UNIVERSITY OF SOUTHAMPTON

Tracing Fine-Grained Provenance in Stream Processing Systems using A Reverse Mapping Method

by

Watsawee Sansrimahachai

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Engineering and Applied Science
Department of Electronics and Computer Science

April 2012

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

Tracing Fine-Grained Provenance in Stream Processing Systems using A Reverse Mapping Method

by Watsawee Sansrimahachai

Applications that require continuous processing of high-volume data streams have grown in prevalence and importance. These kinds of system often process streaming data in real-time or near real-time and provide instantaneous responses in order to support a precise and on time decision. In such systems it is difficult to know exactly how a particular result is generated. However, such information is extremely important for the validation and verification of stream processing results. Therefore, it is crucial that stream processing systems have a mechanism for tracking provenance - the information pertaining to the process that produced result data - at the level of individual stream elements which we refer to as *fine-grained provenance tracking for streams*. The traceability of stream processing systems allows for users to validate individual stream elements, to verify the computation that took place and to understand the chain of reasoning that was used in the production of a stream processing result.

Several recent solutions to provenance tracking in stream processing systems mainly focus on coarse-grained stream provenance in which the level of granularity for capturing provenance information is not detailed enough to address our problem. This thesis proposes a novel fine-grained provenance solution for streams that exploits a reverse mapping method to precisely capture dependency relationships for every individual stream element. It is also designed to support a stream-specific provenance query mechanism, which performs provenance queries dynamically over streams of provenance assertions without requiring the assertions to be stored persistently.

The dissertation makes four major contributions to the state of the art. First is a provenance model for streams that allows for the provenance of individual stream elements to be obtained. Second is a provenance query method which utilizes a reverse mapping method - stream ancestor functions - in order to obtain the provenance of a particular stream processing result. The third contribution is a stream-specific provenance query mechanism that enables provenance queries to be computed on-the-fly without requiring provenance assertions to be stored persistently. The fourth contribution is the performance characteristics of our stream provenance solution. It is shown that the storage overhead for provenance collection can be reduced significantly by using our storage reduction technique and the marginal cost of storage consumption is constant based on the number of input stream events. A 4% overhead for the persistent provenance approach and a 7% overhead for the stream-specific query approach are observed as the impact of provenance recording on system performance. In addition, our stream-specific query approach offers low-latency processing (0.3 ms per additional component) with reasonable memory consumption.

Contents

Declaration of Authorship

I, Watsawee Sansrimahachai, declare that the thesis entitled Tracing Fine-Grained Provenance in Stream Processing Systems using A Reverse Mapping Method and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as:
    - Watsawee Sansrimahachai, Luc Moreau and Mark J. Weal. "Fine-Grained Provenance Tracking in Stream Processing Systems," in *UK e-Science All Hands Meeting 2009*, Oxford, UK, December 2009.
    - Watsawee Sansrimahachai, Mark J. Weal and Luc Moreau. "Stream Ancestor Function: A Mechanism for Fine-Grained Provenance in Stream Processing Systems," in *Proceedings of the 6th IEEE International Conference on Research Challenges in Information Science*, Valencia, Spain, May 2012.

Signed:

Date:

Acknowledgements

I would like to thank all of the people who encouraged and supported me over the past four years of my research.

First and foremost, I would like to thank my supervisors Luc Moreau and Mark Weal for giving me the benefit of their experience. I am also truly indebted for their great support and guidance throughout my PhD study.

I would also like to thank my friends and colleagues at the Intelligence, Agents, Multimedia (IAM) group for their support and friendship. My thanks also go to Thai friends, especially Thai PhD students, for their warm welcome, friendship and help during my time in Southampton.

I gratefully acknowledge financial support from the University of the Thai Chamber of Commerce.

Finally, I would like to thank my family - my parents and my sister - for giving me fully support. Without all of their love and their moral support, this dissertation would never have come into existence. Special thanks to my beloved wife, Kay, who gave me great encouragement and wonderful support.

Chapter 1

Introduction

The use of data-intensive applications that require continuous processing of real-time data streams has grown in prevalence and importance. A data stream is a real-time, continuous, ordered sequence of data items which can be submitted from different kinds of data source [60]. The size of data streams is usually unbounded and once each individual stream element has been processed it is eventually discarded or archived [10]. Because of the unique characteristics of data streams, the scientific community is adopting the data streaming technique for various kinds of applications that need an instantaneous response in order to support precise and on-time decisions. Examples of stream-based applications include sensor network applications [63, 69, 6], real-time location-based systems (GIS) [18], radio frequency identification (RFID) [74, 93], traffic management systems [52, 43], financial (stock) tickers [30, 137], and performance measurements in network traffic management [36, 35].

Major changes in daily life have been caused by recent advancements in micro-sensor and wireless communication technologies. The functionality and usability of sensor technologies enables several kinds of sensors to be deployed in a wide variety of environments (e.g. sensors embedded in a personal smartphone). The price of these devices is becoming cheaper and sensors are increasingly considered as commodity products that anyone can afford to buy. Although radio frequency identification (RFID) is one of the most important emerging technologies in this area, there are a variety of other technologies with various capabilities and costs (e.g. mote [103], SunSpot [123], and Lojack [89]). With the growth of sensor technologies, the time may come in the near future that every real-world object is tracked by several kinds of sensors which report its status or location in real-time. This will lead to a significant increase of wide range environment monitoring and control applications that operate over streaming data with high-volume and real-time processing requirements.

In all of the stream-based applications mentioned above, there are a number of significant requirements that these kinds of systems need to satisfy [120]. The first is that a stream processing system needs to process data streams in real-time or near real-time, and provide an instantaneous response in order to support a precise and on time decision. The second requirement is that a stream system must be able to process stream events on the fly without any requirement to store them. The third requirement for a stream system is that it needs to provide a mechanism to handle stream imperfections since stream data are often missing, delayed, or intentionally omitted for processing reasons. With these requirements, traceability of stream processing systems - the ability to verify and investigate the source of a particular output element - is extremely important. Stream processing systems that do not provide provenance information - the information pertaining to the process that led to result data [65] - can suffer from problems of traceability.

1.1 A problem of traceability

Imagine that in a radioactivity leak incident [115], an operator relies on a real-time mapping application (GIS) in order to manage and control the disaster. The information displayed on the GIS application is submitted by several sensors located near the scene of the incident in real-time at a rate of an event per second. At some point during the incident the operator found an anomaly in a number of results displayed in the GIS application. The operator anticipates that this problem has resulted from damage to, or malfunction of, sensors. The operator queries to find out which raw observations caused this anomaly and which sensors are responsible for sending the observations. Unfortunately for the operator, currently there is no provenance information available from the real-time GIS application. Therefore, the operator cannot easily determine why the unusual information is generated, which were the raw observations that led to the unusual information and which sensors contributed to that information being displayed in the GIS application. He would like to solve this problem as quickly as possible before something terrible happens.

This simple story illustrates the need for tracing individual results produced by existing stream processing systems. In such systems that require low-latency processing and instantaneous response, a mechanism for tracking provenance at the level of individual stream elements, which we refer to as *fine-grained provenance*, is very important. The existence of such functionality would allow users to be able to perform fault-diagnosis in the case of anomalies, to validate processing steps and to reproduce a particular result in the case of stream imperfections. By understanding the process that led to each individual result produced by a stream processing system, users can have confidence in the data that is the output from the system.

1.2 The importance of provenance

According to the Oxford English Dictionary, provenance is defined as: *the history or pedigree of a work of art, manuscript, rare book, etc.; concretely, a record of the ultimate derivation and passage of an item through its various owners.*

In the context of fine art, the term "provenance" refers to the documented history and chain of ownership of art objects. Provenance plays a strong and important role in the art community because museums, galleries, and collectors wish to avoid handling stolen art and wish to confirm that their art is genuine. Knowing the provenance of an artwork also allows collectors and curators to understand and appreciate its importance, and to verify and evaluate the history of that artwork for estimating its value. The provenance for a painting, namely Roses, by Vincent van Gogh is demonstrated in Figure 1.1. In this figure, provenance is presented in the form of its creation and history of ownership which can be used as a crucial part of understanding the importance of the painting and also verifying and investigating derivation history to guarantee that it is the original painting.

Roses, 1890
Vincent van Gogh (Dutch, 1853 – 1890)

```
Paul Gallimard [1850-1929], Paris, 1905.[1]
In the private collection of Bernheim-Jeune, Paris,
from at least 1917;[2] sold 1929 to (Alex Reid
& Lefevre, London).[3] acquired 1929 by W. Averell
Harriman;[4] Mr.and Mrs.  W. Averell Harriman,
New York; gift 1991 to NGA.
```

[1] According to J.B. de la Faille, *The works of Vincent van Gogh*, rev. ed., Amsterdam, 1970, no. F681. The painting is not described in Louis Vauxcelles, "Collection M.P. Gallimard," *Les Arts* (September 1908): 1-32.

[2] Lent by Bernheim-Jeune to a 1917 exhibition in Zurich.

[3] Letter dated 7 February 1929 from Reid & Lefevre to Bernheim-Jeune (Lefevre archives, Hyman Kreitman Research Centre, Tate Britain, TGA 2002/11, Box 228). Exhibited at Alex Reid & Lefevre in Glasgow in 1929 as from a great private collection.

[4] Lent by Marie Harriman Gallery to 1930 exhibition in Buffalo. In that same year the painting was exhibited at the Marie Harriman Gallery itself.

FIGURE 1.1: Provenance of the painting Roses by Vincent van Gogh [104]

The same idea of provenance can be applied to data items or result data generated within computer systems. In computer systems, applications generally produce result data during their execution time. To determine the provenance of result data, it is important to know details of the actual process responsible for the result data's generation. By recording and utilizing the documentation (documented history) of the process (or execution) that produced result data, users would be able to understand how a stream processing result was derived from a complex stream processing analysis, why critical decisions were made by a decision support system or how simulation results were determined by scientific simulation model. Therefore, in the context of computer systems, the provenance of a data item can be conceptually defined as *the process that led to that data item* [65, 97]. For the concrete view, the provenance of a data item is represented by suitable documentation of the process that led to the data.

Provenance provides benefits to computer systems in a number of ways [111, 112]. Provenance about a data product can be used to evaluate and estimate its quality for applications. It can serve as basic information (metadata description) for data discovery, so that users can search to find datasets based on their source data and the processing steps used to generate them. In addition, detailed provenance information (e.g. parameters, source data and operations used) can be utilized for supporting the reproduction of data products or analytical results in data analysis systems. Beside these beneficial uses of provenance, one of the most significant uses of provenance is that it can be utilized to trace the audit trail of result data. This use-case directly supports the requirement for traceability in computational systems. Provenance, in this context, offers detailed information that allows users to verify execution that took place in data generation; investigate the validity of intermediate data generated during execution time; and diagnose processing steps that are potentially the cause of errors.

1.3 Requirements for stream provenance tracking

We now present requirement use-cases for addressing fine-grained provenance tracking over data streams. We build upon the previous example using streaming data in the area of disaster management. The provenance use-cases regarding the disaster management scenario are described in order to explain provenance problems commonly found in this kind of context. Besides this scenario, other example applications such as weather monitoring [80] and sensor network [13] can also be used to illustrate our requirements.

1.3.1 A disaster management scenario

We consider a disaster management scenario such as the Port of Southampton off-site reactor emergency plan [115]. In this scenario a nuclear-powered submarine is docked at a port, the nuclear reactor of the submarine leaks and then finally explodes, spreading highly toxic radioactive material. Workers living in the vicinity of the explosion are injured and a wind-borne plume of radioactivity begins to spread out across a metropolitan region.

Following the radioactivity leak incident, emergency services operators rely on a real-time GIS application for disaster management to confine radioactive material, evacuate residents from a contaminated or hazardous area, and prepare to treat and decontaminate victims. In this incident, radiation and atmospheric sensors located near the scene of the explosion are automatically activated and other sensors in the surrounding areas are immediately activated as well. All sensors feed streams of their measurements to the real-time GIS application residing in a central server, which processes them automatically, integrates them with several kinds of geospatial data and finally graphically displays sensor locations, their readings, and radioactivity propagation prediction, in real-time in order to support emergency decisions. Furthermore, the sensor measurements received by the GIS application are further forwarded to an early warning system where predictions of a possible 'dirty bomb" event are made automatically based on sensor readings in real-time. The prediction results are used to control automatic reactor protection systems in order to prevent a further catastrophic explosion.

1.3.2 Provenance use-cases

Use-Case #1: Ability to trace individual stream events.

During the radioactivity leak incident a system operator pays particular attention to monitoring the occurrence of events in the GIS application. At some point the system operator observes an anomaly on the map display, where the level of predicted radioactivity displayed at a location significantly differs from its neighbours. The system operator questions whether he can trust individual stream events that display on the GIS application or not, considering the unexpected result. To understand this anomaly the operator would like to identify the raw measurements that caused this anomaly, verify the computation and processing steps that took place to produce the displayed result, and determine all the sensors and measurements involved in the event, to ascertain the validity of the prediction. This comprehensive and detailed analysis would allow the system operator to understand whether the predictive model or individual measurements could be the cause of an incorrect anomaly.

The above situation illustrates a common problem pertaining to the traceability of an individual stream event in stream processing systems. The first requirement use-case is identified as follows:

The ability to precisely trace individual stream events is necessary for stream processing systems in order to validate stream processing results. This ability allows users of stream processing systems to understand the computation that took place and the chain of reasoning that was used in the production of a stream processing result.

Use-Case #2: Ability to reproduce stream events.

At some point, due to a temporary loss of network connection of some atmospheric sensors deployed in the vicinity of the radioactivity leak incident, some sensors cannot submit their measurements to the real-time GIS application for a while. During this network outage period, a number of streaming events (sensor measurements) were sensed and stored locally at the sensors. However, after restoration of the network connection, these locally stored stream events were re-submitted. The system operator has detected a significant anomaly on the map display, where the average atmospheric temperature at a region close to the explosion area significantly differs from its neighbours. This anomaly affects the correctness of a prediction result produced by the radioactivity propagation prediction model as well. After receiving a report of the temporary network outage, the operator anticipates that this anomaly results from the absence of some stream events. To handle such a stream imperfection, the operator would like to replay stream execution for the period of missing stream events (the network outage period) in order to obtain accurate stream processing results and use these results to continue further processing for radioactivity propagation predictions. If provenance information pertaining to the past execution of the GIS application is provided, it would allow the operator to easily reproduce stream processing results by utilizing the provenance information together with a set of stream events that were suspended during network outage period. The ability to reproduce stream processing results would also allow the system operator to replay a portion of the past stream execution with new or modified values (input stream events) to obtain up-to-date stream processing results. In addition, it allows the original results generated by stream processing systems to be validated in order that the system operator can have confidence in the results.

