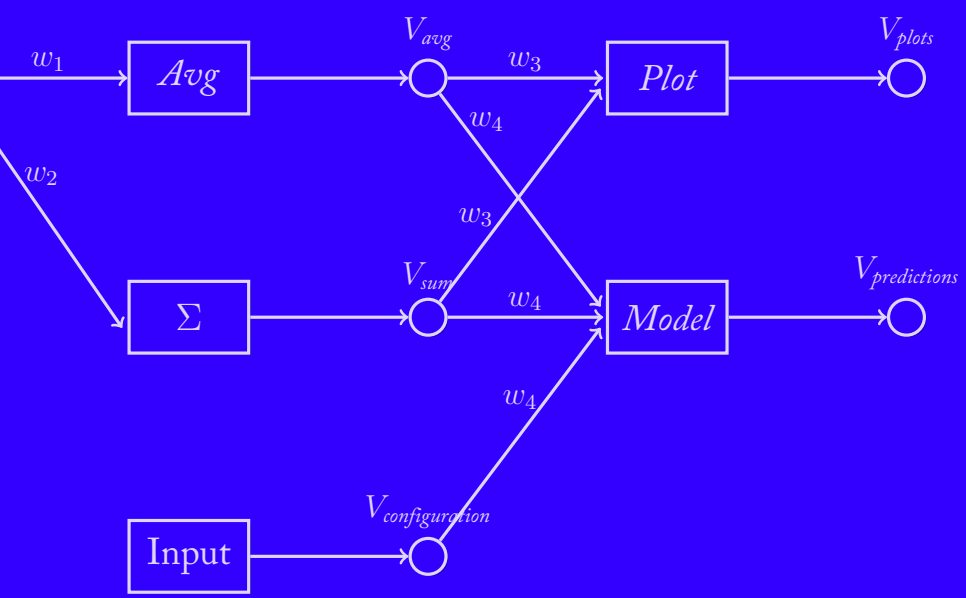
Streaming Workflow Transformation



Streaming Workflow Transformation

Master's Thesis in Computer Science

Tjalling van der Wal

February 2011

Supervisor:
Andreas Wombacher

UNIVERSITY OF TWENTE.

Abstract

This thesis presents 1. a formal model for streaming workflows adapted for transformation and 2. transformation rules for streaming workflows defined according to that formal model. The validity of the transformation rules is demonstrated by formally proofing equivalence. The validity of the formal model is demonstrated by the fact that valid transformation rules can be defined.

Transformation of streaming workflows is the first step towards automatic optimization of streaming workflows. By providing a formal model and transformation rules, this thesis demonstrates that it is possible to build a self-optimizing Streaming Workflow System.

Contents

List of Figures

Chapter One

Introduction

This thesis presents a formal model and transformation rules for streaming workflows. In this chapter the motivation, objectives and research questions are introduced. First, this chapter introduces an example of a streaming workflow that will be used throughout this thesis.

1.1 Monitoring a River

An environmental scientist wants to devise a system to detect ground water influxes that indicate weaknesses in dikes along a river. Part of his research involves deploying sensors to measure water temperature, flow and saline levels. The equipment used is advanced and capable of transmitting the measurements wirelessly to the research centre, sometimes as frequent as every second when required.

To detect potential natural hazards as soon as possible, a computer system at the research centre continuously processes the fresh data and compares it to configured thresholds and predicted values. To reduce false positives, the system utilizes sliding averages and historic data.

The processing done by the system has been defined and configured by means of a workflow specification. This workflow is called a streaming workflow, because it is used to process streams. In a stream of data, the individual data items are related and the order of the data has semantic significance. (As opposed to business process workflows, in which the work items are similar but independent.)

The system also processes a streaming workflow defined by a second scientist. By comparing each week of data with the same week in preceding years, the scientist hopes to prove his own hypothesis on the effects of climate change on the observed river. Although this workflow is less frequently executed, it is nonetheless streaming.

Both workflows independently calculate averages over the same measurements. Luckily the system is smart enough not to calculate the averages twice.

1.2 Sensor Data Problems and Challenges

The system from the example does not exist (yet). Section 1.3 proposes such a system. This section provides the motivation for building it.

Problems The utilization of measurements is often limited to the person or team that has collected them. That is a shame because collecting measurements is expensive. The low utilization of measurements is caused by the following problems:

- Data is not stored centrally. Measurements are stored on personal desktops: not available to other researchers or teams; they may not even know about it. After a researcher leaves the project, the measurements are not accessible any more.
- Data integration is hard. Conventions differ and detail is missing. The quantity T could be either ground, soil or air temperature. This makes it hard to combine and compare measurements from different projects.
- Data governance is difficult. After a researcher leaves or after a project is finished, nobody knows how measurements were collected and how they were processed. How frequent did the sensor 'sense'? Is the data raw or has it been processed?
- The processing is manual and ad-hoc. Everyone uses their own favorite tool, like Matlab, Excel, SPSS or custom scripts. This exacerbates the other problems.

Each of these problems makes it harder to collect and process sensor data in such a way that measurements can be easily shared and reused.

Challenges Apart from the problems above there are recent developments that could provide new opportunities but could also increase the existing problems in the future.

- Although sensor deployments are expensive, individual sensors have become cheaper. This means more sensors can be deployed at the same cost and more data can be collected. But this also means that the amount of data to manage becomes larger.
- Another challenge is included in the example in the previous section: streaming data. Instead of arriving periodically in batches, the data arrives as a continuous stream. This means real-time monitoring is possible, provided a more robust way of managing data is developed.

1.3 Streaming Workflow System

The solution to these problems and challenges is to build a system such as already described in Section 1.1. The system will process incoming sensor data according to processing instructions specified by the user by means of streaming workflows: it is a Streaming Workflow System.

In 'Composable data processing in environmental science — a process view.' [22] the same problems have been described and a similar system is suggested.

Scope of the System Although the example in Section 1.1 and some of the problems in Section 1.2 are specific to environmental sciences, the ambition is to build a generic system that can be used in many more situations. Examples include administrative systems, monitoring systems and control systems, in accounting and security domains. Considering that a cash register can be seen as a sensor that observes sales, it is logical not to limit the Streaming Workflow System to environmental science applications.

Functions and Requirements for the System A Streaming Workflow System must perform the following functions:

1. The system enforces formalization of data processing by means of streaming workflow specifications. Formalization is needed for the system to be able to perform optimization. Additionally, formalization provides governance.

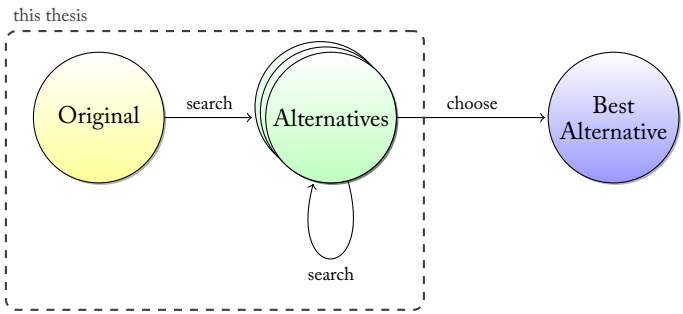


Figure 1.1: Optimization

2. The system shall manage data storage and distribution. If the data is centrally available, it can be shared and reused more easily.

3. The system shall optimize the workflow specifications submitted by users. Optimization shall be done both within and between workflows. Criteria for optimization include computational and storage cost. Optimization is needed to let the system scale in terms of users, data volume and data rate.

These three requirements are sufficient to tackle the problems and challenges from Section 1.2 and to make the example from Section 1.1 reality. This thesis contributes to the third requirement: optimization of streaming workflows.

1.4 Optimization

Optimization is a requirement for the Streaming Workflow System. But what is optimization? And what is needed to build an optimizer?

Optimization Optimization can be described as searching for an alternative that is better than the original. Figure 1.1 shows the basic process. Starting from an original, an optimizer searches for alternatives. The optimizer continues by searching for more alternatives based

on alternatives that have been found earlier. In the end, from all the alternatives found, the best one is chosen.

As example, consider the algebraic formula $x = (a + b) * (a + c)$. By applying algebraic rules the following alternatives can be found: $x = (a + c) * (a + b)$ and $x = a * (b + c)$. The second alternative is better because it requires less computation to find x.

Searching for Alternatives The search for alternatives has two prerequisites: 1. a formal way to describe both the original and the alternatives and 2. a formal definition to tell whether an alternative is valid. In the algebraic example the formula itself is a formal description and any formula that evaluates to the same value is a valid alternative.

In the workflow example from Section 1.1, workflows are described by a specification. Searching for alternatives is done by applying transformation rules to a streaming workflow specification to create alternatives, just like algebraic rules are used to create alternative formulae. The contribution of this thesis is defining transformation rules for streaming workflows.

Choosing an Alternative Which alternative is best depends on the context. In case of the algebraic example, the original formula is the best when trying to solve for $x = 0$, but the alternative $x = a * (b + c)$ is the best when computing the value. The selection of the 'best' alternative streaming workflow specification is outside the scope of this thesis.

1.5 Objectives

This thesis tries to contribute to the idea of a Streaming Workflow System by showing the feasibility of building such a system, in particular the feasibility of creating an optimizer for streaming workflows. The objectives of this thesis are:

1. **Validate** the idea of a Streaming Workflow System by showing it is theoretically possible to perform automatic optimization of user-defined streaming workflow specifications.

It is assumed that optimization is possible if transformation is possible. Therefore this thesis is limited to streaming workflow transformation.

2. **Define** a formal model for streaming workflows with transformation in mind. An existing formal model for streaming workflows is used as a starting point.

3. **Define** transformation rules for both generic and specific activities, and for a reasonable set of common activities (including relational).

1.6 Research Questions

To fulfill the objectives, this thesis will answer the following research questions:

1. How must the (existing) formal model be adapted to allow for transformation?

2. What is a suitable definition of equivalence between workflows?

3. Can transformation rules be defined for common types of activities?

    a) Can rules be defined for generic situations?

    b) Can rules be defined for common (relational) activities?

    c) Can rules be defined for aggregates and join activities?

By answering these research questions, the second and third objective from Section 1.5 are fulfilled. The first objective is indirectly fulfilled by fulfillment of the second and third objective.

1.7 Thesis Structure

The structure of this thesis follows the research questions. In Chapter 2 the formal model is described and in Chapter 3 equivalence of workflows is defined. These two chapters are prerequisite for the next chapters.

In Chapters 4, 5 and 6 transformation rules valid within the formal model are defined. These chapters can be read selectively, but in particular the Copycat Rule (Section 4.2) should be read because for this rule a full proof is provided that uses and illustrates most concepts from the formal model.

Section 4.5 presents an example that shows how two independent but similar streaming workflows can share computation through the application of multiple transformation rules. Section 4.6 shows how transformation rules can be used to optimize the 'Monitoring a River' example as the workflow evolves during a research project.

In Chapter 7 related work with regard to streaming workflows and continuous querying is discussed. In Chapter 8 the results of this research are evaluated against the objectives and in Section 8.4 a suggested approach for actually building an optimizer for a Streaming Workflow System is presented.

Chapter Two

Formal Model

Before a streaming workflow can be transformed, a formal description of the workflow is required. The description must be formal because 1. the description must be manipulatable for a computer system and 2. it must be possible to proof the equivalence between workflows. This chapter introduces the formal model used to describe streaming workflows.

In the first section the 'Monitoring a River' case from the introduction is used as an example to introduce the core concepts of the formal model. In Section 2.2 the behavior of a streaming workflow is formalized. In Section 2.3 assumptions on the system's runtime are listed. These are needed to make the model work and to complete the proofs of transformation rules. Section 2.4 discusses design choices that have influenced the formal model.

Origins The formal model presented in this chapter is based on the formal model described by Wombacher in 'Data Workflow — A Unified Time Controlled E-Science Processing Model' [23]. Compared to the original model, this new model trades expressive power for the predictability needed to do transformations.

It is important to note that although the formal model is less expressive, it does not mean that a Streaming Workflow System would be less powerful. More powerful constructs can be offered by the system, with the limitation that not all of those can be transformed and optimized using the formal model presented here.

