Immediate Stop is directly followed by a Cold Start. More formally,

\[ Y_{icpr} = Y_{istop} Y_{cstart} \]

this variation can be lossy.

• Pause Resume of type D-W: in which a Drain Stop is directly followed by a Warm Start. More formally

\[ Y_{dwpr} = Y_{dstop} Y_{wstart} \]

this variation can generate duplicates. Moreover, it does not guarantee termination of the pause step.

• Pause Resume of type D-C: in which an Immediate Stop is directly followed by a Cold Start. More formally,

\[ Y_{dcpr} = Y_{dstop} Y_{cstart} \]

this variation, is similar to the I-W variation although it can be cheaper in terms of persistence overhead. Moreover, it does not guarantee Termination.

It should be noted that here we excluded that the Fresh Start Control Element from the set of eligible Basic Control Elements. We did this because using FStart for resume restarts the mapping functions which always causes loss and non-identical output streams. However, using this variant can be considered under circumstances in which loss is tolerated.

References