This situation illustrates the need for this kind of system to reproduce stream events. Therefore, the second requirement use-case is identified as follows:

The ability to reproduce stream events or replay stream execution is required for stream processing systems to handle stream imperfections and also to validate the original results produced in this kind of system.

Use-Case #3: Ability to perform provenance tracking on-the-fly.

Because of extreme weather conditions and component degradation, some sensors deployed in the vicinity of the radioactivity leak incidence were damaged. This results in some malfunctioning sensors continuously submitting their faulty measurements into the GIS system. At some point during the radioactivity leak incidence, a system operator has received a report indicating that there is a second explosion in the nuclear reactor. The operator questions why this following explosion was not automatically detected by the early warning system and why the level of radioactive material shown on the map display was not classified as being potentially dangerous radioactive intensity levels. If the stream systems have support for a stream-specific provenance functionality that can be operated dynamically in real-time or near real-time, this would allow the operator to validate stream processing results (e.g. predicted events) in a timely manner, trace back incorrect information to its origin (raw sensor measurements) before particular critical decisions are made, and also continuously verify and display sequence of stream processing steps (processing paths) used to produce those results in near real-time.

This situation illustrates a common problem generally found in stream processing systems and more particularly in stream systems where their results are used for supporting on-time and critical decision making. Therefore, the third requirement use-case is identified as follows:

The ability to perform provenance tracking or execute provenance queries on-the-fly is necessary for stream processing systems in order to validate stream processing results in real-time or near real-time and deliver provenance query results instantaneously.

The three provenance use-cases described above illustrate problems generally found in stream processing systems. These use-cases introduce research challenges related to provenance tracking in this kind of system. We derive a strong requirement for precisely tracing the stream events that caused a given output stream event, which we refer to as *fine-grained stream provenance tracking.* The existence of such functionality would allow raw measurements (stream events) to be checked and stream processing steps to be reproduced, verified and validated.

1.4 Thesis statement and contributions

Our solution to the requirements of fine-grained provenance tracking in stream processing systems can be summarized in the following thesis statement.

A provenance model with a reverse mapping method that precisely captures dependency relationships for every individual stream element enables the problem of provenance tracking in stream processing systems to be addressed. It is designed to support a stream-specific provenance query mechanism, which performs provenance queries dynamically over streams of provenance assertions without requiring the assertions to be stored persistently.

This dissertation makes the following contributions to the state of the art:

1. A stream provenance model that allows for the provenance of individual stream elements, which we refer to as fine-grained stream provenance, to be obtained. This provenance model is based on the following key principles:

    - Dependencies between input and output events of stream operations can be expressed by means of a stream ancestor function that is defined for each stream operation. The key idea is that for a given reference to a particular output element, the stream ancestor function identifies which references to input events are involved in the production of that output. By composing all stream ancestor functions in a stream system, the complete provenance of an individual stream element can be determined.

- The stream provenance model defines the key elements and the structure of information that form the representation of provenance for individual stream elements. These include a provenance assertion - an assertion pertaining to provenance recorded by a stream operation for an individual stream element during processing time - and auxiliary information (e.g. stream topology, configuration parameters, stream operation parameters).

2. A provenance query method which utilizes the stream ancestor functions to obtain the provenance of a particular stream processing result. We demonstrate the expressiveness of the provenance query method by establishing that its query results can be used to reproduce the original stream processing results through the use of a replay execution method.

3. A stream-specific provenance query mechanism based on the idea of using the provenance service as a stream component. This query mechanism enables provenance queries to be computed on-the-fly without requiring provenance assertions to be stored persistently in a provenance store.

4. The performance characteristics of our provenance solution for streams.

    - It is shown that the storage overhead for provenance collection can be reduced significantly by using our storage reduction technique and the marginal cost of storage consumption is constant based on the number of input stream events (about 5 MB per component for 100,000 events).
    - Our provenance solution does not have a significant effect on the normal processing of stream systems given a 4% overhead for the store provenance assertions approach and a 7% overhead for the stream-specific query approach.
    - The amount of memory consumed for our stream-specific query approach depends upon the types of stream operations used and the size of data windows specified for each stream operation and the average time latency for the stream-specific query approach is about 0.3 ms per additional component.

1.5 Presentation Overview

This document is organized as follows.

Chapter 2 discusses the nature of stream processing systems and the characteristics of stream processing systems that differ from traditional data management systems. It also analyzes the state of the art for tracking provenance in different kinds of computational systems. In addition, based on this analysis, techniques used by our stream provenance solution are discussed and conclusions are drawn about the key attributes required for fine-grained provenance tracking in stream processing systems.

Chapter 3 defines a fine-grained provenance model including a data model and stream ancestor functions and a provenance architecture, designed to address fine-grained provenance tracking in stream processing systems. The stream ancestor functions - reverse mapping functions used to express dependencies among individual stream elements - are formalized using the Standard ML notation and thus are not bound to any particular technology or implementation.

Chapter 4 introduces the fundamental concepts of a provenance query mechanism for streams inspired by function composition. The definition of the provenance query mechanism is defined through the generic provenance query algorithm. Additionally, the chapter presents how accurate the provenance query mechanism is and how to validate the query results by establishing that the provenance query results can be utilized to reproduce original stream processing results using a replay execution method.

Chapter 5 introduces a stream-specific provenance query mechanism designed to address the practical challenges related to the unique characteristics of data streams. The key concepts and a programmatic specification of the stream ancestor functions designed for utilization with this kind of query mechanism are detailed. In addition, an example case study is presented to demonstrate how to apply the design of the stream-specific provenance query mechanism in a practical stream-based application.

Chapter 6 presents an evaluation of an implementation of our approach. It considers four different aspects of performance evaluation. Firstly, the storage overhead of the implementation when provenance assertions are stored persistently in a provenance store as the number of stream components increases. Secondly, the impact of provenance collection (system throughput) in a controlled environment. Thirdly, the memory consumption for a provenance service and finally the time latency for the stream-specific provenance query mechanism. Recommendations on the use of the implementation in applications are given.

Chapter 7 outlines various directions for future work and concludes the dissertation.

Chapter 2

Background

In this chapter we provide a review of the state of the art for determining provenance in different kinds of computational systems.

We begin this chapter by presenting essential background knowledge that supports our investigation into the problem of provenance tracking in stream processing systems. This background knowledge can be divided into two main parts. The first part describes the basic definition of data streams and the nature of stream processing systems - computational systems that continuously process transient streaming data and provide real-time or near real-time responses. We also discuss the characteristics of stream processing systems that differ from traditional data management systems. This discussion aims to give a clear overview of how stream processing systems work and what the unique requirements for this kind of systems are. The second part of the background knowledge presents the fundamental concepts of provenance and how the idea of provenance can be applied to computer systems.

After presenting the background knowledge, a review is given of various systems that provide provenance information and offer mechanisms to determine the provenance of data products. These systems are divided into three main categories: provenance in GIS and application specific systems, provenance in database systems and scientific workflow provenance systems. This division is based on the application domain that much research into provenance has been conducted. We also provide a discussion about whether techniques proposed in these previous research studies are suitable to address the requirements for fine-grained provenance tracking in stream processing systems. Furthermore, the related work that investigates and provides the solutions for tackling the problem of provenance tracking in stream processing systems is discussed. This discussion is given with respect to their effectiveness for addressing fine-grained provenance for streams.

The rest of the chapter is organized as follows. First we describe the basic concept of stream processing systems including the definition of data streams and the nature of stream processing systems. Next the fundamental concept of provenance as applied to computer systems is presented. After that a review of various provenance systems is given. We then discuss the related work in the context of provenance in stream processing systems. Finally the analysis conclusions are drawn and the chapter is summarized.

2.1 Stream processing systems

2.1.1 Definition of data streams

A data stream generally refers to information that naturally occurs in the form of a sequence of messages or data values. However, there are several precise and concrete definitions of data streams. Golab and Ozsu [60] define a data stream as a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is unfeasible to store a stream in its entirety. Babcock et al. [10] add that a data stream consists of four characteristics that differ from a traditional data model. First, the stream elements naturally arrive online, and second, for each stream element there has been no control over the order in which data element arrives at the system. Third, the size of data streams is usually unbounded, and finally, once an individual stream element has been processed it is eventually discarded or archived. We will consider a data stream as a continuous, ordered sequence of data items associated with a timestamp.

2.1.2 Characteristics of stream processing systems

There are many characteristics of stream processing systems that differ from traditional data management systems [10, 59, 60, 120]. These significant characteristics are summarized as follows.

1. Straight-through processing - In order to achieve low latency, a stream processing system must be able to process stream events without any requirement to store them. Because a storage operation such as committing a database record naturally causes a massive overhead, these kinds of operation should be avoided in order to minimize overhead in stream processing.

2. Process and respond instantaneously - To process high volumes of streaming data with low latency, stream processing systems must have an optimized engine that efficiently processes streaming data and generates output instantaneously. The execution produced by the stream engine must produce a minimal overhead. Enabling the tasks processed by a stream system to be completed by deadlines, satisfying the real-time processing constraint and guaranteeing predictable outcomes.

3. Continuous and long-running query - In stream processing systems, queries typically run over a period of time (the query life time). First, the queries are predefined or registered during an initialization period. After the data streams have begun, the queries are executed continuously by the stream processing system until the end of each queries life time. Furthermore, a modern stream processing system usually supports a high-level stream processing language - StreamSQL [12, 32, 9, 73]. StreamSQL generally extends the semantics of standard SQL by adding stream-specific operators.

4. Order based and time based operation - Stream systems have to provide stream-specific operations which are created especially for dealing with data streams. These operations are usually divided into two categories: order-based and time-based operations. An example of time-based operations is to perform queries over a five-minute time window. The use of order based and time based operation is an important mechanism to handle stream imperfection including delayed, missing and out-of-order data.

2.1.3 Comparison with traditional database management systems

Database management systems (DBMSs) are designed for supporting queries over finite stored datasets. Stream processing systems, e.g. Aurora [28], TelegraphCQ [29], STREAM [8], Borealis [1], are a new class of data management systems which have emerged to specifically support continuous and infinite streaming data. The important distinctions between traditional database management systems and stream processing systems have been widely summarized [10, 12, 59, 60, 120].

The first distinction is the data management model. Traditional DBMSs are designed for applications that require persistent data storage and support non real-time response. The DBMSs are generally used to store and manage large collections of data elements. Input data is stored persistently on the database, indexed, and then a variety of complex user queries may be executed. The results of queries reflect the current state of the database and contain a set of relative information that answers the user's queries directly. In contrast, stream processing systems are appropriate for applications that process transient streaming data and support real-time or near real-time response. It is impractical to operate on large portions of data or to process entire data elements multiple times like DBMSs. Stream processing systems generally process small portions of data stream elements over a period of time. Once an individual stream event has been processed, it is discarded or archived in order to achieve low latency. Moreover, queries in this kind of system are generally registered during an initialization period. The queries are executed when data arrives at the system and are also continuously executed until the queries expire.

The second distinction involves the kind of queries that each system supports. In a DBMS, only one-time queries are supported; queries that are evaluated once at a point-in time of the current state of the database [127, 10]. Stream processing systems support both one-time queries and stream-specific queries namely, continuous queries. Continuous queries are designed especially for processing streaming data. They are usually pre-defined during system initialization time. When stream elements are fed to the system, this kind of query is evaluated and executed. The result of a continuous query is produced over time and always reflects the streaming data feed to the system. Continuous query answers may be stored and updated as new data arrives, or they may be produced as data streams themselves.

The last important distinction is time-series information support. Because traditional DBMSs are not directly designed to support time-series information, it is very difficult to implement data stream applications which generally process high volumes of time-series data. Although there are some techniques in DBMSs that can be used to store this kind of information, these normally cause very expensive overheads and thereby dramatically slow performance. Examples of such database techniques include encoding the time-series information as data in normal tables and encoding time-series information in binary large objects (BLOBs). Stream processing systems are naturally designed for supporting time-series information. Several basic mechanisms of stream processing systems such as time-based operations and continuous stream queries have been especially designed for dealing with time-series information. Therefore, the stream system is more suitable for managing time-series information than DBMSs.

2.2 Provenance

2.2.1 A definition of provenance

Provenance - sometimes called "lineage" - is a term that has been well recognized in the study of fine art. It generally refers to the documented history or pedigree of a work of art. It is also used to describe the chain of ownership of an artifact. However, provenance can be described in various terms depending on the domain where it is applied. The Oxford English Dictionary defines provenance as: "the history or pedigree of a work of art, manuscript, rare book, etc.; concretely, **a record of the ultimate derivation and passage of an item through its various owners.**" We will consider provenance as the history or derivation of an object which is coming from a particular source to a specific state of an object.

In computer systems, the term “provenance” is defined and characterized by several research studies. Lanter [82], who developed the system for tracking provenance in GIS, characterizes lineage as information describing materials and transformations that are applied in order to produce the result data. In that work, lineage or provenance information is used in order to track how the analytical results in GIS were created. Besides data products, that study states that lineage information includes the associated processes that create the result data. In the context of databases, Buneman et al. [25] define provenance information as the description of the sources of data and the process by which it arrived in a database. They claim understanding provenance is crucial to the accuracy and currency of data in scientific databases. Moreau et al. [97] first propose a definition of provenance as a conceptual view. They then say within a computer system the provenance of a data item is represented by suitable documentation of the process that led to the data.

2.2.2 Provenance in computer systems

To make provenance information available in computer systems, the study conducted by Moreau et al. [97] recommends that computer applications need to be transformed into the new applications, called provenance-aware applications. Provenance-aware applications generally create particular data - process documentation - at execution time, in order to use the data to perform provenance querying, analysis and reasoning over provenance information. The provenance lifecycle begins when a provenance-aware application creates process documentation and stores it in a provenance store (a central storage component that offers a long-term persistent, secure storage of process documentation). Once process documentation has been recorded, the provenance of data results can be retrieved by querying the provenance store in the area of user interest. Finally, the provenance store and its contents can be managed and maintained by a system administrator.

Because many applications cannot create their whole process documentation at a single time, they normally generate them interleaved continuously with execution. The idea of decomposing process documentation has been proposed by Groth et al. [65] in order to record process documentation efficiently. The process documentation consists of a group of assertions, called p-assertions, asserted by the components of an application (actors). There are three types of p-assertion: *Interaction P-Assertion* is a description of the contents of a message which has been sent or received by an actor; *Relationship P-Assertion* is a description of how an actor obtained output data sent in an interaction by applying some function to input data from other interactions; and *Actor State P-Assertion* is a description of the internal state of an actor in the context of a specific interaction [97]. By recording these kinds of p-assertion, the flow of data in a process and input-output dependencies can be explicitly described. Furthermore, these p-assertions constitute a directed acyclic graph (DAG) which is a core element of provenance representation. For a specific data item, the DAG is used to indicate how a particular data item is produced and used.