2.1 Monitoring a River with a Streaming Workflow

Section 1.1 introduced the example of a scientist who tries to protect dikes by detecting groundwater influxes. The data processing needed to detect influxes has been specified by the scientist by means of a streaming workflow. This section explains how a streaming workflow works

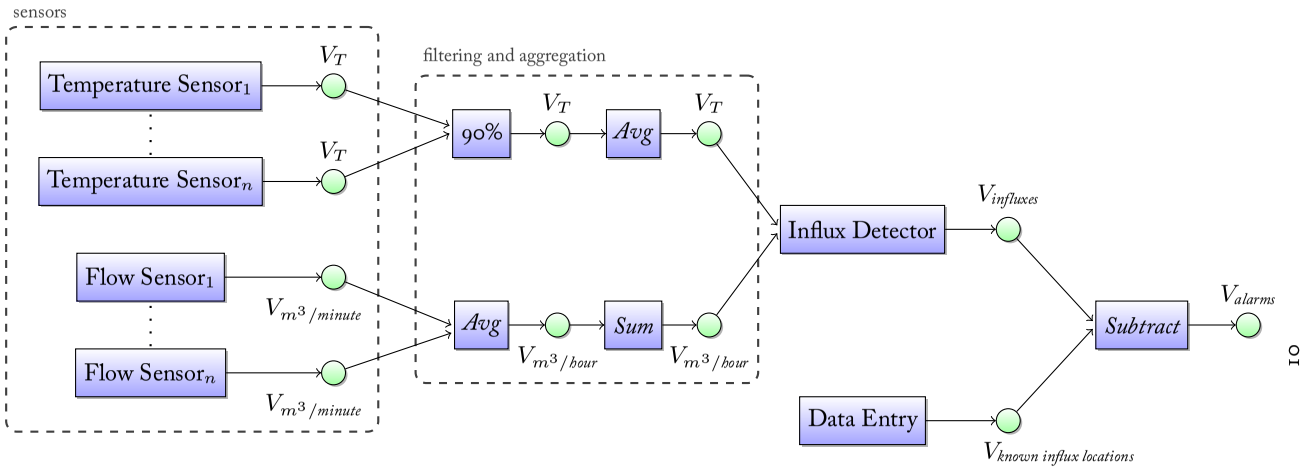


Figure 2.1: Monitoring a River to Detect Groundwater Influxes

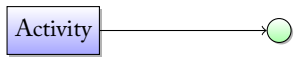
Activity	Output (content of view)	Purpose
		
Temperature Sensor	T in °C at location x at time t	Observe river
Flow Sensor	Flow in m^3/s at location x in time interval t	Observe river
90%	T in °C at location x at time t	Eliminate values outside 90% percentile to address sensor errors
Avg(T)	Average T in °C at time t over last hour	Calculate average temperature in river
Avg(Flow)	Average flow in m^3/s in time interval t	Use average as 'weighted majority vote' to reduce impact of sensor error
Sum(Flow)	Total flow in $m^3/hour$ at time t	Calculate total flow in last hour
Influx Detector	Detected influxes at location x during time interval t	Does what this workflow was created for; all the other activities exist to feed data to this activity.
Data Entry	List of known influx locations	Manually record locations with known issues or special circumstances.
Subtract	List of alarms without false positives	Eliminate known influx locations from list of influxes.

Table 2.1: Description of Workflow Components in Figure 2.1

using this example.

To detect influxes, the scientist has written an algorithm that looks for anomalies in the combination of water temperature and water volume. Before sensor data can be fed to this algorithm, the data has to be filtered to address sensor errors and has to be aggregated. After the influx detection, known locations have to be filtered out to produce a list of locations on the dike that need inspection. So the workflow has the following steps: 1. record the data, 2. filter the data, 3. aggregate the data 4. execute detection algorithm, 5. record known influx locations and 6. eliminate false alarms.

Figure 2.1 shows the streaming workflow created by the scientist. Table 2.1 details the purpose of each individual component in the workflow. Together these components perform the six required steps.

The workflow starts with the sensors deployed in the river; periodically they send data to the research centre. In the formal model everything that produces or processes data is an **activity**. The activities representing the sensors are shown on the left of the workflow as blue boxes. Associated with each activity is a **view** that stores all data produced by an activity. In the workflow, views are depicted as small green circles.

The recording of sensor data is represented by the sensor activities and their corresponding views. The second step to perform is to filter the data to reduce the impact of sensor error. For temperature data, a 90% activity periodically reads the data from all the sensor views and produces a new stream from which outliers have been removed. For the flow data, the average over all sensors is calculated by an *Avg* activity. The output from both activities is again stored in views.

The third step, aggregation, is similar to the filtering: an activity reads the data from a view, applies the required computation and produces a stream which is stored in a new view. This results in two views: one with a sliding average temperature and one with a sliding total flow volume.

The custom influx detection algorithm is the fourth step and is also an activity. It reads the aggregated data and produces a stream of locations with an anomaly in the combination of water temperature and water volume.

As a sixth and final step the list of locations is filtered by means of a manually maintained list that is produced by step five.

Two phrases in the description above have been deliberately left vague: 'periodically' and 'reads the data'. This raises two questions: *when does an activity execute?* and *which data is used for an execution?*. The next section will answer those questions.

2.2 Behavior of Activities

This section describes the behavior of activities. Activities are the active components of a streaming workflow, as opposed to views that just store data and thus have no behavior. The behavior of activities is defined by two components: 1. *when* to execute and 2. *what input data* to use. Both are specified using window sequences.

Definition 1: Window Sequence A window sequence is a regular and predictable sequence of windows on a view. Each window represents a single activation of an activity and the selected data from the view to use for that activation. A window sequence is either defined in terms of time or in terms of arriving tuples.

When an activity must be activated (or executed) is determined by an arithmetic sequence of timestamps or tuple-ids. Elements of such a sequence are called reference points.

Definition 2: Reference Points The reference points of an activity are an arithmetic sequence generated by $[\mathit{epoch} + n * \mathit{delta} \mid n \in N_0]$. For time-based activities *epoch* and *delta* must both be valid time units. For tuple-based activities both must be tuple counts.

An activity can have a different window sequence for each input relation, but they all share the same reference points.

For each reference point the workflow system activates the activity. The system feeds the activity with data selected by the current window and the activity appends zero or more output tuples to the output view.

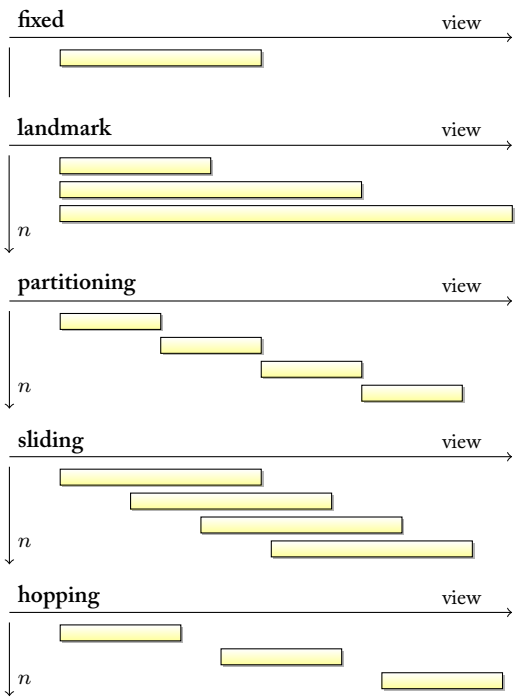


Figure 2.2: Types of Window Sequences. The horizontal axis represents the data in the view and the bars are window instances, indicating the selected data. The vertical axis displays the n used to generate the reference points sequence $[\mathit{epoch} + n * \mathit{delta} \mid n \in N_0]$.

What data must be selected from input views is defined relative to the reference point of the current activation. Three basic types of window sequences are supported.

Definition 3: Regular Window Sequence For regular windows, the lower and upper bound of the data interval selected from a view is defined relative to the reference point. The interval definition is: $\langle$*reference point* $-$ (*ws* $+$ *offset*) . . . *referencepoint* $-$ *offset*]. *ws* is the *window size* of the window and *offset* allows to specify the distance of the window to the reference point. Figure 2.3 visualizes this.

There are exactly three subtypes of regular window sequences based on the relation between the window size (*ws*) and the distance that the reference point shifts between each window (*delta*). These three subtypes are partitioning (*ws* $=$ *delta*), sliding (*ws* $>$ *delta*) and hopping (*ws* $<$ *delta*). Figure 2.2 shows them.

A partitioning window sequence has two specific properties: all windows are disjoint: $\forall i, j \; : \; w_i \cap w_j \; = \; \emptyset$ (mutually exclusive) and all windows together cover the entire view: $\bigcap w_i \; = \; V$ (collectively exhaustive). A hopping window sequence ignores part of the data in the input view.

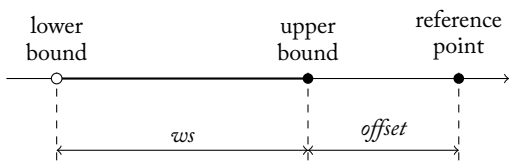


Figure 2.3: Regular Window Definition

Definition 4: Landmark Window Sequence For landmark windows the lower bound is fixed and the upper bound is defined in the same way as for regular windows. The interval definition is: $\langle$*landmark* . . . *reference point* $-$ *offset*].

Definition 5: Fixed Windows For fixed windows the interval is defined with fixed lower and upper bounds. Fixed windows are used for one-time queries and to incorporate historic data.

Definition 6: Window (~ Instance) A single window from a window sequence; obtained by substitution of the reference point of the current activation into the window sequence definition. It serves as the data selection interval for the current activation. The lower bound of a window is exclusive and the upper bound is inclusive.

Definition 7: Standard Tuple-based Window Sequence For tuple-based activities a special window sequence is defined. The standard tuple based window sequence w_e for event based activities is defined as $\mathit{delta} = 1$, $ws = 1$, $\mathit{offset} = 0$.

When activated, an activity is not just called with a bag of input tuples. The workflow system also reports the interval used and all the reference point sequence and window sequence information. *Union* and *Join* activities need this information to distinguish between fresh tuples and tuples they 'have seen before'.

Knowing the specification of the window of the current activation, an activity can calculate the bounds of the previous window. The latter can be used to determine which tuples 'have been seen before' and which tuples are fresh.

Definition 8: Fresh Tuples A fresh tuple is a tuple that was not included in an earlier window instance. Figure 2.4 shows windows from a sliding window sequence with fresh tuples marked orange. In partitioning and hopping windows, all tuples are fresh.

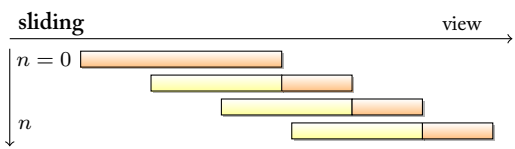


Figure 2.4: Fresh Tuples in a Sliding Window Sequence. Fresh tuples shown in *orange*.

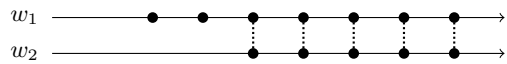


Figure 2.5: Alignment of Window Sequences

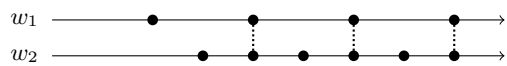


Figure 2.6: Subalignment of Window Sequences

An important property of reference points and window sequences is alignment. This property is used in proofs in Chapter 4.

Definition 9: Window Sequence Alignment Two window sequences w_1 and w_2 align *iff* $\mathit{delta}(w_1) = \mathit{delta}(w_2)$ and $\exists n : \mathit{epoch}(w_2) = \mathit{epoch}(w_1) + n * \mathit{delta}(w_1)$.

Because n is allowed to be negative, alignment is an equivalence relation: it is reflexive, symmetrical and transitive. Reference to alignment of activities is shorthand for alignment of their respective window sequences.

Definition 10: Window Sequence Subalignment Two window sequences w_1 and w_2 subalign *iff* $\exists k \in \mathbb{N}_1 : k * \mathit{delta}(w_1) = \mathit{delta}(w_2)$ and $\exists n \in \mathbb{N} : \mathit{epoch}(w_2) = \mathit{epoch}(w_1) + n * \mathit{delta}(w_1)$.

Window subsequence alignment captures the case where one window sequence is a subsequence of another window sequence. For example, an average over a period of a week that is calculated each week, is a subsequence of an average over a seven day period that is calculated each day.

2.3 Assumptions on the System Runtime

To make the formal model function, several assumptions must be made on the Streaming Workflow System runtime.

Assumption I: The System is Infinitely Fast. The model has a formal and mathematical nature. Execution time is not something that concerns the model. Therefore the model assumes that execution of activities is fast; so fast that execution takes zero time. This assumption has a very nice effect on the notion of equivalence: since execution is infinitely fast, intermediate states of the system will not be observable and will thus not affect equivalence.

Assumption II: The System Ensures Correct Execution Order. When more than one activity is enabled for the same reference point, the runtime of the Streaming Workflow System will ensure that an execution order is chosen that respects the input-output dependencies between the activities. When an activity is executed it is guaranteed that all input views are complete for all reference points less or equal the current reference point.

The system may not be able to fulfill this assumption when the workflow specification contains cycles. This thesis ignores cyclic subgraphs. Cyclic subgraphs must be treated as a single non-transformable activity.

Assumption III: The Initial System is Empty. The workflow model does not allow orphaned views. As a result any data in the system must have been added to the system through an activity. This includes historic data and (configuration) data that is added manually. This assumption is needed to proof the correctness of transformations.

2.4 Design Choices

This section details a number of choices made while designing the formal model. Unlike the assumptions from the previous section, these choices are not needed to make the formal model function. They do however have had their impact on the formal model.

Design Choice I: Streaming and Querying are the Same. Wombacher [23] makes a distinction between a streaming mode of operation and a special query mode. This model chooses to treat them the same. To illustrate this in a more technical manner: Suppose the system is like a functional programming language and the sensor data is stored in lists. To process the data, you need mapping and folding functions. But: these functions can be applied to both finite and infinite lists and also to lists that are still being generated. For the functions there is no difference between stream processing fresh data and batch processing historic data. Likewise the streaming workflow model does not distinguish streaming and querying modes.

The reason for this choice is that it allows the formal model to be agnostic of the current time; *epoch* and other time variables are not limited to *Now* and the future but can also be in the past.

It is personal belief that this makes the model cleaner and less complex but does not needlessly complicate the implementation of the system runtime.

Design Choice II: Triggers are Ugly. In [23], triggers are used to determine when activities should be executed. Triggers are not declarative and their semantics are hard to capture, especially the fact '1 week' does not mean the same as '7 days'. Also a trigger alone can not be manipulated by transformations because it does not describe what data an activity needs for an activation. Therefore in this model triggers have been replaced by window sequences.

Design Choice III: Workflows are Immutable. Once submitted to the system by the user, a workflow is immutable. When a workflow needs to be updated, the user must submit a new workflow to the system. Whenever is said that a workflow is updated, it means that a new workflow derived from the original is added to the system.

The rationale for immutable workflows is data governance: because workflows cannot change, it is always possible to tell how a certain tuple was generated.

The implication of immutable workflows is that many workflows exist in the system that are only slightly different from each other, often in trivial ways. Therefore there is much more opportunity for shared computation than apparent.

Design Choice IV: All Activities are Stream Deterministic. Wombacher [23] distinguishes deterministic and non-deterministic activities. In this thesis, this distinction is ignored and all activities are considered to be stream deterministic: given the same data stream, a just initialized instance of an activity always produces the same output stream. Whether an activity carries internal state across subsequent activations is irrelevant for this type of determinism.

The kind of determinism described in [23] is much stronger as it is defined in terms of individual activations, theoretically allowing out-of-order execution. This sort of optimizations is outside the scope of this thesis.

Stream determinism does require that the output of activities solely depends on the current and previous inputs. The results should not depend on (in particular) the actual execution time. This gives the system the flexibility to delay execution, but also means that transformations can cause earlier execution of activities without breaking equivalence.

Sensors and data entry activities in a workflow are not stream deterministic. This is not a problem because these activities are the leafs of the workflow graph and can not be eliminated anyway for that reason.

2.5 Position in the System

This section describes the position of the formal model within a Streaming Workflow System. Describing what the model *is* and, in particular, what it *is not*, has helped me considerably to think about transformation rules in isolation.

The Model is Not a User Interface. The formal model is not intended as a user interface. A real system will provide a higher level of abstraction. Because the model is not a user interface, seemingly arbitrary and counter-intuitive limitations and assumptions can be made about workflows.

The Model is Not an Execution Plan. The Streaming Workflow System does not actually execute a streaming workflow according to the specification. Different algorithms and data structures may be used depending on activity configuration, expected or observed data rates and workflow structure. Also, common subgraphs may be implemented by

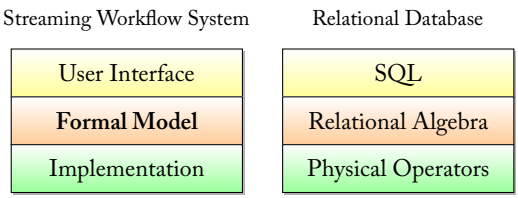


Figure 2.7: Position of the Formal Model in a Streaming Workflow System.

a single algorithm. Because the model is not an execution plan, it is no problem when a transformation rule produces a workflow that seems less efficient.

So What is the Model? The formal model is an abstraction layer that enables the system to transform streaming workflow specifications into *any* shape. The desirability of a shape is no concern in the model.

Figure 2.7 compares the Streaming Workflow System with a relational database management system (RDMS): the model sits between the user interface and the actual implementation of activities, similar to the function of relational algebra.

Chapter Three

Equivalence

In Section 1.4, optimization was summarized as 'searching for an alternative that is better than the original'. One of the questions that arises from that description is: what defines a valid alternative? When is a streaming workflow a valid alternative for another streaming workflow?

A workflow is a valid alternative *iff* it can be interchanged with the original without changing the output of the system. The formal concept for 'valid alternative' or 'interchangeable' is equivalence. Two workflows are valid alternatives for each other when they are equivalent.

Equivalence is denoted by the $\equiv$ operator. In this thesis the notations $=_t$ and $\equiv_t$ are used to state that an equality or equivalence is valid at a certain point in time.

3.1 Monitoring a River the Same Way, but Different

Previous chapters have used the example of a scientist who wants to observe dikes along a river to detect groundwater influxes and his colleague who studies the effects of climate change on the river. What does equivalence mean to them?

To the user of a Streaming Workflow System equivalence means that he gets the results he expected; that the data **apparently** has been processed in the way he has specified the data to be processed. The system must deliver predictable, reproducible and traceable results to guarantee the scientific validity of his research.

Because the user has such specific expectations and noble intentions with the results, the system can not compromise the quality of the results in any way. The only thing the system is allowed to do is replace a workflow by another workflow that is equivalent, but possibly more efficient.

Equivalence simply means that two different workflows deliver the same results. This chapter formalizes this intuitive definition to make it possible to proof that transformation rules preserve equivalence in Chapter 4.

3.2 Equivalence of Activities

This chapter describes equivalence bottom-up, so this section starts with describing equivalence of activities. Intuitively, activities are equivalent when they produce identical output from the same input.

Definition 11: Equivalent Activities Two activities are defined to be equivalent *iff* 1. they run the same code, 2. this code is stream deterministic and 3. they have the same configuration. According to Assumption IV, the second condition is always true.

Note that sensors and other activities at the leafs of the workflow are never equivalent. One could either argue that they do not run code in the sense intended in the definition of equivalent activities or that the location of their deployment is part of their code. Anyhow, the first condition is false.

In transformation rules, activities are considered equivalent *iff* they have the same label. If the labels are different, then the activities are not equivalent. This does not prevent transformation rules from stating that two activities are interchangeable because workflow equivalence will be defined in terms of view equivalence.

3.3 Equivalence of Views

With equivalence defined for activities, the equivalence of views can be defined.

Definition 12: Equivalent Views Two views are defined to be equivalent *iff* at all points in time their content is identical. Content includes all the attributes in the user defined schema plus a subset of the metadata managed by the Streaming Workflow System.

The content of a view for the purpose of equivalence consists of all records (including empty records), with all the attributes in the user defined schema. In addition the *idealized transaction time* (part of the system's metadata) attribute is included; this is the timestamp used by interval predicates to select data from views.

Adding *tuple_id* to the content would make equivalence much harder, but not adding it restricts the kinds of tuple based triggers that can be transformed. This choice requires further consideration.

Equivalence of views can be very simple. The following lemma formalizes a trivial case of equivalence between views. It is a special case of the definition above.

Lemma 1: Trivial Equivalence If two views are 1. produced by equivalent activities, 2. from equivalent input views and 3. with the same window sequence, *then* they are equivalent according to Definition 12.

The first and second conditions must be (indirectly) true for sharing to be possible at all: when doing something different with the data, nothing can be shared and likewise when the data is not from a common source. A lot of transformation rules target situations where the third condition is not true, but where there is some relation between the window sequences that causes a subsumption relation between views.

A simple proof of Lemma 1 can be given by induction over time. Given two views for which all three conditions hold: V_A which holds the output of activity A and V_B which holds the output of activity B. At $t = 0$ both views are empty according to Assumption III, so their content is the same. At time t the content of both views is still the same. At time $t + 1$ both A and B are activated and 1. read the same data interval from equivalent views, 2. perform the same operation on the same data and 3. produce the same output tuples which are appended to V_A and V_B. So at time $t + 1$ the contents of V_A and V_B are still the same. Thus V_A and V_B are equivalent by Definition 12. □

The first rule discussed in Chapter 4 is proofed by showing trivial equivalence. The second rule shows a proof for a situation where the third condition of Lemma 1 is not met.

Strict Equivalence	Weak Equivalence
4.1 Zip	
4.2 Copycat	
	4.3 Shared Loop
4.4 Copy Bypass	4.4 Copy Elimination
	5.1 Union Associativity
	5.2 Selection Pushdown
	5.3 Masquerade
5.4 Single Input Union	
5.4 Idempotent Activities	
	5.4 Collapse and Split Select
	5.4 Intermediate Project
	6.1 Aggregate Split
6.2 Shared Join	

Table 3.1: Transformation Rules by Equivalence Type

3.4 Equivalence of Streaming Workflows

Based on the definition of equivalent views, the definition of equivalent workflows can be given.

Definition 13: Equivalent Workflows (Strict) Two streaming workflows are equivalent *iff* for each view in either workflow, an equivalent view exists in the other.

Many transformation rules introduce views to the workflow to store intermediate results that can be shared and reused. As a result these rules are not valid under the strict definition of workflow equivalence. From a practical (user) viewpoint however additional views are no issue. Therefore a weaker definition of workflow equivalence is added.

Definition 14: Equivalent Workflows (Weak) Given a subset of the views, two streaming workflows are equivalent *iff* for each view in the set, equivalent views exist in both workflows.

This definition weakens the equivalence by restricting the number of views that has to have an equivalent. In practice it will be no problem that the set of equivalent views is restricted because there is an asymmetry as described next.

Definitions 13 and 14 are symmetrical because an equivalence relation has to be symmetrical. In reality there is asymmetry: one workflow will be the specification written by the user and the second workflow is the internal workflow used by the system runtime. The subset used for equivalence is the set of views in the user specification. Additionally, many users are only interested in the end results produced by their workflow. Intermediate views can therefore also be removed from the subset used for equivalence.

Finally, any transformation rule can be rewritten to preserve strict equivalence. This is shown by the Copy Elimination/Bypass Rule in Section 4.4. By performing only the additions specified by the rule and not the deletions, a workflow is created with a maximum set of equivalent views. It is just a little odd to call this a 'transformation'. By extension this means that a Streaming Workflow System can always recreate

views when the subset of views used for equivalence (i.e. user interests) changes.

Comparing Strong and Weak Equivalence Table 3.1 lists the transformation rules presented in the next chapters classified by the type of equivalence. Rules with the same type of equivalence also tend to share other characteristics.

An interesting observation is the fact that rules that adhere to strict equivalence identify duplicate views in a workflow or identify a subsumption relation between views. Application of these rules is always beneficial from a computational or storage perspective. Rules that only preserve weak equivalence try to make duplicate work explicit, allowing the results to be shared and the duplication to be eliminated. Additionally these rules can be used to trade computational cost for storage cost and vice versa.

Independence of Equivalence The definition of equivalence given is independent of both the workflow structure and the execution model. This has the following advantages:

- Transformation rules can structurally change the workflows, as long as the required views are still present.
- The formal model can be changed and implementations can diverge from it. A realistic example would be a filtering activity or a join activity that produces two output views based on two independent predicates.
- Part of a workflow definition can use other semantics and a different execution model. The definition of equivalence allows the formal model to be treated as a subset of the original model defined by [23].

3.5 Local and Global Equivalence

Streaming workflows are composed from multiple activities, each of which has its own semantics. It would be practical if transformation rules can be defined just for basic situations, focussing on a single semantic aspect of a specific type of activity. This is possible because global equivalence is preserved by preserving local equivalence.

Lemma 2: Local Equivalence is Global Equivalence When a subgraph of a workflow is transformed, it is sufficient to show that equivalence is preserved in that subgraph to proof that equivalence is preserved for the full workflow graph. As a consequence, each transformation rule can be discussed in isolation.

To proof Lemma 2, consider transformation T, the set W of all views in a workflow and a subset $S \subseteq W$ of views affected by T. Equivalence is defined under the set of views $E \subseteq W$ (for strict equivalence $E = W$).

For all views $E \cap S$ it is guaranteed that equivalence exists on both sides of the transformation, because T is a valid transformation rule. The views $E \cap (W \setminus S)$ are not affected by the transformation rule, so these do not change equivalence. Since $E = (E \cap S) \cup (E \cap (W \setminus S))$, equivalence under E of the complete workflow is preserved. □

With this lemma it is possible to transform a subgraph of a workflow, without worrying about the effect of the transformation on the whole workflow. It also means that not the entire workflow has to use the semantics defined in the formal model.