In order to make applications provenance-aware, a software engineering methodology, Provenance Incorporating Methodology (PrIMe), has been developed [102, 101]. This methodology enables software developers to ensure that sufficient process documentation is captured so that queries on this documentation can satisfyingly answer users' provenance questions. Moreover, with PrIMe, developers can analyze and adapt applications systematically in order that the functionality offered by the provenance architecture can be fully exploited. The methodology is divided into three different phases: provenance question capture and analysis, actor-based decomposition, and adapting the application.

2.2.3 The open provenance model

Interest for provenance in computer systems is growing because provenance is recognized as a crucial tool for validating results produced in several kinds of applications (especially in database systems and scientific workflow systems). However, provenance representations utilized in various provenance systems are generally designed especially for specific purpose, and thus they can not be exchanged or shared with other systems. As a result, the *Open Provenance Model* (OPM) [98, 99] - a generic provenance model that is designed for supporting interoperability between provenance systems - has been proposed. OPM allows provenance information to be shared and exchanged between disparate systems and it also allows developers to develop software tools for operating on such provenance information. In addition, OPM is not only designed for supporting provenance of digital objects in computer systems but also for provenance of any "thing" (real-world objects).

In OPM, provenance of objects is represented by a causality graph which is a directed acyclic graph that is added with annotations (extra information) to provide a meaningful description of processes. Three types of nodes are provided: *Artifacts* denote an immutable piece of state that may have a physical embodiment in a physical object or a digital representation in a computer system (e.g. data items that input to or output from a particular application module); *Processes* represent an action or series of actions that are performed on artifacts and their execution results in new artifacts; and *Agents* represent contextual entities that enable, facilitate, control, or affect the process execution (e.g. a user who controls the execution of a workflow system).

To capture the causal dependencies between nodes (artifacts,processes and agents), five primitive categories of edges are classified in OPM. The causal dependencies expressed by OPM's edges include a process that used an artifact, an artifact was generated by a process, a process was controlled by an agent, a process was triggered by another process and an artifact was derived from another artifact. Each edge represents a causal dependency from its source (the effect) to its destination (the cause). In addition, edges can be further subtyped from these five categories by using annotations. This feature allows for OPM to express causal dependencies for specific contexts and provide meaningful exchange of provenance information. It is important to note that a provenance graph defined in OPM is designed to explain the derivation of artifacts in the past; it is not intended to describe processes or activities that will happen in the future.

Our stream provenance solution that will be presented in later chapters is also compatible with OPM, though it proposes a compact representation of it was-derived from edges. Using this type of dependency, we can assert that for each stream transformation (stream processing operation), a particular stream element E_2 was derived from another stream element E_1, and thus the input-output data dependencies (dataflow) inside a processing flow of a stream processing system can be captured.

2.3 Provenance systems

Depending on the domain where provenance is applied, different techniques and various types of mechanisms are used to capture and determine the provenance of result data produced in computational systems. The majority of work on provenance has been undertaken by the database and e-science communities. Provenance systems and techniques utilized in the context of databases have been described in several surveys [124, 125, 31]. In the domain of e-science, a comprehensive overview of provenance systems - system that provide lineage retrieval for scientific data products - is presented by Bose and Frew [21]. Similarly, two surveys of provenance related systems focusing primarily on scientific workflow systems are conducted by Simmhan et al. [111] and Davidson et al. [41]. In this section, we aim to review various systems that provide provenance information and offer mechanisms to determine the provenance of data products. We also discuss whether the techniques used by these systems adequately address the requirements for fine-grained provenance tracking in stream processing systems. These systems are divided into the following categories:

- GIS and application specific systems
- Database systems
- Scientific workflow systems

We will give a detailed review for each category of systems and then at the end of each category a discussion about provenance techniques used and conclusions will be provided.

2.3.1 Provenance in GIS and application specific systems

The study of lineage in Geographic Information Systems (GIS) was some of the first research in provenance. Lanter [82] investigated tracking the lineage of result data in GIS. Based on a layer-based GIS model, this study defines the lineage information of GIS data as the transformation of GIS layers from original to final products. Lineage information in this context includes the relationship between map layers and GIS operations applied to each layer. By developing the lineage information program (LIP) [83], command line GIS operations are intercepted and provenance of data can be retrieved by performing queries over lineage information stored in the meta-database (a storage for lineage information).

Another GIS system that introduces a mechanism for tracking provenance is Geo-Opera [4]. Geo-Opera is an extension of OPERA (Open Process Engine for Reliable Activities) [5] that provides a workflow management system for distributed geoprocessing. A process in Geo-Opera consists of a collection of geo-processing tasks linked by connectors used to establish the order of execution. To allow the system to keep track of provenance of data items, all input, output and intermediate data items resulting from executing a task need to be stored. Many of the ideas in Geo-Opera are extended from GOOSE [3] which uses data attributes of a system object to define dependencies between source and output data items. By querying the dependency of tasks, lineage information - a derivation of geo-processing results – can be obtained.

Spery et al. [117] investigated the use of lineage to manage the propagation of geographical updates in the context of corporate GIS databases. A lineage metadata model was proposed that describes the transformation of geographical objects [118]. In the model the transformation of geographical objects (i.e., create, modify or delete) is described by defining relationships, called *filiation links*, between geographical objects over different time periods. For example, a land parcel object which is divided into new parcels is defined as a parent object connected by filiation links to its child objects. Filiation links of all transformations are combined in order to construct a *filiation tree* which can be used to describe the derivation of individual geographical objects. This filiation tree enables users to perform historical queries and obtain lineage information pertaining to the transformation of geographical objects.

In the satellite image processing domain, the Earth System Science Workbench (ESSW) [48] was developed to track the processing of locally received satellite imagery. In order to track the lineage information of the output data products, Lab Notebook and Notebook tools are provided. The Lab Notebook collects metadata including processing steps, relationships between data objects and processing scripts from a researcher's workstation and records them into the database in XML format. After that, Notebook tools are used to generate directed graphs of experiment workflow and display the metadata for image processing data products. To address the challenge of data aggregation, Bose and Frew [20] extends the lineage concept presented in ESSW by introducing a solution for composing lineage information for custom satellite-derived data products. In this solution every data product and data transformation is paired with a lineage object, so lineage or provenance for a particular workflow invocation can be reconstructed by using parent/child relationships between lineage objects (lineage objects are used as a proxy for the relationships between actual data products). Furthermore, the Earth System Science Server (ES3) is proposed as the new version of ESSW [49]. ES3 introduces a novel approach to provenance capture and management that instead of explicitly specifying provenance together with workflow configuration model, automatically extracts provenance information from application's interactions at execution time. This provenance collection technique is similar to the techniques used in provenance systems that capture provenance information at operating system level [100, 130].

The Job Provenance service (JP) [44, 81] is designed to automatically track the provenance of the computations (jobs) that take place in large scale Grids. JP captures a permanent record of each registered grid job - complete information that is necessary to re-run the job (e.g. job description (JDL) and miscellaneous input files). Then, this record is utilized as information to support re-running jobs functionality. Users are allowed to add user-defined annotations (in the form of name-value pairs) to each job record. To obtain the provenance of grid jobs, JP provides a query interface that allows users to retrieve the provenance of grid jobs according to criteria specified on job records or user annotations.

Lineage studies in the area of GIS are mainly designed for dealing with specific purposes. The detail of the lineage metadata model and the architecture of the filiation tree [117, 118], for example, is based on a land-use cadastre case study. ESSW [48, 20] identifies some important issues that can be used as a guidance for the design of generic provenance systems such as the concept of how lineage information is composed for dissemination of data products. However, the techniques proposed in ESSW are still designed especially for capturing the processing steps of satellite imagery. Similarly, in Grid computing domain, the Job Provenance service (gLite Job Provenance) [44, 81] is tied solely to its computing environment (the gLite Workload Management System [45]). Therefore, they are not general enough to be applied to applications in other domains including stream-based applications.

2.3.2 Provenance in database systems

Considerable research efforts have been made by the database community to address the data provenance (sometimes called lineage) problem. This problem can be summarized as: given a specific tuple in an output database, identify tuples in a source database that contributed to it. Tan [124, 125] classifies research studies on data provenance into two distinct approaches: in the lazy (or non-annotation) approach, provenance information is generated on demand, by means of a query, only when requested [136, 37, 38], whereas in the eager (or annotation-based) approach provenance information is propagated at runtime [26, 33, 17, 50, 51, 61, 62, 57, 58]. We now discuss them in turn.

To solve data lineage problem and improve database visualization systems, Woodruff and Stonebraker [136] proposed a data lineage technique called *weak inversion*. Given a particular output item, weak inversion functions are used to regenerate input data items that produced the output. However, the answer returned by this function is not guaranteed to be perfectly accurate. Therefore, a separate verification function is required to examine the answers produced by weak inversion. The drawback of this technique is that in several cases it is not possible to define inversion functions due to fact that particular functions cannot be invertible. Moreover, weak inversion functions and their corresponding verification functions need to be registered by a user who creates a new database. A subsequent research study conducted by Cui and Widom [37, 38] overcomes this limitation by introducing a lineage tracing algorithm. This generic algorithm automatically generate lineage data through analyzing the view definitions and algebraic structure of queries. Based on the tracing algorithm, several schemes for storing auxiliary information are also presented to improve the performance of lineage tracing in data warehouses. The purpose of this study is to provide an infrastructure for data warehousing systems that enables users to "drill through" the lineage of a data item in order to see the original source data that contributed to the data item.

Bunemann et al. [25] formalize the data lineage problem and draw a distinction between two types of provenance: "why-provenance" and "where-provenance". The type of provenance studied by Woodruff and Stonebraker [136] and Cui and Widom [37] is essentially why-provenance, which tries to determine what tuples in the source database contributed to an output data item. Where-provenance, on the other hand, aims to identify locations in the source database from which the data item was extracted. Based on these types of provenance an annotation-based approach called *propagation rules* [26] has been proposed to address where-provenance. In this approach, annotations associated with tuples in the source database can be propagated to the output database based on where data is copied from. The forward propagation rules for each relational operator are defined to determine how annotations are carried from source to output database. With this technique dependency relationships between locations of data in the input and output database can be expressed.

The idea of annotation propagation is further extended by DBNotes [33, 17] - an annotation management system for relational database systems. In DBNotes, an extension of a fragment of SQL, namely pSQL, was introduced. The use of pSQL allows users to specify how annotations should propagate from source to output databases. DBNotes provides three different types of annotation propagation scheme which support different propagation purposes. In its "default" propagation mode, annotations are propagated based on where data is copied from (where-provenance). By examining the annotations automatically propagated through a SQL query, the provenance of a piece of data through a sequence of query transformation steps can be easily determined. However, in both DBNotes [33, 17] and propagation rules [26], each provenance-related annotation can only be attached to a particular value of specific attribute. To overcome such a limitation, Geerts and Van Den Bussche [51] proposed an extension mechanism called "Colors and Blocks" that allows each provenance-related annotation to be associated with both a single value of attributes and a set of values (or multiple values of attributes). This mechanism is based on the idea that annotations are treated as first-class citizens of the database along with tuples and attribute values. In this study, a new algebra - namely color algebra - that includes common relational operators (e.g. projection and selection) properly re-defined and new operators designed to account for the colors and blocks approach is introduced to query both attribute values and annotations. MONDRAIN [50] - a prototype implementation of this annotation mechanism in relational DBMSs - is also presented. This study claimed that using the colors and blocks approach, the existing schema of the database is not required to be re-structured (only extra tables for storing annotations need to be added).

Building on the ideas from [37, 38, 25], Trio [135, 2] - a database system that manages data, accuracy and lineage - proposed an integrated technique (combining both lazy and eager approaches) to capture lineage information in database systems. In Trio, lineage is recorded at the granularity of tuple level. Lineage information is generated automatically whenever a TriQL [16] - an extension of SQL introduced especially for dealing with uncertainty and lineage information - is executed. The idea similar to annotation propagation in DBNotes [17] is applied to propagate tuple identifiers from source tables to an output table. However, because Trio stores only one level of lineage for each database tuple (it means only direct ancestors are recorded for each tuple), in order to obtain the complete provenance of a particular tuple, Trio provides recursive traversing lineage algorithm as part of TriQL. So users can utilize build-in functions of TriQL to perform queries over lineage information in order to obtain the complete provenance of particular tuples they are interested in.

Another study proposed a technique to capture provenance in database systems called *provenance semirings* [62]. This study not only tries to address the "why-provenance" problem, but also identifies the need to understand "how-provenance" which describes how the input data leads to the existence of the output data. In this technique, provenance-related annotations in the form of variables are attached to relational tuples in the source database. When a query is executed, variables of relevant tuples are propagated and form polynomials with integer coefficients for the output tuples. In [61], an application that applied the technique of provenance semirings is described in the context of collaborative data sharing. Using a semiring of polynomials, provenance representation of output data items in database systems can be captured and thus the problem of how-provenance can be addressed.

Glavic and Alonso [58] demonstrate the disadvantages of non-relational provenance representation used by existing approaches [37, 38, 135, 50]. In these previous approaches provenance information is recorded and accessed using a different data model than the actual data (relational data items used as the input and the output of SQL queries in the database). To tackle this limitation, the solution called Perm system [57] (Provenance Extension of the Relational Model) which represents provenance as a single data model (relation) containing both original query result tuples and contributing tuples (tuples used in the production of the original query result tuples) is introduced. Perm transforms an original query (SQL query) q into a provenance-related query q' by rewriting relational operators of q in order to propagate provenance information alongside query results. The advantages of using the same data model and query rewriting mechanism is that rewriting SQL query can be operated and optimized by standard relational database techniques. The Perm system focuses on "why-provenance" problem as it is claimed that it better addresses their user requirements than "where-provenance".

Advances in provenance tracing for database systems encourage applications in other domains to adopt provenance techniques used in this context. Several database provenance solutions [136, 37, 38, 57, 58] are mainly focused on the why-provenance problem, which tries to answer the question: "why is a piece of data in the query output". Recent studies in this area [33, 17, 135, 2, 50, 51, 61, 62] extend the previous studies by introducing enhanced solutions for where- and how-provenance in order to provide better understanding of query results. Although research studies in this area mainly focus on capturing SQL-based transformations, which is different from our intention that tries to describe general stream transformations, some fundamental provenance tracing concepts can be applied into our provenance solution.

Based on literature in the context of database systems, our provenance solution for streams aims to address a form of "why-provenance" for stream processing systems since it aims to identify a minimal set of input stream elements used in the production of a particular output stream element. Our provenance approach combines both eager and lazy techniques, eagerly propagating minimum provenance-related information at runtime, and relying on provenance queries to extract fine-grained provenance, lazily, on demand. Instead of using a simple or non-structural annotation, our provenance solution for streams uses a structural annotation (event key). We show that this type of provenance-related annotation is more suitable for expressing dependency relationships between input and output stream elements.