Chapter Four

Transformation Rules

In this chapter a series of generic transformation rules is presented. Each of these rules is valid within the formal model from Chapter 2. The rules demonstrate that the formal model can be used to define transformation rules for generic situations (Research Question 3a).

Each rule in this chapter is informally described using graphs and text. The Copycat Rule (Section 4.2) contains a full formal proof; for the other rules an indication is given of how a similar proof could be constructed.

Section 4.1 starts with a simple rule. In Section 4.5 a workflow is transformed using all the rules presented in this chapter and in Section 4.6 the river example is used to illustrate how the Streaming Workflow System can optimize an evolving workflow.

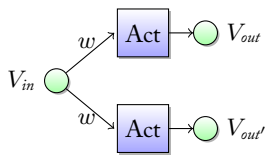
Describing Transformation Rules Transformation rules are described visually by two graphs that represent the before (left hand side, or lhs) and after (right hand side, or rhs) state. In addition preconditions and introduced window sequences are stated. Quantification of the graph structure and preconditions is handled either intuitively or by choosing a specific case; further research or an implementation will have to formalize this to make the rules generic.

Applicability of Rules A transformation rule is applicable when 1. a subgraph match is found, 2. all preconditions are true, 3. all variables referenced by the rule are defined and 4. in each precondition and variable definition the lhs and rhs are of the same type.

4.1 Zip Rule

This section starts with a very simple rule to reiterate the concepts from the formal model. Be not deceived by its simplicity as it is the most

Before transformation (lhs):



After transformation (rhs):

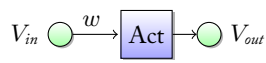


Figure 4.1: Zip Rule

fundamental rule of all. Many other rules increase the complexity of the workflow graph, trying to create possibilities for the Zip Rule to be applied.

The Zip Rule is the embodiment of the notion of trivially equivalent views from Lemma 1. To proof the validity of the rule, start with proofing that V_{out} and $V_{out'}$ are trivially equivalent. Views must meet three conditions to be trivially equivalent:

1. Produced by equivalent activities.
   Check; expressed by the fact that all activities in Figure 4.1 have the same label.

2. From equivalent input views.
   Check; the input views of all activities in Figure 4.1 are *identical*, which is stronger than equivalent.

3. The same window sequence.
   Check; all input relations in Figure 4.1 have the same label.

For each view on the left hand side (lhs), an equivalent view exists on the right hand side (rhs), because V_{out} and $V_{out'}$ are trivially equivalent. Therefore the workflow before application of the rule is strict equivalent with the workflow after application of the rule [Definition 13]. □

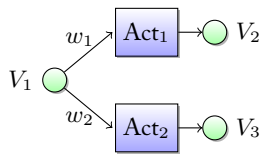
4.2 Copycat Rule

The Copycat Rule is very similar to the Zip Rule. Only one condition from the trivial equivalence of views is not fulfilled in the following situation. This section shows a non-trivial proof for a rule and illustrates scenarios that a Streaming Workflow System is expected to optimize.

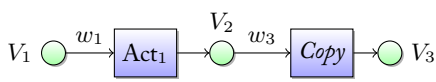
Scenario Although copying someone else's work maybe a crime in an academic context, being 'inspired' by the way a colleague deals with certain tasks is absolutely acceptable for day-to-day work. Reinventing existing solutions is often not the best way to spend time.

The Copycat Rule enables workflow optimizations in the following scenarios. For the second and third scenario it is important to remember that from the system's point of view workflows never change (see Assumption III).

Before transformation (lhs):



After transformation (rhs):



Preconditions:	
Act$_1$ and Act$_2$ are equivalent	
w_1 and w_2 are both time-based	
epoch(w_1)	$\neq$ *epoch*(w_2)
delta(w_1)	$=$ *delta*(w_2)
offset(w_1)	$=$ *offset*(w_2)
ws(w_1)	$=$ *ws*(w_2)
$\exists n > 0$: *epoch*$(w_2) =$ *epoch*$(w_1) + n *$ *delta*(w_1)	

Definition of w_3:	
epoch(w_3)	$=$ *epoch*(w_2)
delta(w_3)	$=$ *delta*(w_2)
offset(w_3)	$= 0$
ws(w_3)	$=$ *delta*(w_2)

Figure 4.2: Copycat Rule

- A user sees a workflow from a colleague and thinks that it is really a clever idea. So the user creates an (almost) identical workflow for himself. This is a literal copycat situation!
- Some task is reassigned. Not familiar with the task, the new assignee starts off by doing things the way the former assignee did. When he understand things better, he will make his own modifications.
- Some business practice is repeated regularly (e.g. every fiscal period). Each time the procedure is slightly tweaked, resulting in a new workflow being added to the system.

All three scenarios share one observation: the main difference between two workflow specifications is when they were specified. In terms of window sequences all variables are the same except for the *epoch* variable.

The Rule The *Before* and *After* situation in Figure 4.2 show the structural changes to the workflow. The preconditions formalize the observations mentioned before: everything is the same, except the *epoch*.

The preconditions contain one special requirement: the *epoch*s must align. As a consequence of this alignment and the other preconditions, the windows defined by w_2 are a proper subset of the windows defined by w_1. This means that V_3 is a proper subset of V_2. Since it is known when Act_2 was defined, all that needs to be done to create V_3 from V_2 is copy everything added after Act_2 was defined. *Copy* and w_3 are defined in such a way that everything in V_2 that also should be in V_3 is copied. Since w_3 specifies the same reference points as w_1, new data added to V_2 is copied to V_3 in the same system state transition.

Formal Proof For a transformation to be correct, the workflow before application must be equivalent to the workflow after application. Because equivalent workflows have equivalent views [Definition 13], it is sufficient to proof that $V_2 \equiv V_{copy}$ in a workflow that combines the lhs and rhs of the transformation as shown in Figure 4.3.

In this proof the assumption is made that all activities have been defined before the start of the system with different *epoch*s and the transformation has been applied but without removing the replaced activities/views. This variation from production scenario does not affect the

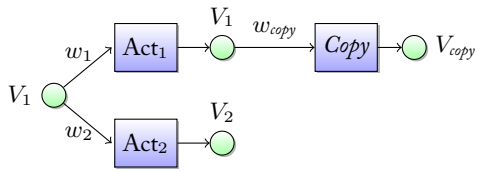


Figure 4.3: Copycat Rule LHS and RHS Combined

validity of the proof because the *epoch*s cause the activities to behave as if they have yet not been defined. The workflow system runtime treats an activity with a future *epoch* as non-existent. (Of course optimization can already be applied to these activities.)

Equivalence of views is defined as 'having the same content at all points in time' [Definition 12]. Therefore the equivalence $V_2 \equiv V_{copy}$ is shown with a proof by induction, using the time t as induction variable.

Σ_t represents the state of the system at time t *before any* activity enabled at t is executed. Σ_{t+1} represents the state of the system *after all* activities enabled at t have been executed. For intermediate states the notation $\Sigma_{t'}$ is used.

Given

1. The combined workflow of both sides of the transformation. See Figure 4.3.
2. The preconditions for the transformation:
   a) $\text{Act}_1 \equiv \text{Act}_2$
   b) w_1 and w_2 are both time-based
   c) $\exists n > 0 : \mathit{epoch}(w_2) = \mathit{epoch}(w_1) + n * \mathit{delta}(w_1)$
   d) (implied by above: $\mathit{epoch}(w_1) < \mathit{epoch}(w_2)$)
   e) $\mathit{delta}(w_1) = \mathit{delta}(w_2)$
   f) $\mathit{offset}(w_1) = \mathit{offset}(w_2)$
   g) $\mathit{ws}(w_1) = \mathit{ws}(w_2)$
3. The window w_{copy} as defined by the transformation:
   a) $\mathit{epoch}(w_{copy}) = \mathit{epoch}(w_2)$
   b) $\mathit{delta}(w_{copy}) = \mathit{delta}(w_2)$
   c) $\mathit{offset}(w_{copy}) = 0$
   d) $\mathit{ws}(w_{copy}) = \mathit{delta}(w_2)$
      (select all since prev execution of Act_1)
4. *Copy* verbatim copies *valid_on* of input tuples.

Basic Step At $t = 0$, all views in the system Σ_0 are empty. Since empty views have identical content, the conclusion is that $V_2 =_0 V_{copy}$, by definition of an empty system [Assumption III].

Induction Step According to the induction hypothesis, in system state Σ_t, it is true that $V_2 =_t V_{copy}$. In the next system state Σ_{t+1}, it must be true that $V_2 =_{t+1} V_{copy}$.

There are 4 cases to consider. t either aligns with all window sequences or does not align with any. If t aligns, then it falls into one of three sections of a timeline partitioned by $epoch(w_1)$ and $epoch(w_2)$ as illustrated in Figure 4.4. The first execution of an activity is no different from any other execution of that activity. Therefore $t = epoch(w_1)$ and $t = epoch(w_2)$ are not separate cases.

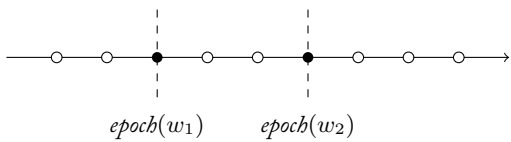


Figure 4.4: Partitioned Timeline

The first thing to proof is that either all or none activities align with t. As a result, no cases have to be considered in which some activities do align and some activities do not align.
Because $epoch(w_{copy}) = epoch(w_2)$ [3a] and $delta(w_{copy}) = delta(w_2)$ [3b], *Copy* aligns trivially with Act_2 by definition of alignment. Because $delta(w_1) = delta(w_2)$ [2e] and [2c], Act_1 and Act_2 also align (in fact [2e]+[2c] *is* the definition of alignment, see Definition 9)
Alignment is an equivalence relation (thus transitive) therefore all activities align and therefore all activities do align with t or all activities do not align with t.

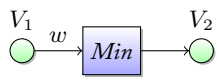
Lets consider the first case:

- No alignment.
  t is not a reference point of any of the activities in the workflow. None of the windows align with t so none of the activities is enabled for t, no activity is executed and there are no changes to the views and thus $\Sigma_{t+1} = \Sigma_t \implies V_2 =_{t+1} V_{copy}$.

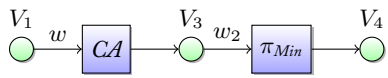
In the following 3 cases, the windows of all activities align with each other and with t. Because t is a valid reference point of at least one of the activities in the workflow system, it is called *Now*. This switch of terminology is made to relate to the notation used to specify workflows more closely.

- Before the first time Act_1 is enabled.
  $\mathit{Now} < \mathit{epoch}(w_1) \Rightarrow \mathit{Now} < \mathit{epoch}(w_2) \wedge \mathit{Now} < \mathit{epoch}(w_{\mathit{copy}})$
  none of the activities is executed and thus $\Sigma_{t+1} = \Sigma_t \Rightarrow V_2 =_{t+1} V_{\mathit{copy}}$ [2d,3a]
- Starting the first time Act_1 was enabled but before the first time Act_2 is enabled.
  $\mathit{Now} \geq \mathit{epoch}(w_1) \wedge \mathit{Now} < \mathit{epoch}(w_2)$
  - Act_1 is executed and adds $n \geq 0$ tuples to V_1
  - Act_2 is not executed so V_2 does not change
  - *Copy* is not executed because
    $\mathit{epoch}(w_{\mathit{copy}}) = \mathit{epoch}(w_2)$ [3a] so V_{copy} does not change
  - $V_2 =_t V_{\mathit{copy}}$ and nothing changes, thus $V_2 =_{t+1} V_{\mathit{copy}}$
- Starting the first time Act_2 is enabled.
  $\mathit{Now} \geq \mathit{epoch}(w_2)$
  - Because of the input dependency relation between Act_1 and *Copy*, the workflow system will ensure that *Copy* is always executed after Act_1 for the same value of *Now*. [Assumption II] There are 3 possible execution orders: $(\text{Act}_1, \text{Act}_2, \mathit{Copy})$, $(\text{Act}_1, \mathit{Copy}, \text{Act}_2)$ and $(\text{Act}_2, \text{Act}_1, \mathit{Copy})$. In this proof the first order is assumed. The other two orders can be proven similarly to result in $V_2 =_{t+1} V_{\mathit{copy}}$.
  - Act_1 is executed because $\mathit{epoch}(w_1) < \mathit{epoch}(w_2)$ [2d] and adds zero or more tuples to V_1, producing $\Sigma_{t'} = \Sigma_t \cup \mathit{output}(\text{Act}_1)$
  - Act_2 is executed and adds n tuples to V_2, identical to the tuples Act_1 has added to V_1 because $\text{Act}_2 \equiv \text{Act}_1 \wedge w_1 \equiv_{\mathit{Now}} w_2 \wedge V_1 = V_1$ (same operation, same selection, same source thus same output), producing $\Sigma_{t''} = \Sigma_{t'} \cup \mathit{output}(\text{Act}_2)$ [2a,2efg,1]

Before transformation (lhs):



After transformation (rhs):



Definition of w_2:	
$epoch(w_2)$	$= epoch(w)$
$delta(w_2)$	$= delta(w)$
$offset(w_2)$	$= 0$
$ws(w_2)$	$= delta(w)$

Figure 4.5: Shared Loop Rule

- Now V_2 has more tuples than V_{copy}; but execution takes zero time [Assumption I], so this is not a point in time for the purpose of view equivalence.
- *Copy* is executed because $epoch(w_{copy}) = epoch(w_2)$ [3a]. Execution takes place after Act_1 has added n tuples to V_1 . The window w_{copy} selects exactly the n tuples just added to V_1. *Copy* copies n tuples to V_{copy}, producing Σ_{t+1} since all activities have been executed.
- In Σ_{t+1} n tuples have been added to both V_2 and V_{copy}. These tuples are equivalent because (indirectly) they have been produced by equivalent activities from the same input data. Thus *Copy* must be implemented so that it literally copies the $valid\_on$ value from the input tuples. [4])
- Thus $V_2 =_{t+1} V_{copy}$

Conclusion of Proof For all t has been proofed that $V_2 =_t V_{copy}$, thus by definition of view equivalence $V_2 \equiv V_{copy}$ in the combined workflow in Figure 4.3.

Since the other views have not been redefined, every view on the lhs of the rule has an equivalent on the rhs of the rule and vice versa. The rule preserves strict equivalence and thus is the Copycat Rule a correct transformation of a streaming workflow. □

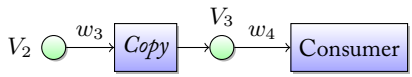
4.3 Shared Loop Rule

This rule uses a very simple observation: if you need both the *Min* and *Max* of a set of data, you can find them both by iterating over the data only once. A Common Aggregate (*CA*) activity does exactly that: it computes various common aggregates like *Min*, *Max* and *Count* over a set of data. To present the user with the view he expected, the rule adds an extra *Project* (π) activity to select only the requested aggregate.

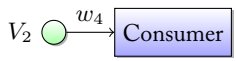
The validity of this rule is proofed by treating it as the Copycat Rule limited to a subset of schema attributes. A Common Aggregate activity is the aggregate it replaces, except that it outputs more attributes and *Project* is *Copy* except that it outputs less attributes.

Contrary to the Copycat Rule, the Shared Loop does not preserve strict equivalence. Because the view produced by the Common Aggregate contains more attributes, the rule only preserves weak equivalence.

Before transformation (lhs):



After transformation (rhs):



Preconditions:

$w_4 \subseteq w_3$

Consumer can be any activity.

Figure 4.6: Copy Elimination Rule

After transformation (rhs):

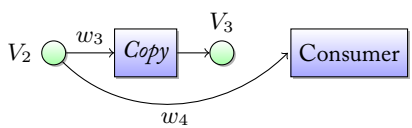


Figure 4.7: Copy Bypass Rule (Righthand Side)

4.4 Copy Elimination/Bypass Rule

This section shows how a rule that only preserves weak equivalence can be adapted to preserve strict equivalence. First the weak variant is explained.

The Copycat Rule adds a *Copy* activity to a workflow. Often the filtering performed by the *Copy* activity is redundant because the consumer of the *Copy* activity has a window sequence that already does that. This will be the case when the user has defined a long workflow with multiple processing steps.

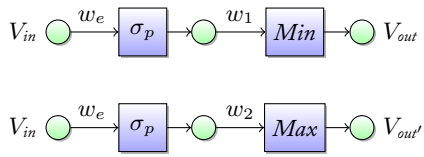
Figure 4.6 shows the Copy Elimination Rule that takes advantage of that situation. The statement $w_4 \subseteq w_3$ in the preconditions means that the reference points of w_4 are a subsequence of the reference points of w_3 (thus fewer windows), or that each window of w_4 selects a subset of the corresponding window of w_3 (smaller windows); it can also mean both. Because user will often define workflows with multiple processing steps on some data (like a pipeline), situations where $w_3 = w_4$ are expected not to be uncommon.

The Copy Elimination Rule not only eliminates an activity, it also eliminates the corresponding view. Therefore the rule does only preserve weak equivalence between the lhs and rhs but no strict equivalence. This is no problem, assuming that the user is not interested in the intermediate results.

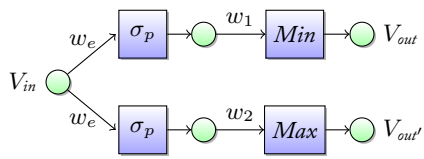
Now, the users interests have changed. To debug/inspect his workflow the user wants to see the intermediate views. The Streaming Workflow System can no longer apply the Copy Elimination Rule because it breaks equivalence when V_3 is included in the subset over which equivalence is defined.

Figure 4.7 shows an alternative transformation of the workflow that exploits the same idea of removing redundant filtering. This rule only bypasses the *Copy* activity and is hence called the Copy Bypass Rule. Because only an input relation is rerouted, the transformation preserves strict equivalence because for each view on either side of the transformation an equivalent view exists on the other side, namely the exact same view.

a. *Workflows as specified and seen by the user(s):*



b. *In the system runtime:*



c. *Apply Zip Rule:*

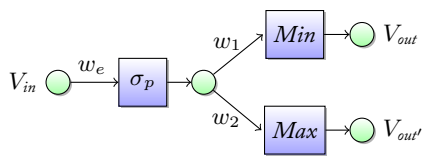


Figure 4.8: Example of a Multi-step Workflow Transformation

The precondition $w_4 \subseteq w_3$ implies alignment between the window sequences; therefore the validity of the Elimination and the Bypass Rule can be proofed similarly to the Copycat Rule with induction over time.

4.5 A Multi-Step Example

To illustrate how workflow transformation works, this section presents a bigger examples that includes all rules described in this chapter. The workflows are show in Figures 4.8 and 4.8.

In the example two workflows have been specified. One of them calculates the seven day sliding minimum value over a subset of measurements that satisfy a certain preposition. The other calculates the seven day sliding maximum over the same subset, although this workflow has been defined to start processing two weeks later than the first.

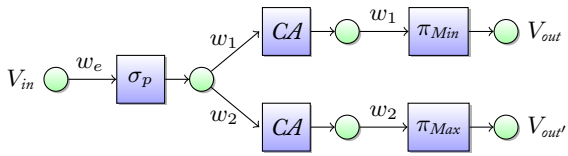
a. To start with, the workflows such as specified and seen by the users of the Streaming Workflow System are shown. The two workflows are completely separate from the user's point of view even though they share the same view as data source. Because the workflows haven been defined two weeks apart, the *Min* and *Max* activities have different window sequences.

b. If the two workflows were specified by the same user, he could have created a single workflow with multiple output views instead. Regardless of how the user(s) has/have specified the workflows, within the workflow system's runtime they are one workflow because they share a view.

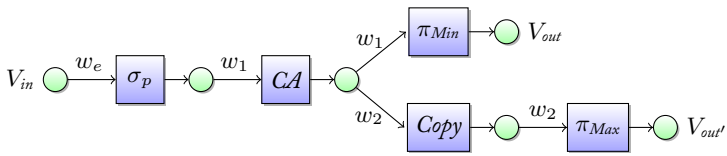
c. The user did not care about when the selection is actually performed and has therefore used the standard event-based window sequence w_e, so the first transformation that can be applied is the Zip Rule. Both branches of the workflow perform the same selection on the same set of data using the same window sequence. If the user had specified different window sequences, the system would have been able to use the Copycat Rule on the selection activities instead.

d. Next, the Shared Loop Rule is applied to both branches of the workflow individually. This results in two additional Project activities being added to filter out the requested aggregate values. The user did not require delayed processing, so $\mathit{offset}(w_1) = \mathit{offset}(w_2) = 0$, and as a

d. Apply Shared Loop Rule to Both Branches:



e. Apply Copycat Rule:



f. Apply Copy Elimination Rule:

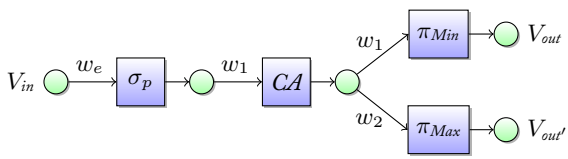


Figure 4.8: Example of a Multi-step Workflow Transformation (cont.)

consequence the window sequences of the new projection activities are the same as for the aggregation activities.

e. Because the only difference between w_1 and w_2 is the *epoch*, the two Common Aggregate activities are an easy match for the Copycat Rule. One aggregate is removed and replaced by a *Copy* activity.

f. Given that $w_2 \subseteq w_2$, the preconditions for the Copy Elimination Rule are fulfilled and the *Copy* activity can be removed.

The workflows in ***a.*** and ***f.*** are weak equivalent. The view produced by the Common Aggregate has no equivalent in the original workflow and must therefore be excluded from the equivalence. Weak equivalence is acceptable here because this is exactly the asymmetric situation as explained in Section 3.3.

4.6 Monitoring a River Slightly Differently

Reconsider the streaming workflow from Figure 2.1. A scientist has devised a method to detect groundwater influxes in dikes along a river. He uses a streaming workflow to filter and aggregate the sensor data before feeding it into the Influx Detector activity

The scientist has concluded that he needs to filter out more measurements because the temperature sensors generate more noise than expected. He changes the 90% activity into a 80% activity and submits the new workflow specification to the system. Using the Zip Rule, the system can share the 'flow' branch of the workflow between the original and the revised workflow.

In another update the scientist adds *Avg*, *Min* and *Max* activities to the 'flow' branch because they are needed by his new Influx Detection algorithm. Again the system uses the Zip Rule to reuse the unchanged parts of the workflow. Additionally the system applies the Shared Loop Rule, so the extra aggregates have virtually no extra processing costs.

The story above assumes that the scientist wants the system to reprocess the data already processed by the old workflow. If he had specified that only new data has to be processed by the new workflow, the system could have applied the Copycat Rule to obtain basically the same sharing as achieved through the Zip Rule.

Chapter Five

Tuple-based Transformation Rules

Many activities manipulate views in terms of individual tuples. Common characteristics of these activities are: 1. they operate at the tuple level, 2. time-based selection is meaningless because of this and 3. they are used to manage data and have no intrinsic purpose in the workflow.

In the 'Monitoring a River' example in Figure 2.1 tuple-based activities have been left out for clarity, although the workflow implicitly contains a *Union* to collect all sensor data from a specific type of sensor before filtering the data. If the scientist would want to limit his research to a smaller section of the river, he would add *Select* activities to the workflow. The usage of both an implicit *Union* activity and optional *Select* activities illustrates that tuple-based activities are for data management and have no intrinsic purpose.

Many operators in relational algebra can be classified as tuple-based activities. The most notable exception is the Join; a Join does not quite match the characteristics above, setting it apart from the other the activities discussed here. The Join activity is not only discussed in Section 5.2 but also in the next chapter.

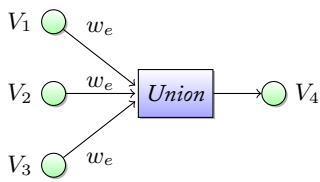
This chapter starts with two rules that do come directly from relational algebra.

5.1 Union Associativity Rule

A common activity is *Union*. It copies all tuples from any input view to the output view, preserving all tuples and all attributes. The output view is sorted on time. Contrary to the union operator in relational algebra, a *Union* activity does not have to perform duplicate elimination as all tuples are unique by source and timestamp.

A well-known property of the union operator is associativity; which means that $(A \cup B) \cup C = A \cup (B \cup C)$.