2.3.3 Scientific workflow provenance systems

Much of the research into provenance has come in the context of domain specific and scientific workflow management systems. A workflow management system (e.g. myGrid/Taverna [105, 71], Vistrail [47], and Kepler [22]) is a computer system that manages and defines a series of independent tasks within scientific data-intensive analyses in order to produce a final output (data product). In such systems, tasks are chained together and each task takes input data from previous tasks. A workflow specification is generally represented as a graph, where nodes represent data processing tasks (or data transformations) and edges represent data flows between tasks. Tan [125] describes the general characteristics of provenance techniques used in workflow management systems. In these kind of systems data processing tasks are treated as "black boxes" [41]. Details pertaining to data transformations are typically hidden, and only input-output data, dependencies between input-output data and short auxiliary information (e.g. description about the software used) are recorded. Hence, the workflow provenance is classified as "coarse-grained provenance" [125]. We now discuss provenance techniques used in workflow management systems in detail.

In the life science domain the Taverna project [105, 71] has developed a tool for the composition and enactment of bioinformatics workflow. To allow scientists to understand how results from experiments were obtained, Taverna provides support for provenance tracking. In Taverna, provenance information, which identifies the source and processing of data, is collected by recording metadata information and intermediate results during the enactment of a workflow. The provenance information recorded includes technical metadata explaining how each task has been performed. In addition, the information regarding types of processor (tasks), status (current state of the processor), start and end time, and a description of the service operation used, are also recorded.

REDUX project [14] is another study that proposes a mechanism for capturing provenance information in scientific workflows. This study argues that a single representation of provenance cannot satisfy all existing provenance queries used in this kind of systems. So, a provenance model that supports multiple levels (four-layers) of provenance representation is introduced [15]. The first layer captures the abstract description of workflows consisting of processing tasks and relationships (links) among them. The second and third layers capture information provided at execution time of the workflow including input data, parameters supplied at runtime, etc. The final layer of the provenance model captures runtime specific information such as start and end time of individual tasks executed and status code. Using the multi-layered provenance model allows users (scientists) to comfortably deal with complexity and size of provenance information. Users can navigate from abstract layers into lower detailed execution layers, depending on the amount of provenance information needed to validate particular processing results. In addition, the multi-layered provenance model offers users to control over information they wish to share and retain for their experiments and the model also supports the reproducibility of workflow results.

Vistrail [47, 27] builds on the similar idea of multi-layered provenance representation presented in REDUX [14, 15]. In this study a "Change-based provenance" mechanism has been proposed to capture provenance information for the evolution of the workflows and their data products [109]. Vistrail not only records intermediate results produced during workflow execution, but also records the operations (actions) that are applied to the workflow. In this system, the modification of workflows, for example adding or replacing modules (data processing tasks), deleting modules and setting parameters, is captured by tracking the steps followed by a user. Intermediate data products generated by the workflow are also recorded. By storing both data products and modification of workflows, Vistrial can ensure reproducibility of experimental processes and provide support for the systematic tracking of workflow evolution.

Following the idea introduced in REDUX [14] and Vistrail [109], Kim et al. [77] proposed the three stages (layers) provenance mechanism for capturing provenance information in large-scale scientific workflow systems. This approach is implemented in the Wings/Pegasus framework [42, 53, 54]. For the first two layers, "application-level provenance" consisting of a definition of reusable workflow templates - abstract specification of workflow describing types of processing tasks and data flows among them - and input data (of workflow instance) defined by Wing is recorded. The final layer captures "execution provenance" - information gathered during the processing of workflow - which includes intermediate data, detailed information about transformations, performance information, etc. In addition, workflow refinement information, which describes refinement process automatically performed by Pegasus to make the workflow execution efficient, is also captured at the final layer of this provenance mechanism.

Karma provenance framework [110, 113] was developed to collect and query provenance of data products produced by scientific workflows executed in a service-oriented architecture (SOA). Two forms of provenance are captured in Karma [114]: *Workflow provenance* - information generated by a central workflow engine - describes workflow's execution and associated service invocations; and *Data provenance* - information generated by each service in a workflow - describes the derivation of a data product, including input data sources and transformations used to generate that data product. The interesting feature of Karma is that provenance information recorded can be grouped and ordered along four dimensions: the execution level (e.g. workflow and service), the location of the workflow component involved, the time at which the computation took place, and the dataflow used in the computation of data products. These different layers of abstraction allow users to conveniently retrieve provenance information at the level of granularity in which they are interested.

In the area of scientific data management, the collection-oriented modeling and design (COMAD) provenance framework [92, 7] presented an annotation-based approach specifically designed to deal with collections of data. This framework is implemented within Kepler [90] for representing the provenance of scientific workflows. In this model each COMAD module (processing task) takes collections of data as an input, accesses particular collections, and produces output collections by adding new computed data to the data structure it received. To allow the system to trace provenance of scientific data products, output collections are embedded with a metadata annotation containing explicit data dependency information. This metadata annotation is used to describe the derivation of output objects computed in a scientific workflow. In addition, as described in [23, 24], this system introduced a solution to minimize the provenance information recorded for a workflow run by allowing provenance annotations on collections to cascade to all descendant elements.

Another provenance study in the area of scientific workflow is Provenance Aware Service Oriented Architecture (PASOA) [65, 64, 67]. PASOA investigated the concept of provenance and built an infrastructure for recording and reasoning over provenance in the context of e-Science. PASOA is mainly designed for supporting interactions between loosely-coupled services. In this study the idea of decomposing process documentation has been proposed in order to record provenance information efficiently. Each part of the process from the whole process documentation is defined as a p-assertion. By capturing p-assertions regarding the content of messages (interaction p-assertions) along with causal relationships between messages (relationship p-assertion) and the internal states of services (service state p-assertions), the documentation of processes that led to a result can be recorded. Based on this idea, the Provenance Recording for Services (PReServ) [66] software package has been developed. This implementation allows developers to integrate process documentation recording into their applications.

To conclude, the use of provenance in scientific workflow systems differs from that in other application domains. Provenance is not only used for describing the origin of result data, but also for troubleshooting and replaying workflow execution. As described in several surveys [125, 41], almost all provenance solutions detailed above follow the "conventional" model of provenance in scientific workflow systems that captures only input/output data products of transformations (processing tasks), and causal dependencies between them. Processing tasks of workflows are treated as "black-box transformations" for which only short descriptions or links are recorded. Recent studies in this domain [109, 77] extend the conventional model by introducing novel solutions that not only dynamically capture dependencies between input/output of transformations during execution time, but also record information describing the modification of workflow specification. In addition, because of complex computation and massive amount of data used in scientific workflows, provenance information of a data product may be relatively large and difficult to understand. To allow users to deal with the complexity and large size of provenance information several provenance solutions [14, 109, 77, 114] presented the idea of multi-layered provenance representation. Using the idea of multi-layered provenance representation allows for users to focus on provenance information at the level of granularity they are interested in and thus they can better understand and conveniently consider the provenance of results.

Our provenance solution for streams extends the PASOA provenance mechanism [67]. In our model, each stream operation is treated as a "grey box" [22] - black-box modules (stream operations) to which are provided additional provenance annotations describing input-output dependencies. So, provenance information can be collected based upon dependencies between input and output elements of the stream operation. However, because workflow provenance systems (e.g. PASOA) need to store all dependencies and intermediate data objects, the amount of information recorded can potentially cause a storage burden problem when dealing with high volume data streams. Therefore, one of the practical requirements for our provenance solution for streams is to find an enhanced technique that can address this storage problem. The idea of how our provenance solution can address the practical storage problem will be described in the next chapter.

2.4 Provenance in stream processing systems

The usefulness of provenance in a certain domain is linked to the granularity at which it is collected [111]. As discussed in [125], there are two granularities of provenance considered in the literature: course-grained and fine-grained provenance. Course-grained provenance refers to process documentation captured through processing of a workflow. In this granularity, details regarding data transformations are typically hidden. In contrast, fine-grained provenance provides relatively detailed documentation which elucidates the derivation of a data item that is in the results of a transformation step. This particular granularity of provenance is of interest to the database community for capturing SQL-based transformations. In this section, we apply the terms of provenance granularity to stream provenance literature. The literature regarding stream provenance techniques can be divided into two main categories based on the granularity of process documentation these techniques achieve. *Course-grained stream provenance* refers to provenance information that is captured at the level of streams or sets of stream events. On the other hand, *fine-grained stream provenance* refers to provenance information that is collected at the level of individual stream events.

2.4.1 Coarse-grained stream provenance

In the area of distributed stream processing, Vijayakumar [133] defines provenance of data streams as information that helps to determine the derivation history of a data product, where the data product is the derived time bounded stream. To address the provenance problem an information model and architecture for capturing and collecting stream provenance has been proposed [131, 132, 86]. The provenance collection begins with recording the static provenance – descriptions of input streams and pre-defined continuous queries - during system initialization time. After that, the provenance information (dynamic provenance) is recorded only when something changes in the input stream environment during processing. The change events include rate and accuracy changes. Because the change events have attached timestamps, it can associate them with the set of events in the output stream that is affected by the change events. With this collection model this research confirms that the overhead for provenance collection is minimal.

A similar coarse-grained technique for recording provenance has been used in sensor archive systems [39, 79]. In this technique, processing modules of archive systems have to employ standardized logging methods to capture only important events during their processing. Each log record contains a timestamp, the module identifier and the error code and message. By storing all log records in a metadata database which is separate from a sensor database, provenance and data quality of sensor data can be tracked. Another study conducted by Ledlie et al. [84] proposed the idea of collecting provenance of sensor data - the history of how and where sensor data came to be - and utilizing it as an index for identifying data items in sensor data storage. Although the detail regarding how to capture provenance information is not described, the overview concept, including the granularity for provenance recording, is discussed. This study recommends that provenance information needs to be recorded at the level of tuple sets or collections of sensor readings, and it should be grouped by a particular time period in order to achieve efficient performance. For example, provenance information is recorded for all sensor data over the span of one hour.

The studies pertaining to coarse-grained stream provenance have proposed ideas to record provenance by identifying streams or sets of stream events and processing units as the smallest unit for which provenance is collected in stream processing systems. However, the level of granularity for capturing provenance information in these models is not detailed enough to address our problem. To deal with our requirements provenance information collected in this kind of system needs to be recorded at the level of individual stream events in order that data dependencies for each individual stream event in a particular processing step can be examined.

2.4.2 Fine-grained stream provenance

Several studies on fine-grained provenance for streams have been undertaken in the context of sensor data management. Park and Heidemann [107] proposed an annotation-based approach called *tuple-level link* to capture provenance information of sensor data that is processed and republished in sensornet systems [108]. To allow any user to follow it back to the original source data, each sensor record is embedded with a URI compatible link called a *predecessor link*. The predecessor link is encoded with the location of the source repository and a table at that repository, the search used to retrieve the data from that table, and a timestamp. By resolving the predecessor link for each sensor record, sensornet's users can examine dependencies between input and output data in which they are interested. Furthermore, this study also provides a compression technique called *incremental compression*. The main target of this compression technique is to reduce link size and provide reasonable storage costs for provenance collection. Although the tuple-level link approach offers efficient storage consumption and can express data dependencies for individual stream elements, there are still some limitations. The predecessor link is designed especially for supporting the process of transforming on-line sensor data in sensornet systems and is not general. Therefore, it is difficult to apply this technique to stream-based applications in other domains.

Lim et al. [88] proposed a systematic method for assessing the trustworthiness of data elements (stream events) in sensor networks. In this study data provenance is used as a crucial information for computing "trust scores" - an information associated with each stream element provides an indication of the trustworthiness of the stream element. Two types of provenance are captured [87, 40]: *the physical provenance* (represented as a network path that each stream element passes through) describes where each stream element was produced and how it was delivered; *the logical provenance* represents the semantic meaning of stream elements in the context of a given application (e.g. sensor category). Similar to other annotation-based approaches, the physical provenance is delivered along with stream elements during execution time and it is the type of provenance exploited to compute trust scores. The logical provenance is utilized for grouping provenance data into semantic events that users are interested in. The drawback of this technique is that only selection and aggregation are the transformations (or operations) that this study focuses on. Considering several kinds of stream transformations used in existing stream systems (e.g. Windowed operations [28, 1] and Binary join [8]), the technique proposed in this study probably fails to address provenance problems in general-purpose stream processing systems. In addition, it is claimed that a simple provenance representation (network path/tree of sensor nodes) currently used in this study cannot express data dependencies for applications that have complex processing flows (e.g. split and merge or loops in data flows) [87].

The study by Wang et al. [134] argued that the annotation-based approach, which is typically designed for transaction-oriented systems, cannot satisfy the unique requirements for recording provenance in high-volume and continuous streams. Therefore, a model-based provenance solution, called *Time-Value-Centric* (TVC), was introduced to support stream processing in medical information systems [19]. In the TVC model, relationships between input and output data stream elements can be described in terms of some invariants. This model supports three primitive invariants for dependency specification: time, value and sequence. *Time* is a primitive that captures dependencies in terms of the time window that the output element depends on. *Value* captures dependencies in terms of the predicate of the attributes of the input elements. The last primitive, *sequence*, captures dependencies in terms of the sequence number of arriving elements. The model assumes that all elements of all data streams are persisted and each data element recorded is tagged with a unique timestamp. By composing these primitive invariants, dependencies between input and output stream elements for each stream transformation can be explained. The following study, conducted by Misra et al. [96], extends the previous study by identifying a practical challenge pertaining to a storage problem. Because every stream element and their intermediate result data needs to be stored, this persistence of high volume stream events potentially results in a storage problem. This study proposes a technique called *Composite Modeling with Intermediate Replay* (CMIR) to eliminate this storage problem. A group of stream processing units involved in stream processing are aggregated into a virtual group, called a *virtual PE* (Processing Element). Only streams that act as input and output streams of the virtual PE are persisted and defined dependencies. By applying CMIR, stream processing systems do not require the persistence of all intermediate streams, thereby reducing storage costs consumed by provenance recording.

MediAlly [34] is a recent health monitoring system that applies a Time-Value-Centric (TVC) model to provide fine-grained provenance tracking functionalities. To support energy-efficiency, MediAlly adopts an Activity Triggered Deep Monitoring (ATDM) paradigm [95] as an energy saving mechanism, where data streams (e.g. ECG - the electrical activity of the heart) are collected and relayed to a central server only when monitored context information (e.g. locations (from GPS), personal information) is evaluated and satisfies given predicates. In this system, TVC is utilized as a backbone of provenance sub-system. The interesting point of this study is that instead of dealing with actual medical sensor data streams, TVC is used to capture dependency relationships between elements of contextual information streams. However, to the best of our knowledge, due to the fact that all elements of all related contextual information streams need to be stored persistently as in the previous study [134], the practical limitation pertaining to a storage problem remains unsolved.

Another study by Huq et al. [72] identified the problem of maintaining fine-grained data provenance for streaming data. In stream processing systems, the use of sliding window operations has become very common resulting in a single stream element contributing to many output stream elements. As consequence, to facilitate fine-grained provenance tracking, a single stream element needs to be stored multiple times depending on the overlap of sliding windows. This potentially results in a storage problem. To reduce storage costs for fine-grained data provenance, a temporal data model inspired by bitemporal model [78] is proposed. In this model, a timestamp utilized as a database version number is added to each stream element recorded. This is to ensure that a query on a particular database state in the past achieves the same result regardless of the query execution time. Then each individual stream element recorded is also embedded with temporal attributes: *valid time* representing the time that each stream element is generated; *transaction time* representing the time each stream element is inserted into the backend database. Using the temporal attributes allows users to identify input stream elements used in the production of output stream elements, thus the provenance of each individual stream element can be retrieved.

The TVC model [134, 96] is one of the most recent fine-grained provenance solutions that provides the ability to express data dependencies for individual stream events. Nevertheless, this model still has some limitations, since it can fail to identify precisely input stream elements that are used in the production of an output. The use of only “value” primitive to identify all past input elements contributing to a particular output element is an example of this limitation [75, 76]. In this case some irrelevant stream elements that are not involved in the production of the output can probably be included in the results resulting in this primitive fails to provide exact provenance query results to users. Another limitation of this model pertains to storage consumption for provenance collection. Because all intermediate stream elements need to be stored for computing the provenance of an output element, the persistence of high volume stream events potentially results in a storage burden problem.

Although the subsequent study [96] introduced the solution (CMIR) to address the storage problem, there is still a question about this model as to how to compose different types of primitive invariant in order to derive dependency relationships for virtual PE. The other similar model [72] has the same limitation as TVC due to the fact that this model is required to store all intermediate stream elements for fine-grained provenance tracking as well. Therefore, to address these limitations, we introduce a finer-grained provenance solution that precisely captures the provenance of every individual stream element without requiring every intermediate stream elements to be stored persistently. The detail of our provenance solution and the idea of how the storage problem can be addressed will be presented in the next chapter.

2.5 Analysis Conclusions

We have presented a wide range of systems and models that address the problem of provenance in computational systems. We also have investigated related studies on provenance tracking in stream processing systems. From our analysis, we have come to the following five key conclusions.

1. *The “why-provenance” is the primary type of provenance on which provenance solutions for stream processing systems should be focused.*

   In Section 2.4, we have discussed several related studies on provenance tracking in stream processing systems. Based on this discussion we found that almost all studies aim to provide a type of provenance similar to why-provenance [25], which tries to explain the presence of individual stream processing results generated. For example, TVC model [134, 96] and subsequent studies [72, 34] try to identify raw sensor data that contributed to an analytical result in medical information systems. This encourage us to conclude that the why-provenance problem is a crucial provenance problem in this context.

However, we noted that the existing solutions for fine-grained provenance tracking in stream processing systems still have some limitations as they fail to provide exact provenance query results to users [134, 96] and the storage problem, which results from the persistence of high volume stream events, still remains [96, 72, 34]. Therefore, we conclude that the why-provenance problem is not yet completely solved by existing stream provenance solutions and it should be exploited as the primary type of provenance that our provenance solution aims to address.

2. *Provenance solution for streams processing systems should combine both lazy and eager approaches to tracing data provenance.*

   As described in [124, 125], existing solutions for tracing data provenance can be classified into two distinct approaches: the lazy and eager approaches. The lazy approach computes the provenance of data when needed by using a query or an inversion function. On the other hand, in the eager approach, provenance information is computed and carried along with data at execution time.

   Our provenance solution for streams blurs the distinction between lazy and eager approaches. It propagates structured information (structured annotation) at runtime, which is exploited for retrieving provenance by queries, on demand. Combining these two distinct approaches offers several advantages. First, the limitation of inversion functions [136] - some stream transformations cannot be invertible - can be eliminated. Second, we can reduce additional overhead resulting from computing the full provenance of each stream element at runtime (in our approach structured annotations are just propagated with stream elements and the provenance of each stream element is computed later by using a query). Third, we can support a variety of queries performed over provenance information on demand.

3. *The idea of capturing input-output dependencies of data transformations should be considered as the significant provenance collection concept for stream processing systems*

   In Section 2.3.3, we have described several scientific workflow systems. In such systems, only information about data products and input-output dependencies of data transformations is generally captured; the detail related to data transformations are typically hidden. This idea in which data transformations are treated as "black boxes" is classified as the conventional model for provenance in workflow systems [41, 125].

   General-purpose stream processing systems and workflow management systems have similar characteristics where the specification of both systems can be represented as a graph (a set of interconnected nodes where each node representing a data transformation or a stream operation). Several provenance solutions for streams apply the idea of capturing input-output dependencies of stream transformations as well [134, 96, 88, 72, 34]. In our provenance solution, each stream operation is treated as a "grey box" [22] in which stream operations - black box modules - are provided with additional annotations describing input-output dependencies. Provenance of individual stream elements can be retrieved based on dependency relationships between input and output elements of the stream operation.

4. *Timestamp should be utilized as the key element for expressing dependencies between data items in a data stream.*

   Based on our definition of streams summarized from [60, 10], each stream element in a data stream is associated with a timestamp. In this context the timestamp can be utilized as a unique identifier for each individual element in a data stream.

   Almost all solutions for fine-grained provenance tracking in stream processing systems apply this concept by adding a timestamp to each individual stream element stored in a data repository [134, 107, 72, 34]. Our provenance solution for streams adopts the idea of using timestamps as well. The key element of structured information (event key) that we propagate at runtime is a timestamp. By utilizing timestamps as a unique identifier we can exactly identify all input elements that are used in the production of a particular output element and thus the provenance of an individual stream element can be retrieved.

5. *The practical challenge pertaining to the persistence of high volume of stream events needs to be addressed.*

   In Section 2.1.2, we described some unique characteristics of stream systems that differ from traditional data management systems. One of significant characteristics is that a stream processing system must be able to process stream elements without any requirement to store them. Using the storage operation naturally causes a massive overhead to stream processing and storing all high volume stream events potentially causes a storage burden problem.

   Several studies in the context of stream provenance have recognized the need for stream provenance systems to address this storage problem [131, 107, 96, 72]. Therefore, our provenance solution for streams introduces an enhanced mechanism to reduce the storage requirement. It also supports a stream-specific query mechanism that can perform provenance queries on-the-fly. By using this query mechanism, the storage burden problem can be addressed as provenance information is not required to be stored persistently.

2.6 Summary

The literature demonstrates that provenance is a crucial tool for confidence in the results produced by different kinds of application. Research into database systems shows that the idea of provenance tracing can be successfully implemented and transferred to working systems, for example lineage tracing in data warehouses. The use of provenance in database systems also provides an inspiration for applications in other contexts to apply and extend the provenance capturing techniques for specific purposes. Our provenance solution is also inspired by the idea of annotation propagation [26, 33, 17, 135, 62]. We use a structured annotation (event key) as a reference to each individual stream element and we propagate the annotation along with each stream element in order to use it for describing dependencies among stream elements later.

Due to the characteristics of input-output dependencies of workflow systems being quite similar to that of stream processing systems, the scientific workflow provenance literature encourages us to apply some efficient provenance techniques into our target applications. The idea of capturing input-output dependencies by hiding details regarding data transformations is quite suitable for stream processing systems. This inspires us to design our provenance model for streams based on the PASOA provenance mechanism [67]. The literature in the context of provenance in stream processing systems indicates that the characteristics of high volume and high generation rates of data streams is a significant issue. Coarse-grained provenance solutions try to avoid the storage problem resulting from this unique characteristic by recording only provenance of important events in stream processing. However, the level of granularity for capturing provenance information in these solutions is not detailed enough to satisfy our requirement use-cases. Fine-grained provenance solutions which capture provenance of every intermediate stream event are more suitable for addressing our problems.

To sum up, our provenance solution, which is based on the key critical analysis conclusions presented in Section 2.5, improves over the state-of-the-art in multiple ways. It defines a fine-grained notion of provenance for streams similar to why-provenance, which can explain the presence of individual elements in streams. In doing so, it identifies a class of stream operations for which such fine-grained provenance can be determined. It blurs the distinction between lazy and eager approaches [124, 125], since it propagates structured information at runtime, which is exploited for retrieving provenance by queries, on demand. It introduces an enchance mechanism that reduces the storage requirement compared to a related stream provenance approach. Furthermore, to satisfy unique requirements of stream processing systems, it introduces a novel stream-specific query mechanism that can perform provenance queries on-the-fly. This mechanism allows the problem of the persistence of all intermediate stream elements to be solved.

Chapter 3

A provenance model for streams

At the beginning of this dissertation we outlined the requirement for provenance support in stream processing systems. This requirement is to track provenance information at the level of individual stream events so that data dependencies for each individual stream event in a particular processing step can be examined. In this chapter, we begin to address this requirement by introducing a provenance model for stream processing systems. The aim of this provenance model is to describe the fundamental concepts of provenance representation in stream processing systems indicating how provenance of individual stream elements can be represented and how dependency relationships among stream elements can be precisely expressed. We also describe a provenance architecture for stream processing systems that is designed to comply with the stream provenance model.

To provide a concrete and precise solution supporting the stream provenance model, it is necessary to understand and to answer several questions regarding how a particular stream operation works. To do this, we define programmatic specifications for stream operations. With these programmatic specifications we can precisely describe how the output element for each stream operation is produced in terms of input elements. We then use these specifications to define *a stream ancestor function* for each stream operation. The purpose of this stream ancestor function is to explicitly express dependency relationships between input and output elements of a stream operation.

The contributions of this chapter are as follows:

1. A provenance data model that describe the key elements and the structure of information that form the representation of provenance for individual stream elements.

2. A set of primitive stream ancestor functions which can precisely express dependency relationships between input and output elements of stream operations.

This chapter is organized as follows. First, a fine-grained provenance model for stream processing systems is presented. Second, a novel stream provenance architecture is demonstrated. This is followed by the presentation of the programmatic specifications of primitive stream processing operations. After that, the specifications of stream ancestor functions are detailed. Finally, conclusions are drawn.

3.1 Fine-grained stream provenance model

3.1.1 Basic assumptions for fine-grained stream provenance model

To design a fine-grained provenance model, we make the following basic assumptions which describe the infrastructure of a stream processing system and a data stream model that our provenance model is intended to support.

- Streams of input events submitted to a stream processing system are assumed to come from a variety of data sources such as software programs that regularly generate data values or hardware devices (e.g. micro sensors) that submit their measurements in real-time.
- A stream processing system is represented as a set of interconnected nodes, with each node representing a stream operation or a stream processing unit (SPU). Input stream events flow through a directed graph of stream processing operations (stream processing flow) and finally, streams of output events are presented to applications that subscribe to receive results from the stream processing system.
- Each data stream in a stream processing system consists of a sequence of time ordered stream events and each individual stream event is composed of an event key - a unique reference of an individual stream event - and a content of stream event (data). The event key is added by the stream provenance system and can be assumed as standard.
- Streams are implicitly timestamped [119]. In this kind of stream timestamps, every stream event that first enters a stream processing system from a data source is timestamped based on the order that each event arrives. Timestamps are derived from a stream processing system time.
- Provenance recording is started from the beginning of a stream processing system's execution and all components of our stream provenance system are assumed to run on a single machine.

The stream processing infrastructure, which is based on our basic assumptions for fine-grained stream provenance model, is illustrated in Figure 3.1.

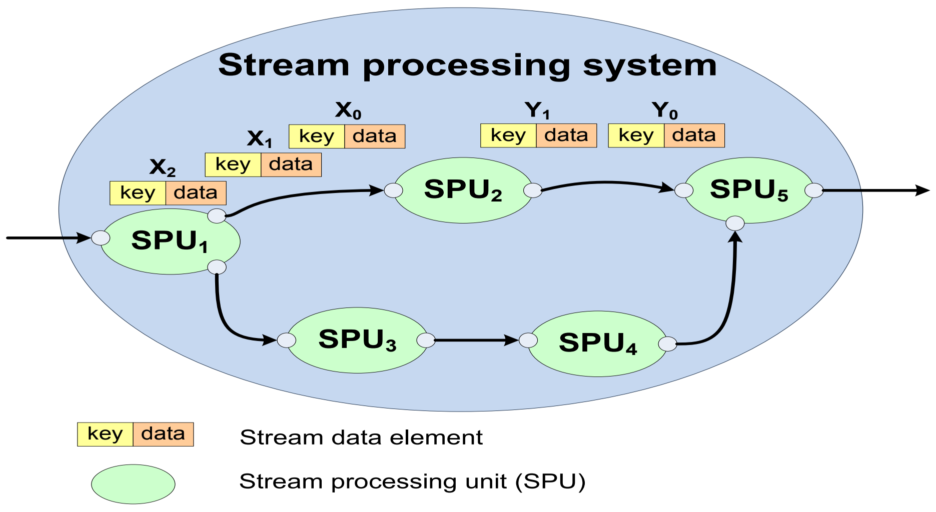


FIGURE 3.1: Infrastructure of a stream processing system

3.1.2 Fundamental concepts for fine-grained stream provenance tracking

Our fine-grained provenance tracking solution for streams can be divided into two main parts: a provenance data model for streams and stream ancestor functions. A provenance data model defines the structure and the key elements of provenance representation for streams provenance-related information to be stored in long-term data storage in order that the provenance of stream processing results can be retrieved. Another important part, a reverse mapping method (namely stream ancestor function) is utilized as a crucial mechanism for our provenance solution to express dependency relationships among individual stream elements. In this section, we discuss the basic concepts for fine-grained stream provenance tracking including the idea of stream ancestor functions and how the provenance of individual stream elements can be captured. After that, the generic provenance data model will be presented in detail in the next section.

Our fine-grained provenance tracking solution focuses on interactions between stream components (stream operations). In the context of stream processing systems, interactions consist of the stream events sent between stream operations. By capturing all the intermediate stream events that take place between stream operations involved in the processing of results, we can analyze or verify stream computation to ascertain the validity of the results. Based on the idea of capturing interactions between stream components, we distinguish between the whole of provenance related information collected by our stream provenance system and its individual parts. Interactions are represented by *provenance assertions* (or p-assertions) - assertions pertaining to provenance that are recorded by stream operations (assertors) during a stream processing system's execution. Each provenance assertion represents an individual stream event exchanged between two stream operations. The provenance related information collected by our stream provenance system consists of a set of provenance assertions.