Before transformation (lhs):



After transformation (rhs):

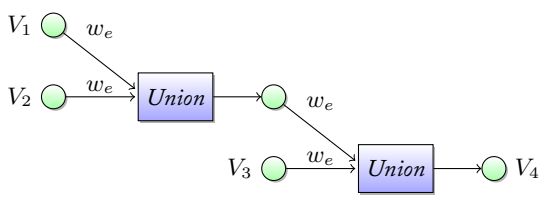
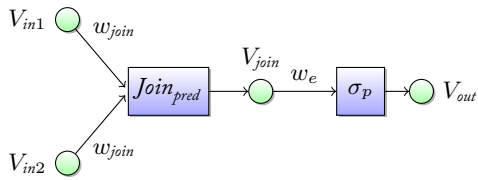
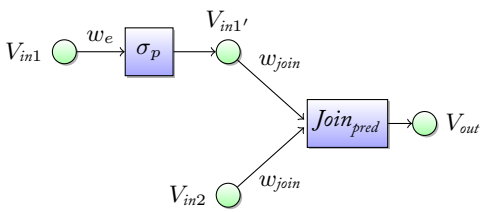


Figure 5.1: Union Associativity Rule

Before transformation (lhs):



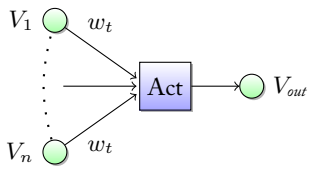
After transformation (rhs):



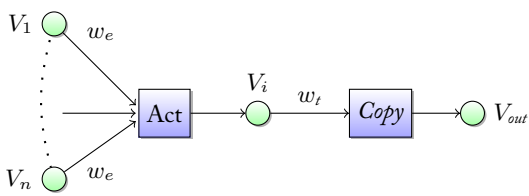
Preconditions:
$atoms(p) \subseteq schema(V_{in1})$
$atoms(p) \cap schema(V_{in2}) = \emptyset$

Figure 5.2: Selection Pushdown Rule

Before transformation (lhs):



After transformation (rhs):



Preconditions:
w_t is time- or tuple-based
Act is a tuple-based activity
Act has ≥ 1 inputs

Figure 5.3: Masquerade Rule

In Figure 5.1 a transformation rule is shown for the case of three input views. Because of the fresh tuple semantics of a *Union* and the w_e window sequence used, the activities behave identical to the relational operator and the associativity known from relational algebra can be directly used.

The usefulness of this rule is illustrated with the following scenario. In a sensor deployment constantly sensors are being added and removed. Assume a workflow is defined with a union of V_1 and V_2. Remember that workflows are immutable [Design Choice III]. Now a new sensor is deployed and a new workflow is added to the system with a union of V_1, V_2 and V_3. Using the Union Associativity Rule, the system only has to add the tuples from V_3 to an existing *Union*. The Union Associativity Rule is useful when considering many slightly different workflows over time because workflows are immutable.

5.2 Selection Pushdown Rule

The rule in Figure 5.2 shows another well-known principle from relational algebra: selection can be pushed through joins. The benefit of doing this is reduction of the amount of tuples that have to be processed by the expensive *Join* activity. In [14] it is observed that many continuous querying systems do the opposite: they push down the *Join* to increase the potential for sharing. This rule can be used both ways.

In [15] it is argued that both pulling up and pushing down are not optimal and another approach is presented that prevents unnecessary work (which they call zombie tuples) but also prevents the same operation being applied to a tuple twice.

5.3 Masquerade Rule

Most tuple-based activities use the fresh tuple semantics [Definition 8], which in combination with the standard tuple based window sequence w_e [Definition 7] results in behavior very similar to their namesakes in relational algebra. The previous two rules only transform workflows that adhere to this standard.

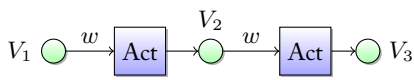
Sometimes these activities however do not use the standard tuple based window sequence w_e, because the user has specified another window sequence. Although the usefulness of other window sequences is

Before transformation (lhs):

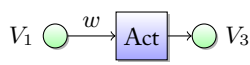
After transformation (rhs):

Figure 5.4: Single Input Union Rule

Before transformation (lhs):



After transformation (rhs):



Preconditions:

Act is idempotent when combined with w

Figure 5.5: Idempotent Activities Rule

unclear, it is clear that they can be specified in the formal model. Therefore, transformation rules have to take them into account. Doing so on a per rule basis would lead to unnecessary rules and complexity within rules.

The Masquerade Rule as shown in Figure 5.3 converts an inherently tuple based activity into an activity with the standard window sequence w_e. To hide the difference (in particular the update frequency), an additional *Copy* activity with the original window sequence is added to the workflow. After application of the Masquerade Rule, tuple-based activities can be transformed using transformation rules directly derived from relational algebra.

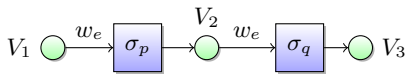
When the Copy Elimination Rule can be applied after application of the Masquerade Rule, the introduced overhead is eliminated. This will often be the case as argued in Section 4.4. In this way the system eliminates an arbitrary restriction defined by the user, increasing the performance of the individual workflow and the shareability of the intermediary results.

Because an intermediary view is introduced, the Masquerade Rule does preserve weak equivalence but not strict equivalence. A proof of this equivalence would use induction over time to show that the introduced view V_i is always 'ahead' of the view V_{out} and that the *Copy* activity copies the tuples at the correct time to V_{out}. To proof that the content of the tuples is correct, consider that Act is tuple based and that *Copy* does not change the contents of the tuples.

Other Uses Similarly to the Copycat Rule, only fresh tuples are selected and copied by the introduced *Copy* activity because the window sequence is partitioning. This way, the Masquerade Rule also handles situations where a *Union* is defined with a sliding window. The *Union* activity is fresh tuple based so this does not have any performance impact, but it makes the *Union* more shareable because after application of the rule it is only unique in the combination of its *offset* and input relations.

Another alternate use of the Masquerade Rule is when a hopping window is defined for a *Union* activity. The hopping causes the *Union* to act like a timeline selection. This selection can be done by a separate activity, increasing the potential for sharing the *Union*.

Before transformation (lhs):



After transformation (rhs):

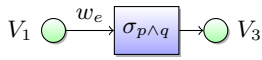
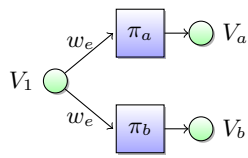


Figure 5.6: Collapse/Split Select Rule

Before transformation (lhs):



After transformation (rhs):

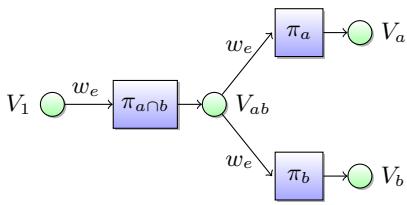


Figure 5.7: Intermediate Project Rule

5.4 Other Rules

Given the Masquerade Rule, many more rules known from relational algebra can be directly mapped to workflow transformations. This section gives four examples. All these rules assume the Masquerade Rule has been applied when necessary to transform the workflow to only use the w_e window sequence.

Single Input Union Rule There is nothing (in the formal model) preventing the user from creating a *Union* activity with only a single input relation. Also, the system may be able to deduce that some input view will always be empty and can therefore remove an input relation, resulting in a *Union* with one input. In that case, the *Union* does effectively nothing and can be removed with the rule shown in Figure 5.4.

Idempotent Activities Rule In some cases an activity does nothing when applied twice because it is idempotent. Figure 5.5 shows a rule that eliminates one of the activities in such as case.

Select and *Project* are idempotent combined with any window sequence because these activities follow the fresh tuple semantics. *Copy* in combination with a partitioning window sequence is also idempotent.

Time-based activities can also be idempotent. Any distributive aggregate over a certain time interval that is its own super aggregate is idempotent. An aggregate over a tuple-based interval is not idempotent, because a tuple interval has no universal meaning.

Collapse and Split Select Rule Selection is commutative, therefore multiple *Select* activities can be combined, split-up and (by repeated application of the rule) reordered. Figure 5.6 shows the rule; which side of the transformation is the before state and which side the after state is rather arbitrary.

Intermediate Project Rule The rule in Figure 5.7 allows the system to insert an extra intermediate *Project* activity. This could be done to minimize the size of tuples to be stored after application of the Shared Loop Rule (see Section 4.3), by eliminating attributes that are never going to be used.

Chapter Six

Advanced Transformation Rules

In this chapter transformation rules for two important activity types will be discussed: aggregates and joins. Much research has been done for the optimization of aggregates and joins in the context of streaming query processing. The rules in this chapter will demonstrate that the formal model can handle these common and important activity types.

6.1 Aggregate Split Rule

Much research has been done to optimize aggregates over sliding windows in streaming query processing [4, 16]. Also the 'Monitoring a River' example contains plenty aggregates: the majority of activities in Figure 2.1 is an aggregate (sensors do not count in this case since they do not process data).

A common optimization strategy is that for many aggregates the value over a big window can be computed from the values of smaller windows. The Aggregate Split Rule uses this idea to optimize a certain subclass of aggregates: distributive aggregates.

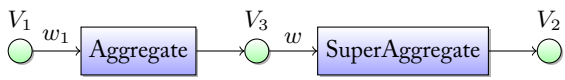
Definition 15: Distributive Aggregate An aggregate function $f()$ is distributive *iff* there exists a function $g()$ such that $f(A \cup B)$ can be computed using $g(\{f(A), f(B)\})$. Examples of distributive aggregate functions are *count*, *sum*, *min* and *max*. For *count*, $g = sum$, for the other examples $g = f$. (This definition is taken from [13])

The function g is referred to as the SuperAggregate in the transformation rule. In the workflow model a view with the intermediary results of a distributive aggregate has the same relational schema as the view with the end results. This is not true for other types of aggregates.

An *avg* aggregate, common as it is, is not distributive because the intermediate results need to contain both *sum* and *count*. The Common

Before transformation (lhs):

After transformation (rhs):



Preconditions:
w is time-based and sliding.
Aggregate is distributive.
$\exists n \in N_1 \ : ws = n * delta$

Definition of w_1:	
$epoch(w_1)$	$= epoch(w)$
$delta(w_1)$	$= delta(w)/n$
$offset(w_1)$	$= offset(w)$
$ws(w_1)$	$= ws(w)/n$

Figure 6.1: Aggregate Split Rule

Aggregate activity introduced in the Shared Loop Rule in Section 4.3 is distributive and can be defined to include *avg*.

Observations Application of the Aggregate Split Rule will optimize a workflow based on the following observations.

- In a sliding window sequence, each window overlaps at least two other windows. Given the semantics of a distributive aggregate it is possible to compute the value of a aggregate from the values of smaller aggregates [Definition 15]. These smaller aggregates can be used in the computation of multiple bigger aggregates when they are stored in a view. This sets the rule apart from other rules: it enables sharing *within* a single workflow.
- Many users will look at the same data at various levels of aggregation (drilling up and down). So often data is needed at multiple aggregation levels.
- Creating intermediate results from smaller windows increases the chance of intermediate results being shareable: in the rhs in Figure 6.1 there is a possibility that a view equivalent to V_3 is already available in the system; this effectively eliminates added costs when a new workflow is added.

The Rule Figure 6.1 shows the rule. The rule is very simple and effectively only rewrites Definition 15 using the syntax of the formal model.

Cost Reduction This transformation rule reduces the total amount of work the workflow system must do because $\mathit{cost}(\text{Aggregate}, w_1) + \mathit{cost}(\text{SuperAggregate}, w)$ is less than $\mathit{cost}(\text{Aggregate}, w)$ under the assumption that $\mathit{size}(V_3) \ll \mathit{size}(V_1)$. Assuming that the cost of executing the SuperAggregate is very low compared to the Aggregate, the total cost is reduced to $(1/n)$ where n is the number of partitions into which the original window is split.

An example is given in Figure 6.2 in which the computational cost of activation is shown for each window. If V_1 contains one tuple per minute and w is defined as 'last two hours, update every hour' then the Aggregate has to process 120 tuples every hour. After applying the rule, the Aggregate has to process 60 tuples every hour and the

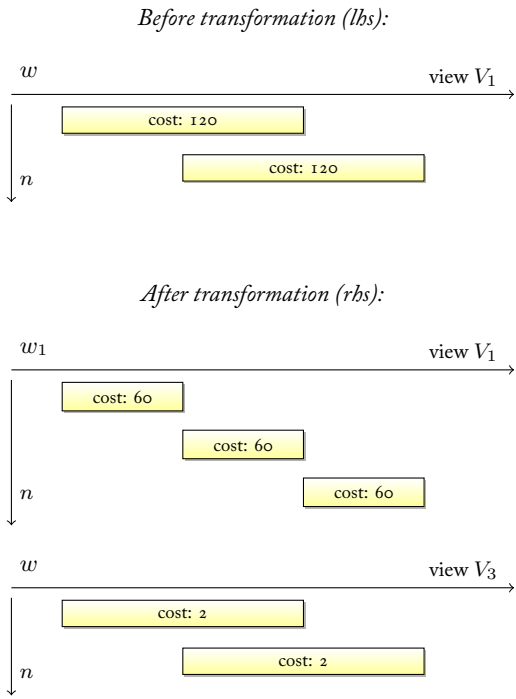


Figure 6.2: Computational Costs in the Aggregate Split Rule

SuperAggregate has to process only the 2 tuples produced in the last two hours by the Aggregate. Thus the cost of execution has dropped from 120 to $60 + 2 = 62$, assuming the amount of tuples is a reasonable cost indicator.

This reduction of computational cost clearly does not apply to partitioning and hopping window sequences since there is no overlap to exploit. A different optimization criterium for a Streaming Workflow System is spreading system load. When the Streaming Workflow System is optimizing for system load, the Aggregate Split Rule can be used on partitioning and hopping window sequences to break up large chunks of work so system load can be spread more evenly.