We express dependencies between input and output events of stream operations by means of a *stream ancestor function* that is defined for each operation. The key idea of the stream ancestor function is that for a given reference to an output element, the stream ancestor function identifies the references of input events that are involved in the production of that output. The stream ancestor function does not work directly with individual stream elements, but instead it operates on a representation of each individual stream element - provenance assertion - that is recorded by each stream operation. We assume that every provenance assertion contains an event key, which consists of a timestamp, a sequence number, a stream identifier and a delay time. The event key plays an important role in the mapping process of the stream ancestor function. It serves as a unique reference for identifying each provenance assertion of an individual stream event in a stream. By composing stream ancestor functions for all stream operations in a stream system, all the elements of the intermediate data streams (represented by particular provenance assertions) involved in the processing of a particular output event, which we refer to as the complete provenance of a stream processing result, can be exactly identified. The concept of stream ancestor function is illustrated in Figure 3.2.

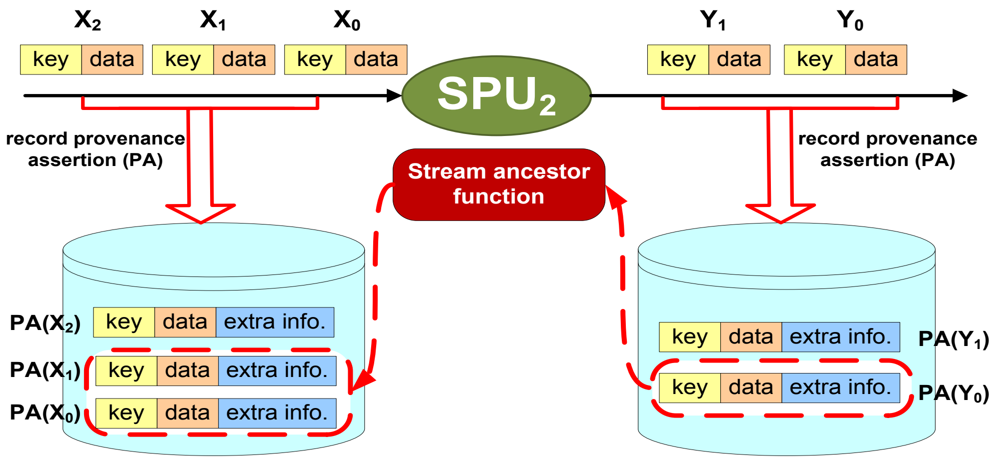


FIGURE 3.2: unoptimized stream ancestor function

Figure 3.2 shows how the stream ancestor function is used to express dependencies between input and output stream elements. In this example we focus on one particular stream processing unit (SPU_2). We can determine the input events involved in the processing of the output event Y_0 by passing the provenance assertion $PA(Y_0)$ - the representation of the stream event Y_0 - to the stream ancestor function defined explicitly for SPU_2. The stream ancestor function resolves input-output dependencies through the use of unique event keys and returns the provenance assertions $PA(X_0)$ and $PA(X_1)$ which represent the stream events X_0 and X_1 belonging to the input stream of SPU_2.

To extend the concept of stream ancestor functions, we introduce an enhanced solution called *optimized stream ancestor functions*. The aim of the optimized version of stream ancestor functions is to minimize storage consumption of the original function by recording only necessary information. The key idea of this solution is that only contents of stream events that act as the first input to a stream processing system are stored. For every intermediate stream event, we record only its key. Considering the fact that we use provenance assertions as representations of individual stream elements; therefore, the generation of the provenance assertions for every intermediate stream event does not include the contents of stream events, except for the first-input events - stream events that enter a stream processing system from data sources (e.g. sensors) and are first processed by stream operations. By applying this solution, we can rely on only event keys to identify the input events (representations of input events) that contribute to a particular output event without requiring any content. With the concept of optimized stream ancestor functions the amount of storage consumed for provenance collection is reduced and thus this can address the storage burden problem and also eliminate the requirement for storing every intermediate stream element. Figure 3.3 illustrates the concept of optimized stream ancestor functions.

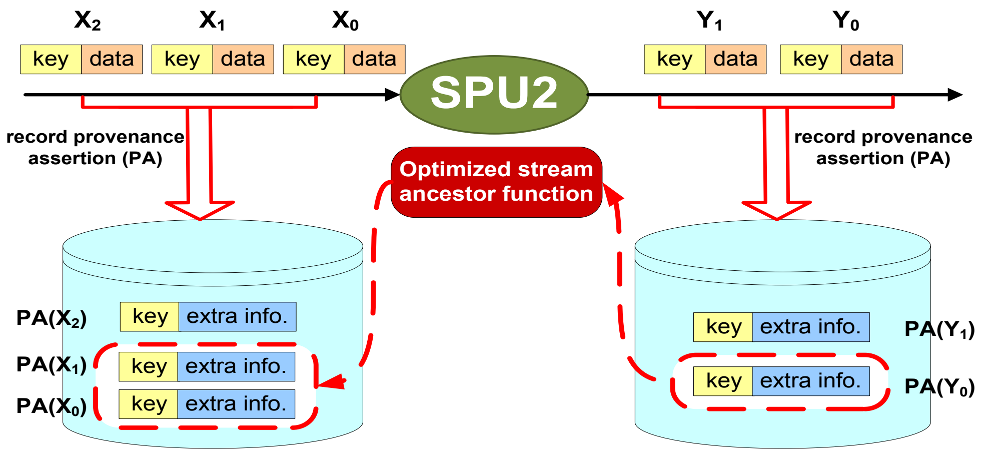

X_2
X_1
X_0
key data
SPU2
Y_1
Y_0
record provenance assertion (PA)
Optimized stream ancestor function
record provenance assertion (PA)
PA(X_2)
PA(X_1)
PA(X_0)
key extra info.
PA(Y_1)
PA(Y_0)

FIGURE 3.3: optimized stream ancestor function

As illustrated in Figure 3.3, each provenance assertion that represents individual intermediate stream event contains only an event key. The content of each intermediate event is discarded because it can be obtained later by replaying the execution of stream operations if required. In this example the optimized stream ancestor function takes the provenance assertion $PA(Y_0)$ representing the output event Y_0 as an input and returns the provenance assertions $PA(X_0)$ and $PA(X_1)$ which represent the input events X_0 and X_1 that are involved in the production of Y_0.

3.1.3 A provenance data model

We now present the provenance data model that underpins fine-grained provenance tracking in stream processing systems. The aim of this provenance data model is to describe the detail and the structure of information to be stored in a provenance store in order that provenance of the stream processing results can be retrieved. We identify the key elements that form the data model including entities representing units in a stream processing system and relationships between entities. The provenance data model for stream processing systems is illustrated in Figure 3.4.

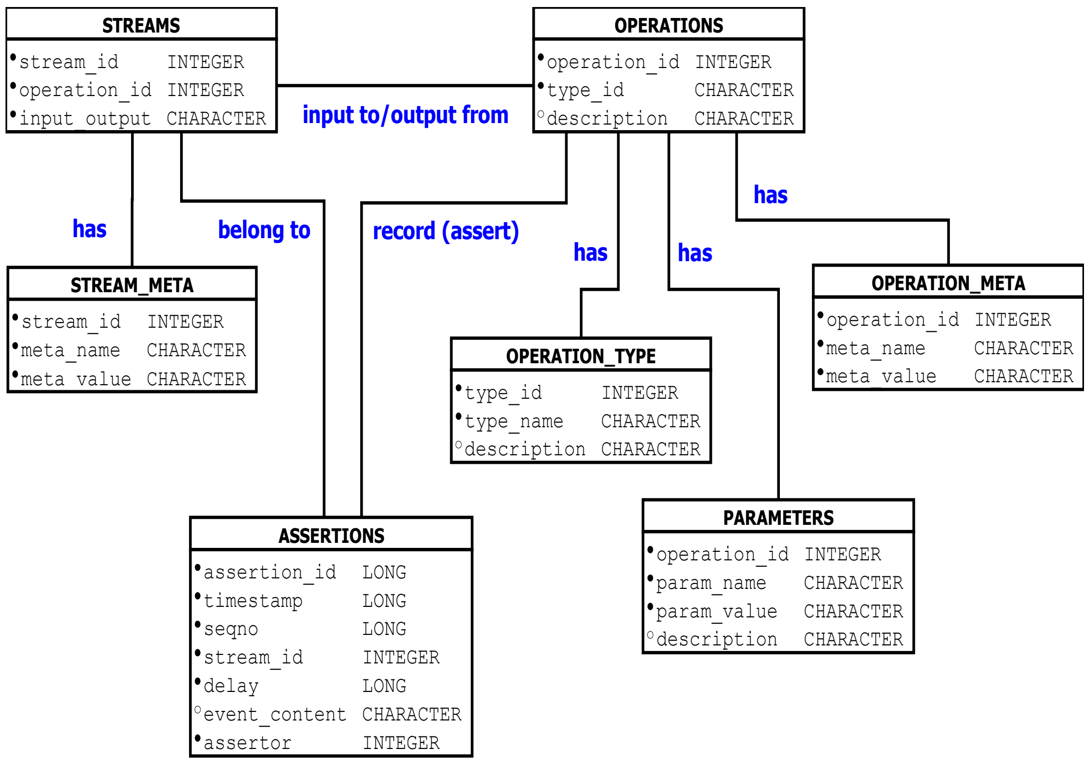


FIGURE 3.4: The provenance data model

As presented in Figure 3.4, the information contained in our provenance data model can be divided into two parts based on how the information is collected:

- *Static information* - the provenance related information gathered during initialization time or registration time of a stream processing system. Static information includes stream topology information, configuration parameters and metadata.
- *Dynamic information* - the provenance related information collected during execution time. Dynamic information includes a set of provenance assertions recorded by stream operations during execution time.

For static information we identify two main entities required for provenance collection in streams: *stream operations* and *input-output streams*. *Stream operations* are either continuous stream queries or application code that are predefined or registered during initialization time. They are executed continuously over streams of input elements until the end of the operation's life time. *Streams* in this context can be classified into two types: input streams and output streams. *Input streams* are streams that are consumed by stream operations in order to use them for stream processing. *Output streams* are streams that are outputted from or generated by executing stream operations. Output streams from one stream operation form input streams for other operations. The interconnection between stream operations associated with input-output streams forms a stream processing graph (or stream topology) that represents the internal data flow in a stream processing system. Note that, in the case that several versions of stream processing graphs are utilized for a stream application, metadata entities for both stream operations and streams can be used to specify a graph id for each processing graph.

Based on the idea of capturing all interactions (stream events) between stream operations during execution time, we identify provenance assertions as a key unit for dynamic provenance information. We assume that a data stream in a stream processing system consists of a sequence of time ordered stream events. Each individual stream event is composed of an event key and an event's content (data). Therefore, a provenance assertion which is a representation of an individual stream event exchanged between two stream operations is required to contain both important parts of a stream event (an event key and an event's content).

In our provenance data model, a provenance assertion is composed of four parts: an assertion identifier (*assertion_id*), an event key, an event's content (*event_content*) and an assertor identity (*assertor*). *An assertion identifier* serves as a primary key used for identifying each individual provenance assertion. *An event key*, which is utilized as a crucial element for supporting stream ancestor functions, consists of four composite parts including a timestamp, a sequence number (*seqno*), a stream identifier (*stream_id*) and a delay time (*delay*). A timestamp and a sequence number serve to define a temporal order and sequential order (position of an event) among stream events in a stream. They are assigned by a stream processing system when each stream event arrives at the system. A delay time for event processing is another important part of an event key. In our context, processing delay time is the amount of time spent for processing each individual stream element. Because processing delay time for each stream event is generally different, it is necessary to include the delay time in an event key. Furthermore, as we assume that each stream event in a stream is assigned a unique timestamp and dependencies between input and output events of stream operations can be represented as one-to-one or many-to-one relationships (input-output dependencies will be described in detail later in Section 3.3 - the specifications of primitive stream operations), therefore a delay time together with a timestamp for each stream event are adequately used by stream ancestor functions as variables for computing time dependencies between stream events. *An event's content* contained in a provenance assertion is an exact duplicate of the message or data contained in the stream event. Finally, *an assertor identity* is defined within provenance assertions in order to associate a particular provenance assertion with the stream operation that records it.

It is important to note that to support stream reproduction, the dynamic provenance information (provenance assertions) needs to be recorded by all stream operations in a stream processing system during execution time. The static provenance information (e.g. stream topology and stream operation parameters) is required to be specified at system initialization time as well for supporting the dynamic execution of our replay method. If one of them (static and dynamic provenance information) is not recorded correctly, stream reproduction cannot be properly performed (the detail of how stream reproduction or stream replay execution works will be discussed later in Section 4.2). In addition, it should also be noted that the extra info. contained in each provenance assertion (presented in Figure 3.2 and 3.3) is the combination of an assertion identifier and an assertor identity used for supporting provenance retrieval.

As shown in Figure 3.4, our provenance data model consists of seven entities which are related to each other. From the data model diagram, each stream operation has its own operation type (e.g. time-window and length-window operation), it is configured by some parameters and it has operation metadata. Each stream is either an input stream or an output stream of a stream operation and it has stream metadata as well. For the assertions entity (provenance assertions), each assertion is recorded or asserted by a stream operation and it belongs to a particular stream. By using the entities and the relationships between the entities defined in our provenance data model, we can construct a directed acyclic graph (DAG) which represents the provenance of individual stream elements in a stream processing system.

3.2 Provenance architecture for stream processing systems

To support the stream provenance model we have specified a provenance architecture for stream processing systems. In our context *a stream provenance system* is defined as a computer system that is responsible for dealing with the issues pertaining to the recording and querying of provenance information generated by a stream processing system during its execution time. Such a system is an implementation of a *provenance architecture for stream processing systems*. The provenance architecture allows us to describe the structure of the stream provenance system, system components and interactions between each components. Our provenance architecture for stream processing systems is shown in Figure 3.5.

Our provenance architecture for stream processing systems is adapted from the logical architecture for workflow provenance systems developed in the PASOA project [65]. Our architecture can be divided into two main parts: a persistent subsystem and a non-persistent subsystem. The provenance service plays a central role in the persistent subsystem. The provenance service is designed specifically to deal with provenance information that is stored persistently in a persistent data repository (provenance store). It is also designed to provide recording and querying functionalities which support the different phases of the provenance lifecycle in computer systems. To encapsulate the detail of such functionalities, the service provides two interfaces - a recording interface

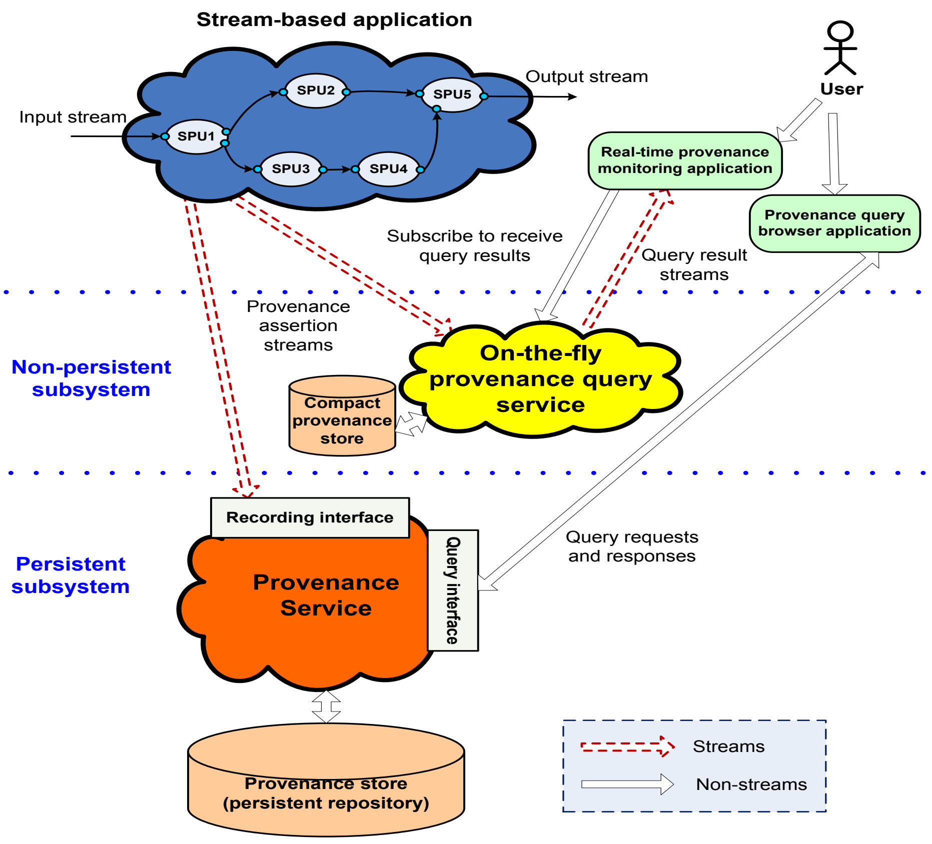

Stream-based application
Input stream
SPU1
SPU2
SPU3
SPU4
SPU5
Output stream
User
Real-time provenance monitoring application
Provenance query browser application
Subscribe to receive query results
Query result streams
Provenance assertion streams
Non-persistent subsystem
On-the-fly provenance query service
Compact provenance store
Recording interface
Query interface
Persistent subsystem
Provenance Service
Query requests and responses
Provenance store (persistent repository)
Streams
Non-streams

FIGURE 3.5: The provenance architecture for stream processing systems

and a query interface. These interfaces specify the messages accepted and returned by a provenance service and they are the entry point to the provenance service that allows applications to inter-operate with different implementations of provenance services.

To capture the provenance of stream processing results, provenance assertions are recorded by each stream operation for each individual stream element in a stream processing system. The provenance assertions are not generated and recorded at a single time, but instead their generation is interleaved continuously with execution. After each provenance assertion is received by the provenance service via the recording interface it is recorded into a provenance store - a central storage component that offers a long-term persistent storage of provenance assertions. The provenance service does not just provide provenance recording functionality, but instead it supports provenance query functionality as well. Once provenance assertions have been recorded in the provenance store, users who operate a client application (e.g. a provenance query browser application) can send a variety of provenance related queries to the provenance service in order to collect precise and accurate provenance of stream processing results.

Another important component is an on-the-fly provenance query service that is a central component of the non-persistent subsystem part. In this context, the on-the-fly provenance query service is designed to be utilized as a stream component. During an initialization period, users who manage a client application (e.g. a real time provenance monitoring application) need to subscribe to receive query results from the on-the-fly provenance query service. As provenance assertions are being recorded and received by the on-the-fly provenance query service, provenance queries are performed automatically and continuously over the streams of provenance assertions. It is important to note that the generation of on-the-fly provenance queries is based on configuration parameters and stream topology information specified during system registration time. Such information is stored in a compact provenance store - a compact version of the provenance store used only for storing static provenance information. Finally, provenance query results are generated as streams and they are streamed back to the subscriber application. On-the-fly provenance queries are discussed in more detail in Chapter 5.

The remaining sections will present the primitive stream operations. These operations are recognized as common stream operations developed in several stream processing system projects [60, 28, 8, 1]. The purpose of the presentation of these primitive operations is to demonstrate how the output element for each stream operation is produced in terms of its input elements. Based on these operations we then present a list of specifications of stream ancestor functions. After presenting each stream ancestor function, we present a small example illustrating how each primitive stream operation and its stream ancestor function work in practice. It should be noted that our goal is not trying to explain all of existing stream operations but we aim to demonstrate that the idea and the method used in the specifications of our primitive stream operations and their stream ancestor functions can be transfered to other new stream operations if required. Furthermore, because our primitive stream operations (and also stream ancestor functions) are designed to be composed together in order to build more complicated stream operations (we will discuss in detail later in Section 4.1.1), this ability can be usefully exploited to deal with custom or user-defined stream operations.

In the following sections, all stream operations and stream ancestor functions are defined by using Standard ML (SML [56]) - a general-purpose functional programming language. By using Standard ML, we can illustrate how continuous queries are formulated in stream operations and how each stream ancestor function works in order to express dependencies between input and output stream elements. In addition, the reasons that Standard ML is selected over other functional programming languages (such as Haskell) is that Standard ML is a strict (eager evaluation - subexpressions are alway evaluated completely before the function is applied) and strong type checking language. Although Haskell provides the ability to represent infinite streams in a finite manner, such ability relies on delayed evaluations, which are forced when stream results are requested. Forcing delayed evaluations may result in the forcing of other delayed evaluations as well. Since the purpose of our study is to explicitly represent computation that would be performed in stream components - potentially in hardware, for our study, we instead prefer a call-by-value language such as Standard ML, which also provides a typed discipline, but with explicit control of evaluation.

3.3 Primitive stream processing operations

3.3.1 The basic notation of a data stream

A data stream is an ordered sequence of tuples (stream events) which consist of an event key and event data. An event key, which is used as a unique identifier for stream events, contains a timestamp (TIME), a sequence number (LargeInt), a stream identifier (STREAMID) and a delay time for event processing (TIME). In the context of Standard ML, we represent a data stream as a list of events (*'a EVENT list*) where a stream event can contain a varying number of elements and it can be any type of content.

The following are data types for representing a data stream.

$datatype\ STREAMID\ StreamID\ of\ int;$
$datatype\ KEY\ =\ Key\ of\ TIME\ *\ LargeInt.int\ *\ STREAMID\ *\ TIME;$
$datatype\ 'a\ EVENT\ =\ Event\ of\ KEY\ *\ 'a;$

The time data type and time comparison operations are represented as follows:

$datatype\ TIME\ =\ Time\ of\ LargeInt.int;$

$(*\ fn\ :\ TIME\ *\ TIME -> TIME\ *)$
$fun\ ++\ (Time(x), Time(y))\ =\ Time(x+y);$

$(*\ fn\ :\ TIME\ *\ TIME -> TIME\ *)$
$fun\ --\ (Time(x), Time(y))\ =\ Time(x-y);$

$(*\ fn\ :\ TIME\ *\ TIME -> bool\ *)$
$fun\ GT\ (Time(x), Time(y))\ =\ (x > y);$

$(*\ fn\ :\ TIME\ *\ TIME -> bool\ *)$
$fun\ GTE\ (Time(x), Time(y))\ =\ (x >= y);$

$(*\ fn\ :\ TIME\ *\ TIME -> bool\ *)$
$fun\ LT\ (Time(x), Time(y))\ =\ (x < y);$

$(*\ fn\ :\ TIME\ *\ TIME -> bool\ *)$
$fun\ LTE\ (Time(x), Time(y))\ =\ (x <= y);$

$(*\ fn\ :\ TIME\ *\ TIME -> bool\ *)$
$fun\ EQ(Time(x), Time(y))\ =\ if\ x\ =\ y\ then\ true\ else\ false;$

The following is an example of a data stream represented using a list of events.

$[Event\ (Key\ (Time\ 22, 1, StreamID\ 1, Time\ 1), 55),$
$Event\ (Key\ (Time\ 32, 2, StreamID\ 1, Time\ 2), 60)$
$Event\ (Key\ (Time\ 42, 3, StreamID\ 1, Time\ 1), 65)] :\ int\ EVENT\ list$

3.3.2 Shared functions

Shared functions are defined in order to separate some common routines used in several operations from the core function of stream operations. The benefits of defining shared functions is that only a core function for each stream operation is presented, the definitions of stream operations are simplified, and they can be more easily understood. These shared functions include *TScope*, *LScope*, *getDataList*, *tStampGT*, *tStampEQ* and *withTime*.

TScope and *LScope* are functions used to filter elements of a list (B) that are outside the scope of a data window (the scope ranges from lower bound (lb) to upper bound (ub)). The difference between these two functions is that elements of a list are filtered by a timestamp (t) for the former and by size of data window (l) for the latter. The *getDataList* function is used to get the event content (event data) of stream events in a list (B). To determine the temporal order between two stream events, *tStampGT* and *tStampEQ* functions are introduced. Each of these functions compares event timestamps and return a boolean value indicating the result of the comparison. Finally, *withTime* function is used to calculate a delay time for processing for each stream event. It takes an internal function of a stream operation as an input and returns a delay time together with the output of that internal function. The definitions of these shared functions are presented below.

```
(* fn : TIME * TIME * 'a EVENT list → 'a EVENT list *)
fun TScope(ub, lb, [ ]) = [ ]
| TScope(ub, lb, ((evt as Event(Key(t, _, _, _), _)) :: B)) =
    if (ub GTE t) andalso (t GTE lb) then
        evt :: TScope(ub, lb, B)
    else TScope(ub, lb, B)
end

(* fn : int * 'a EVENT list → 'a EVENT list *)
fun LScope(l, [ ]) = [ ]
| LScope(l, (evt :: B)) =
    if (l > 0) then
        evt :: LScope((l − 1), B)
    else [ ];
```

```
$(* \; fn \; : \; 'a \; EVENT \; list \; \rightarrow \; 'a \; list \; *)$
$fun \; getDataList \; ([\,]) \; = \; [\,]$
$| \; getDataList((Event(\_, e)) :: B) \; = \; e :: getDataList(B);$

$(* \; fn \; : \; 'a \; EVENT - > \; 'b \; EVENT - > \; bool \; *)$
$fun \; tStampGT \; (Event(Key(t_1, \_, \_, \_), \_)) \; (Event(Key(t_2, \_, \_, \_), \_)) \; = \; (t_1 \; GT \; t_2);$

$(* \; fn \; : \; 'a \; EVENT - > \; 'b \; EVENT - > \; bool \; *)$
$fun \; tStampEQ \; (Event(Key(t_1, \_, \_, \_), \_)) \; (Event(Key(t_2, \_, \_, \_), \_)) \; = \; (t_1 \; EQ \; t_2);$

$(* \; fn \; : \; (unit - > \; 'a) - > \; 'a \; * \; TIME \; *)$
$fun \; withTime \; \; func \; =$
$let$
    $val \; timer \; = \; startRealTimer();$
$in$
    $let$
        $val \; e' \; = \; func();$
    $in$
        $let$
            $val \; d_n \; = \; Time(Time.toMilliseconds(checkRealTimer(timer)));$
        $in$
            $(e', d_n)$
        $end$
    $end$
$end;$
```

3.3.3 Map

One of the most common stream operations that operate on a single stream element at a time is the map operation. The map operation applies an input function to the content of every stream element in a stream. The functioning of the map operation can be seen in Figure 3.6.

As shown in Figure 3.6, an average function (AVG) is used as an input function that applies to every input stream event. The input is the output stream of the time window that will be presented later. Note that we use a simple form of stream event which consists of a timestamp, a sequence number and event data representing a stream event in the diagram. In addition d specified in the diagram is a delay time for the processing of the map operation. The definition of the map operation is presented in Figure 3.7.

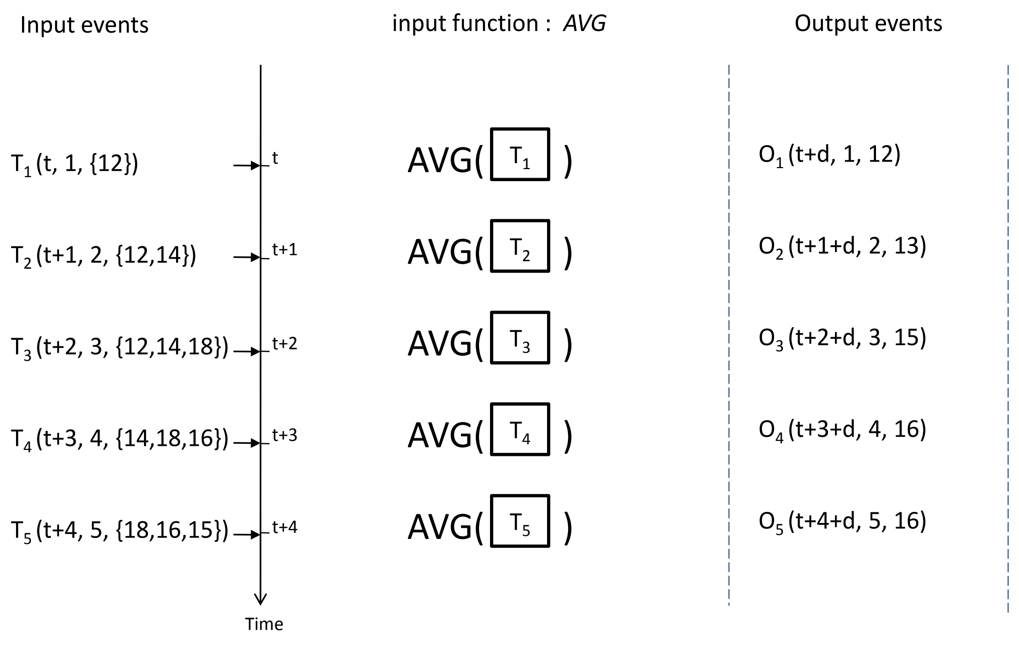


FIGURE 3.6: Output example for the map operation

Stream operation: Map - $Map(F, sid)$
Let S be a stream of type $'a\ EVENT\ list$
Let F be a function of type $('a \rightarrow 'b)$
Let sid be a stream identifier of an output stream
$Map(F, sid)\ S$ is defined as:

```
$(* \ fn \ : \ ('a -> 'b) \ * \ STREAMID -> 'a \ EVENT \ list -> 'b \ EVENT \ list \ *)$

$fun\ Map(F, sid)\ [\,] \ = \ [\,]$
$|\ Map(F, sid) \ \ ((Event(Key(t, n, \_, \_), e)) :: S) \ =$
$let$
    $val\ (e', d_n) \ = \ withTime \ \ (fn\ () \ => \ F \ \ e)$
$in$
    $(Event(Key(t++d_n, n, sid, d_n), e')) :: Map(F, sid) \ \ S$
$end$
```

FIGURE 3.7: The definition of the map operation

As illustrated in Figure 3.7, the map operation takes a stream of type $'a\ EVENT\ list$ as an input, applies a function F to every stream event in the input stream and finally produces an output stream of type $'b\ EVENT\ list$. This operation keeps sequence numbers of stream events the same and applies a function F over the content of stream events. Examples of map operations include several kinds of aggregated operation: SUM, AVG, MIN and MAX. In addition, it is important to note that for all primitive stream operations, we utilize *withTime* function to obtain the amount of time spent (delay) for stream processing. The timer of this function is started when each stream event is being received and it is stopped after each event has been processed.