Repeated Application The SuperAggregate is a distributive aggregate itself, allowing the rule to be applied repeatedly, creating a hierarchy of aggregates along the time domain. Repeated application is limited by the fact that when the difference in size between V_1 and V_3 becomes too small, the assumption $cost(\text{Aggregate}, w_1)+cost(\text{SuperAggregate}, w) < cost(\text{Aggregate}, w)$ breaks.

Formal Model Limitations The cost reduction achieved by the Aggregate Split Rule is based on the assumption $size(V_3) \ll size(V_1)$. When the rule is applied repeatedly, the assumption breaks as observed in the previous paragraph.

Another case in which the assumption breaks is when a big window is frequently updated; for example: 'last week, update every minute'. As a week does contain 10080 minutes, is it very likely that the assumption breaks. In 'Resource sharing in continuous sliding-window aggregates' [4] this is avoided by using a hierarchy of aggregates. To compute an aggregate over a week sized window, compute daily and hourly aggregates and compute per minute aggregates. Next, to compute an aggregate over a 10080 minute window, start with the 6 days that are fully contained and 'pad' the interval before and after those days with hours and minutes to complete the window.

This approach can not be expressed in the formal model itself. The only way to implement this approach is to create an activity that implements this logic internally and reads from multiple input views selectively the required pieces of an aggregate.

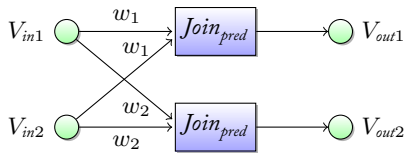
An important concern in [4] is the efficiency of lookups; the formal model does not facilitate reasoning about this.

6.2 Shared Join Rule

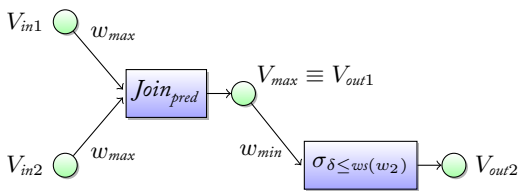
Another relational operator that has attracted much attention in continuous query processing is the join operator. It is used to join together data from different sensors. In the 'Monitoring a River' example, flow measurements and temperature measurement have to be paired with each other within a time window because influxes can only be detected by looking at abnormal combinations of the two.

An observation made by [14] is that when two joins have the same predicate, the output tuples of the join with the smallest window are

Before transformation (lhs):



After transformation (rhs):



Preconditions:

w_1 and w_2 align.
$ws(w_1) \neq ws(w_2)$

Definitions:

$\delta = |valid\_on_1 - valid\_on_2|$ for each tuple

$$w_{max} = \begin{cases} w_1 & \text{if } ws(w_1) > ws(w_2) \\ w_2 & \text{otherwise} \end{cases}$$

$$w_{min} = \begin{cases} w_2 & \text{if } ws(w_1) > ws(w_2) \\ w_1 & \text{otherwise} \end{cases}$$

Figure 6.3: Shared Join Rule

contained in the output tuples of the join with the biggest window. The Shared Join Rule uses this observation to optimize *Join* activities in streaming workflows.

The Rule The Shared Join Rule is shown in Figure 6.3. The biggest *Join* is used as starting point. In the output tuples of the *Join*, the *valid_on* timestamps of both joined input tuples are preserved. Based on the distance between those two timestamps, an *Select* activity can filter out any tuple that would not haven been generated by the *Join* with the smallest window. (Thus it is not possible to transform tuple-based *Join* activities!)

The rule preserves strict equivalence. For any views on either side of the transformation an equivalent view is present on the other side [Definition 13]. Intuitively, this is true because there is clearly a subsumption relation between V_{out1} and V_{out2}.

Formal Model Limitations 1 For *Join* activities, the most natural window sequence would seem to be tuple trigger, but with a time-based window size. This is at the moment not possible in the formal model.

Formal Model Limitations 2 In [14] various algorithms are presented that differ in performance characteristics. The Shared Join Rule represents the most naive and most unfair method. It is considered unfair because big window joins can delay small window joins when executed together.

An alternative algorithm is presented that gives precedence to the join with the smallest window. This algorithm produces the tuples for the smallest join first and later produces the additional tuples for the bigger join.

The formal modal assumes that the workflow system is infinitely fast [Assumption I] and assumes sequential execution of dependent activities [Assumption II]. For a workflow on which the Shared Join Rule has been applied this means that the *Join* activity will always be completed before the filtering is started. Because every single activation of an activity is batch-based and activities cannot produce intermediate results, the fairer algorithm can not be expressed in the formal model.

Chapter Seven

Related Work

This chapter presents related work on streaming workflows and continuous querying. In the first section some remarks are made on the complications of comparing the formal model with related work. The other sections discuss transformations and optimizations presented in related work.

7.1 Complications

Comparing related work with the formal model can turn into an unbounded discussion for two reasons:

1. The Formal Model is Turing Complete The formal model presented in Chapter 2 puts no limits on what an activity can do. As a consequence the formal model is Turing complete: any computational process can be expressed. Figure 7.1 takes this to the extreme and shows the ultimate streaming workflow.

Figure 7.1: The Ultimate Workflow.

On the one hand it is clear that the extreme case in Figure 7.1 is not desirable; on the other hand, treating activities as black boxes is a deliberate design choice of the formal model.

2. The Formal Model is an Intermediate As described in Section 2.5 and illustrated by Figure 2.7 on page 21, the formal model is not intended to be a user interface (~ language) and the formal model is nei-

ther an execution plan or implementation blueprint. It is an intermediate abstraction intended to ease transformation and optimization.

Because the formal model is Turing complete, any approach or technique presented in related work can be mapped to the formal model. Because the formal model is an intermediate abstraction, adding an additional mapping (above or below in the system stack) is always possible. The real question is in which cases this reasoning is acceptable.

7.2 Query Graph Transformations

The formal model represents one or more queries as a big operator graph. The system can be optimized by applying transformation rules to the operator graph. Several such transformation rules are described in chapters 4, 5 and 6; those chapters are not intended to be a complete.

Below are some transformations of the operator graph as described in related work. Most of them can easily be rewritten as transformation rules in the formal model.

Everything from Relational Algebra The STREAM system [3, 17] adds streams as an extra type to standard relational processing. This is done in a way that preserves all the standard relational transformations. Also in the formal model many existing relational transformations can be expressed thanks to the Masquerade Rule from Section 5.3.

Filter-Window Commutativity A suggested transformation of the query graph in STREAM combines streams and relations: a selection operator is commutative with the filtering done by a window because a window is essentially a filter. This type of transformation could be easily described in the formal model.

Expression Signatures In NiagaraCQ [8] queries are grouped based on the signature of subexpressions. For example two select operators that only have a different constant will be grouped and executed together. The output of the grouped operator is split using a table that records these constants.

Grouping of exactly identical subexpressions happens in the formal model through zipping (see sections 4.1 and 4.5). Transformation rules

can be used to extract subexpressions from a query that is identical to a subexpression from another query. Continuing the example of two select operators: define a transformation rule that creates the disjunct of multiple predicates as an identical subexpression and and splits the results afterwards. The table used by NiagaraCQ to split the output is in the formal model solution embedded in the query graph.

Inserting, Combining and Reordering The Aurora system [6] uses the *boxes and arrows* paradigm, not only to represent the query, but also as an interface for the user. Aurora can apply various optimizations to the query graph. 1. inserting additional projections to minimize storage and communication costs, 2. combining sequential boxes into a single bigger box to eliminate overhead and 3. reordering of boxes to push selection down.

These are transformations that can be easily described in the formal model. The first requires the knowledge of which attributes are used by an activity; for each type of (custom) activity this needs to be described in a separate rule.

7.3 Adaptive Processing

An entirely different approach to streaming query processing are Eddies [5], which have been implemented in the TelegraphCQ [7] system. Instead of treating one or more queries as a big operator graph, the system associates *lineage* information with each tuple. Based on which operators a tuple has visited, which operators can be visited next and for which queries is a tuple dead, the Eddy routes each tuple individually through all required operators until it is complete for all queries.

SteMs [18] are an extension of the idea of Eddies in combination with joins. A SteM operator embodies the internal state that a join operator needs to maintain for each input relation. By adding a few routing restrictions to the Eddy, a join can be computed by sending tuples through the correct SteM operators. SteMs break the encapsulation of a classic join operator, which has two advantages: 1. the internal state of joins can be shared and 2. the Eddy can better adapt because the physical operations are better observable.

That storing the lineage of a tuple is not only useful for adaptive stream processing but can also be used to eliminate duplicate and use-

less processing is demonstrated in [15]. By including the evaluation of predicates in the tuples lineage, they prevent needless tuples from being generated by joins and they prevent repeatedly evaluating the same (expensive) predicate.

Storing the lineage of a tuple would be no problem in the formal model since activities are free to choose the schema of their output tuples. Adaptive routing and SteMs however could be a feature of a Streaming Workflow System, but the semantics can not be expressed in the formal model. Instead it could be used as a way to implement joins. The formal model can then be used to group joins on the same physical machine.

A problem with SteMs is that views in the formal model are purely relational and can not easily be used as 'scratchpad' for activities.

7.4 Optimization Criteria

Computational cost is just one criterium for which one or more queries can be optimized. There are other motives to transform a query plan. The formal model was intended to be agnostic of the optimization criterium, so this section details some criteria from related work.

Communication Cost The COSMOS [24] system tries to minimize the communication cost in a distributed stream processing system by query rewriting and merging query results. Communication cost is a problem because data rates in stream processing can be very high and in a distributed system communication can be intercontinental.

Because the formal model does not support multiple output streams from a single activity, many transformations define an activity that creates a superset of the results of the transformed activity and split this superset using additional select activities. This makes it very easy to minimize communication costs in the formal model because a merged result stream is often already available.

In [21] an algorithm is presented that minimizes data transmissions in sensor networks by locating operators at appropriate nodes in the network. The rational for such optimization is presented in [12] by claiming that CPU use is cheaper (in terms of power consumption of embedded devices) than communication.

Pushing selection and other processing steps down to leafs of the streaming workflow graph is not a problem to express in the formal model.

The SPADE [11] system tries to optimize for minimum communication because communication is the main cause of latency in the system.

Fairness of Optimizations Although a transformation may lead to global optimization, for individual queries the effects can be negative. In Section 6.2 the Shared Join rule is described. In [14] an approach similar to the Shared Join rule is presented; they consider various join algorithms that have different effects on the original queries. Their particular concern is the negative effect on the latency of the smallest of two joins when two joins are combined. After being combined, the query with the smaller join will produce results slower than originally because it has to wait for the bigger join.

These fairness considerations can not easily be expressed in the formal model since it assumes an infinitely fast system (Assumption I). This may be an acceptable limitation when considering non time critical systems, where the reduction of the total amount of work is the biggest concern.

7.5 Provenance

The formal model has a strong focus on exact and traceable results. This is due to the setting under which this thesis project started. In the example from Section 1.1, the Streaming Workflow System is introduced as a scientific tool in an academic setting. Reproducible results are the cornerstone of sound academic research.

The thesis of Reinier-Jan de Lange [10] on provenance in sensor networks has clearly contributed to the focus on exact and traceable results. This focus is most explicit in Design Choice III, which states that workflows must be immutable for the sake of provenance/governance.

In related work on streaming query processing, provenance does not seem to be a concern. Rapid and predictable response-times are much more important and results do not have to be exact and reproducible.

7.6 Approximation

Approximation is very a very natural thing to do, as sensor data itself is an approximation of reality. Due to the focus on exact and reproducible results, the formal model has not considered approximations. But because the topic is so common in related work, a section here is still warranted.

Using approximations improves the performance of a system in several ways. In DataCube [13] it is observed that many of their users do not require exact results and often use statistical approximations. These approximations can be easily maintained incrementally, which is not possible for the exact holistic aggregates. Making algorithms over sliding windows incremental is the focus of [9].

A specific use of approximations is coping with peak load. In [19] the TelegraphCQ [7] system is extended with load shedding. Additional queues are inserted between data sources and the query. When the system can not keep up, these queues overflow and tuples must be dropped. The dropped tuples are synopsized and a shadow query plan processes these synopsizes. The estimated results from the synopsis tuples are merged with the exact results from the original query and presented to the user together.

A optimization problem with approximations is that approximations often require building internal state. In [17] it is suggested to not only share intermediate results, but also the synopsizes build by operators. In the formal model it is hard to share synopsizes because of the structure of views.

Chapter Eight

Conclusions

This chapter will first give an overview of the 'Monitoring a River' example that shows how the work presented this thesis can be useful. Then answers to the research questions are given, followed by the results and contribution. In Section 8.4 a suggested approach towards actually building an optimizer for a Streaming Workflow System is presented.

8.1 Monitoring a River: Overview

Throughout the previous chapters the example of a scientist who wants to detect ground water influxes along a river has been used to illustrate the Streaming Workflow System, the formal model and transformation rules.

In Section 1.1 the example was introduced and in Section 2.1 a streaming workflow was presented that processes flow and temperature measurements. The workflow showed that the formal model can indeed be used to describe the data processing required in the example.

Chapter 4 presented generic transformation rules that can transform streaming workflows based on their abstract structure and (window sequence) characteristics. To illustrate the usefulness of these rules, a more abstract and hypothetical example was introduced in Section 4.5. The example showed that two individual streaming workflows can share computation through application of the right transformation rules in the right order. This abstract example also illustrates the (hypothetical) situation that would exist within the runtime of a Streaming Workflow System.

In Section 4.6 the river example was used to show how the Streaming Workflow System can use the transformation rules presented in Chapter 4 to optimize a streaming workflow as it evolves during a research project.

In Chapter 5 transformation rules for tuple-based activities were presented. Although the example did not include those explicitly, they are nonetheless indispensable for streaming workflow processing. The introduction to Chapter 5 describes this.

Aggregate functions and joins are a key component for streaming workflows; in the 'Monitoring a River' example aggregates account for the majority of activities. Transformation rules for those activities are described in Chapter 6.

The 'Monitoring a River' example has demonstrated that the formal model and transformation rules presented in this thesis are up to the task of modeling and transforming a streaming workflow.

8.2 Answers to the Research Questions

This section answers the research questions postulated in Section 1.6

1. **How must the (existing) formal model be adapted to allow for transformation?** Chapter 2 has explained the new formal model. The big change with respect to the original model is the predictability of activities. The old model used triggers to determine when an activity is activated. The new model replaces them with window sequences that are fully predictable.

   Another change is the definition of deterministic activities. Design Choice IV defines 'stream determinism', which is weaker than the determinism in the old model but is sufficient for the transformations presented.

   Other additions to the model include the concept of fresh tuples and the definition of a standard window sequence for tuple based activities.

2. **What is a suitable definition of equivalence between workflows?** In Chapter 3 equivalence of activities, views and workflows was defined. In the following chapters it was shown that this definition is suitable: the definition captures the purpose of an optimizer and the definition can be used to proof that transformation rules preserve equivalence.

   The definition of workflow equivalence was split into strict and weak equivalence to make the asymmetry between theory and

practice explicit. The weak equivalence of two workflows is defined with a subset of the views in those workflows to honor the fact that in practice equivalence is not symmetrical and the fact that an equivalent view can be recreated when needed.

3. **Can transformation rules be defined for common types of activities?** Chapters 4, 5 and 6 have shown various transformation rules. For many rules a proof was provided and for others an indication was given how a similar proof can be constructed. This shows that the rules are valid and that the definition of equivalence from Chapter 3 is suitable.

   Rules were presented for generic situations, for common (relational) activities and for aggregates and joins. It was shown that common activity types can be described and that existing continuous querying research can be described in the formal model.

8.3 Results and Contribution

The result of this masters thesis project is a formal model for streaming workflows and a set of transformation rules valid within that model. The quality and quantity of the rules is such that the primary objective of this research is considered to be met.

Although the expressive power of the model has been reduced at the cost of predictable behavior, it is still more than adequate for modeling streaming workflows. The added limitations and changes do not impair the functionality of the model. In addition, the new formal model can be interpreted as a subset of the original model. Therefore the transformation rules can be applied to any (sub-) workflow in the original model that happens to fit in the subset defined by the new model. Because preserving global equivalence is achieved by preserving local equivalence, only a subgraph of a workflow needs to fit in the new formal model for transformation rules to be applicable.

By adapting the model and defining transformation rules, it has been shown that automatic optimization of streaming workflows within an Streaming Workflow System is very well possible. The next section proposes how to continue towards this goal.

8.4 Suggested Approach to Building an Optimizer

In Section 1.4, optimization was described as searching for an alternative that is better than the original. In the same section it was made clear that choosing the best alternative is outside the scope of this thesis, limiting this thesis to searching for alternatives. The contribution of this thesis is actually smaller than that: this thesis presents transformation rules to create an alternative streaming workflow from a given one. How to navigate the search space of all possible streaming workflows is still open.

Below is a suggested approach to building an actual optimizer for the Streaming Workflow System.

1. First of all, build an state space explorer! Model checkers are build to explore huge state spaces efficiently, so pick a graph-based model checker like GROOVE [1, 20] and extend it with support for transformation rules for streaming workflows.

   All the graph manipulation functionality is already available, so what needs to be added is support to express and manipulate window sequences; therefore a concise vocabulary for window sequence matching and manipulation will have to be chosen or developed. The language will have to support expressing concepts like alignment (Definitions 9 and 10) and to support introducing new window sequences using simple arithmetic as in the Aggregate Split Rule (Figure 6.1).

   Since an optimizer assumes that all transformation rules preserve equivalence, it is not needed at first to track which views are equivalent. But eventually it must be known which view in the system runtime maps to a view in the user-defined workflow. Therefore support is needed to specify in a transformation rule which views are equivalent.

2. Second, find some work for the state space explorer. Compile a set of theoretical (isolated) workflows and transformations, based on the transformation rules presented in this thesis and on the possibilities of the chosen window sequence manipulation language. This set can be used to test the state space explorer and to explore the limitations of the formal model and of the window sequence semantics.

3. Third, find some *real* work for the state space explorer. Compile a set of workflows and transformation rules that are more or less realistic. Build larger compounded workflows from the isolated workflows, because in the runtime of a Streaming Workflow System, all workflows submitted by the users are treated as a single big workflow with many outputs.
4. Finally, build an optimizer. Pick a metric to optimize streaming workflows for. Compute this metric for all streaming workflows generated by the state space explorer. Evaluate the metric by comparing the original workflow with the workflow selected as the best by the metric.
5. Additionally, improve the optimizer. Use heuristics to constrain the search space and to steer the search towards the optimum for the metric, whilst still being able to find an optimal streaming workflow. This might be challenging to implement as it requires modification of the state space exploration algorithm.

The first and second step can be started based on the results of this thesis. The other three steps can be repeated for multiple application domains.

Summary Create a set of workflows and transformation rules to test with. Adapt an existing graph-based model checker to generate the full search space (or at least up to a maximum search depth). Define and evaluate metrics. Use heuristics to steer the state space exploration.

8.5 Unanswered Questions

The previous section has presented an approach to actually build an optimizer. This section presents some unanswered questions that future work should answer.

How Can Not Streaming Data Be Modeled in the Formal Model? The way views are modeled in the current formal model is not suited to handle meta data repositories and other types of semi-static data. They are needed, for example, to record the calibration status of a sensor.

Another type of storage for which views are not very well suited is to share 'scratchpad' data between activities.

Should the System Handle Delayed Data? Handling delayed data (out of order) is either trivial or very complex. In certain domains users are accustomed to the fact that sensors fail and die; the user is glad the data arrived at all. In other domains this may not be the case, so the system has to mitigate late arriving data to a certain extend. Workflows that are not used for real-time monitoring may incorporate calls to external web services which can be slow or temporarily unavailable. The Borealis [2] system provides dynamic provisions of query results.

8.6 Evaluation

Foremost, this research has been a thought experiment and this thesis reflects that. The result of this research is an abstract but clear starting point for the approach suggested in Section 8.4.

This thesis is light on technical detail and literature references because building an optimizer has never been a goal of this research and neither has this research been intended to be a literature study into stream processing or scientific workflows.

Bibliography

[1] Groove. http://groove.cs.utwente.nl/.

[2] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. An abstract semantics and concrete language for continuous queries over streams and relations.

[4] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 336–347. VLDB Endowment, 2004.

[5] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. *SIGMOD Rec.*, 29:261–272, May 2000.

[6] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 215–226. VLDB Endowment, 2002.

[7] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. Telegraphcq: Continuous dataflow processing

for an uncertain world. In *Proceedings of the 2003 CIDR Conference*, 2003.

[8] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a Scalable Continuous Query System for Internet Databases. pages 379–390, 2000.

[9] Anita Dani and Janusz Getta. Conceptual modelling of computations on data streams, 2005.

[10] Reinier-Jan de Lange. Provenance aware sensor networks for real-time data analysis. Master's thesis, University of Twente, 2010.

[11] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134, New York, NY, USA, 2008. ACM.

[12] J. Gehrke and S. Madden. Query processing in sensor networks. *Pervasive Computing, IEEE*, 3(1):46 – 55, 2004.

[13] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 03 1997.

[14] Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref, and Ahmed K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 297–308. VLDB Endowment, 2003.

[15] Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, and Garrett Jacobson. The case for precision sharing. In *in VLDB*, pages 972–986, 2004.

[16] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, 2005.

[17] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation in a data stream management system. Technical Report 2002-41, Stanford InfoLab, 2002.

[18] Vijayshankar Raman, Vijayshankar Raman, Amol Deshpande, Amol Deshpande, Joseph M. Hellerstein, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In *In ICDE*, pages 353–364, 2003.

[19] F. Reiss and J.M. Hellerstein. Data triage: an adaptive architecture for load shedding in telegraphcq. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 155 – 156, 2005.

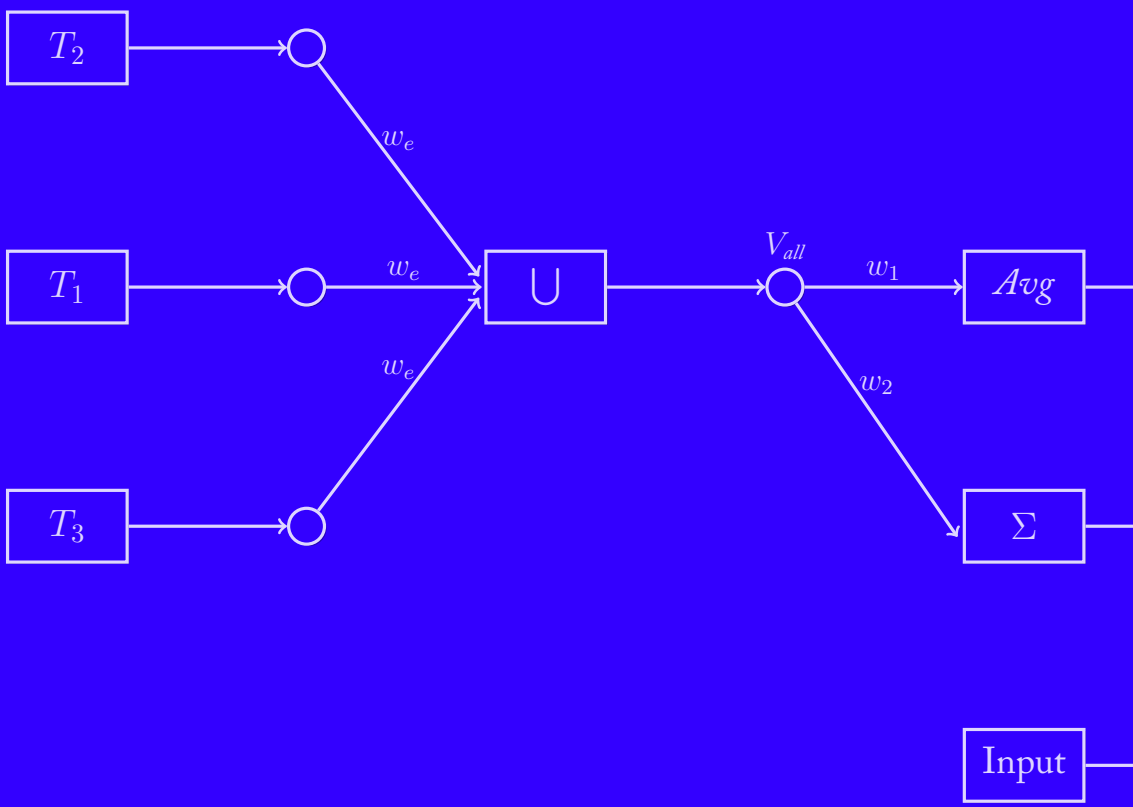
[20] A. Rensink. The groove simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.

[21] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. Operator placement for in-network stream query processing. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '05, pages 250–258, New York, NY, USA, 2005. ACM.

[22] A. Wombacher. Composable data processing in environmental science - a process view. In *Proceedings of IEEE International Conference on Services Computing (SCC2008), Honolulu, Hawaii, USA*, volume 2, pages 499–502, Los Alamitos, July 2008. IEEE Computer Society Press.

[23] Andreas Wombacher. Data Workflow — A Unified Time Controlled E-Science Processing Model. 2009.

[24] Yongluan Zhou, Karl Aberer, Ali Salehi, and Kian lee Tan. Rethinking the design of distributed stream processing systems.


T_2
T_1
T_3
w_e
w_e
w_e
$\cup$
V_{all}
w_1
w_2
Avg
Σ
Input