3.3.4 Filter

Another stream operation that operates on a single stream element at a time is the filter operation. The filter operation screens events in a stream for those that satisfy an input predicate (filter condition). Figure 3.8 illustrates the functioning of the filter operation.

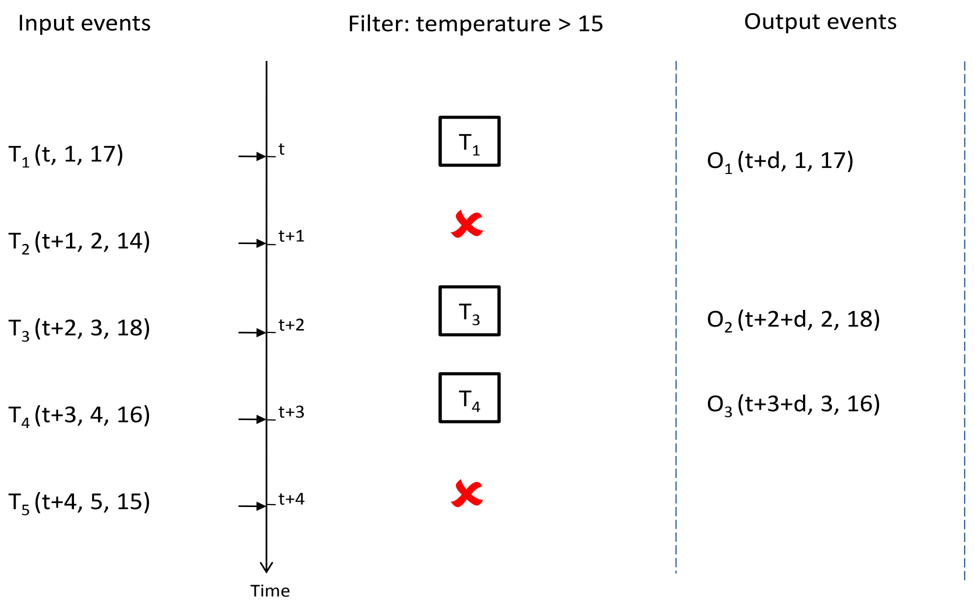


FIGURE 3.8: Output example for the filter operation

The definition of the filter operation is given in Figure 3.9:

Stream operation: Filter - $Filter(P, sn, sid)$
Let S be a stream of type $'a\ EVENT\ list$
Let P be a predicate that evaluates over stream events
Let sn be a sequence number generated according to a number of output events
Let sid be a stream identifier of an output stream
$Filter(P, sn, sid)\ S$ is defined as:

```
$(* fn : ('a -> bool) * int * STREAMID -> 'a\ EVENT\ list$
$ -> 'a\ EVENT\ list *)$

$fun\ Filter(P, sn, sid)\ [\,] = [\,]$
$|\ Filter(P, sn, sid)\ ((Event(Key(t, \_, \_, \_), e)) :: S) =$
$let$
    $val\ (Pred, d_n) = withTime\ (fn\ ()\ => P\ e)$
$in$
    $if\ Pred\ then$
        $(Event(Key(t + +d_n, sn, sid, d_n), e)) :: Filter(P, sn + 1, sid)\ S$
    $else$
        $Filter(P, sn, sid)\ S$
$end$
```

FIGURE 3.9: The definition of the filter operation

As presented in Figure 3.9, the filter operation applies a predicate P (filter condition) over the content of an input stream. In the case that an input event satisfies the predicate, an output event with a new sequence number sn is produced. A new sequence number for each output event is generated according to the number of output events. If the input event fails to satisfy the predicate, the filter operation produces nothing.

3.3.5 Windowed operations

In this section, windowed operations - operations that operate on sets of consecutive events from a stream at a time [28] - are presented. Two common types of windowed operations are described: a time window and a length window. We will detail the abstract functions first and then the definitions of the windowed operations are presented.

3.3.5.1 Abstract functions for the windowed operations

We define the abstract functions for the windowed operations to separate common routines contained in both time window and length window operations. These abstract functions include $Window$ and $Window_t$.

The $Window$ function is the abstract function defined for input-driven window operations - the windowed operations that each output is produced in terms of input events. Another function, $Window_t$, is the abstract function defined for clock-driven window operations - the windowed operations that each output is produced based on the presence of elements in a tick stream (a stream used as a trigger for execution of windowed operations). Both functions takes a window scope function (either $TWindowScope$ or $LWindowScope$) as an input parameter. The window scope functions are used to determine which stream elements of an input stream are in the window extent. The definitions of the abstract functions for windowed operations are presented as follows.

```
$(*\ fn\ :\ 'a\ *\ 'b\ EVENT\ list\ *\ STREAMID\ ->\ (TIME\ ->\ 'a\ ->$
$'b\ EVENT\ list\ ->\ 'b\ EVENT\ list)\ ->\ 'b\ EVENT\ list\ ->\ 'b\ list\ EVENT\ list\ *)$
$fun\ Window(w, B, sid)\ \ winScope\ \ [\,]\ =\ [\,]$
$|\ Window(w, B, sid)\ \ winScope\ \ ((evt\ as\ Event(Key(t, n, \_, \_), e)) :: S)\ \ =$
$let\ \ \ \ val\ ((scope, eList), d_n)\ \ =\ \ withTime\ (fn\ ()\ \ =>$
                                    $let$
                                      $val\ scope\ \ =\ \ winScope\ \ t\ \ w\ \ ([evt]\ @\ B)$
                                    $in$
                                      $(scope, getDataList(scope))$
                                    $end)$
$in$
    $(Event(Key(t ++ d_n, n, sid, d_n), eList)) :: Window(w, scope, sid)\ \ winScope\ \ S$
$end$
```

```
(* fn : 'a * 'b EVENT list * STREAMID -> (TIME -> 'a -> 'b EVENT list
 -> 'b EVENT list) -> 'c EVENT list -> 'b EVENT list -> 'b list EVENT list *)
fun Window_t(w, B, sid) winScope [ ] _ = [ ]
| Window_t(w, B, sid) winScope ((evt_t as Event(Key(t, n, _, _), e)) :: T) [ ] =
let val ((scope, eList), d_n) = withTime(fn () =>
                  let
                    val scope = winScope t w B
                  in
                    (scope, getDataList(scope))
                  end)
in
  (Event(Key(t ++d_n, n, sid, d_n), eList)) :: Window_t(w, scope, sid) winScope T [ ]
end
| Window_t(w, B, sid) winScope ((evt_t as Event(Key(t, n, _, _), e)) :: T) (evt_i :: S) =
let val ((B', scope, scope', eList, eList'), d_n) = withTime(fn () =>
                  let
                    val scope = winScope t w B
                    val B' = [evt_i] @ B
                    val scope' = winScope t w B'
                  in
                    (B', scope, scope', getDataList(scope), getDataList(scope'))
                  end)
in
 if tStampGT evt_t evt_i then Window_t(w, B', sid) winScope (evt_t :: T) S
 else
  if tStampEQ evt_t evt_i then
   (Event(Key(t ++d_n, n, sid, d_n), eList')) :: Window_t(w, scope', sid) winScope T S
  else
   (Event(Key(t ++d_n, n, sid, d_n), eList)) :: Window_t(w, scope, sid) winScope T (evt_i :: S)
end

(* fn : TIME -> TIME -> 'a EVENT list -> 'a EVENT list *)
fun TWindowScope t w B = TScope(t, t --w, B)

(* fn : 'a -> int -> 'b list -> 'b list *)
fun LWindowScope t l B = LScope(l, B)
```

3.3.5.2 Sliding time windows

A sliding time window is a windowed operation (a data window) where the extent of the window is defined in terms of a time interval. At any point in time, the time window generates an output event from the most recent input events over a given time period.

Figure 3.10 and 3.11 show the time window operation in action.

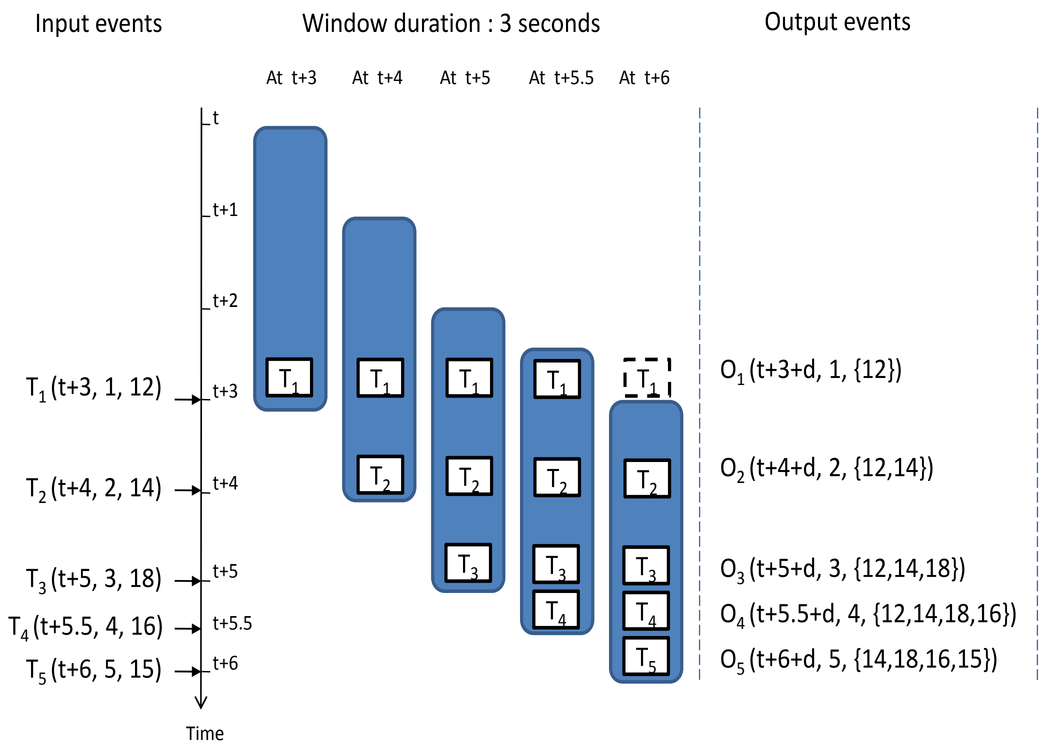


FIGURE 3.10: Output example for the input-driven time window

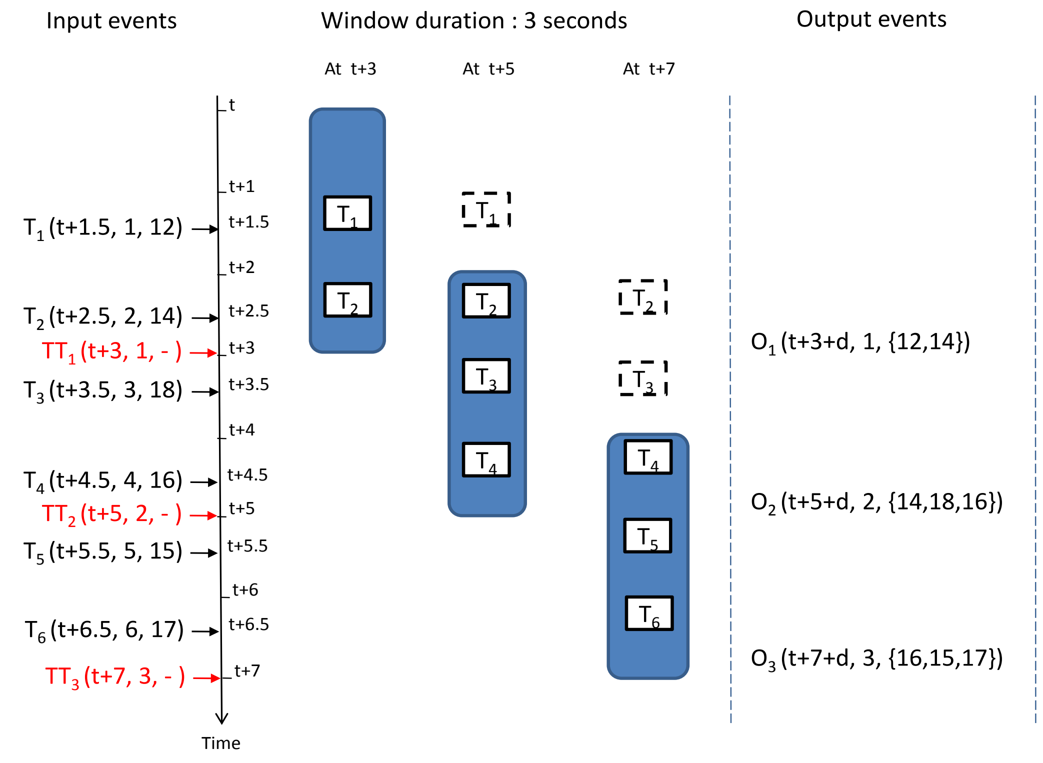


FIGURE 3.11: Output example for the clock-driven time window

The diagram in Figure 3.10 illustrates how the sliding time window (input-driven time window) contents change as input events arrive and shows the output events produced in terms of input events. The diagram starts at a given time t. Input events arrive in the time window at time t + 3 and t + 4 seconds and so on. As shown in the diagram, the extent and the execution of the time window are determined based on the arrival of input stream events. For example, at time t+4 seconds, based on the arrival of the event T_2, the events T_1 and T_2 are identified as input events in the extent of the time window and finally the time window produces the O_2 event as an output. After each output event is generated, the time window changes progressively (move forward in time) in order to capture a new set of stream events. The diagram in Figure 3.11 shows the functioning of another type of sliding time window operation - the clock-driven time window - in which each output event is produced based on the presence of elements in a tick stream (a stream that is used as a trigger for execution of stream operations). The diagram starts at a given time t and input events (T events) arrive in the time window at time t + 1.5 and t + 2.5 seconds and so on. In addition, tick stream events (TT events) arrive the time window operation at time t + 3 and t + 5 and t + 7 seconds. The time window operations (both the input-driven time window and the clock-driven time window) are defined as follows.

Stream operation: Input-driven time window - $TW(w, B, sid)$
Let S be a stream of type $'a\ EVENT\ list$
Let w be a duration of the time window
Let B be a data buffer used to temporarily store events
Let sid be a stream identifier of an output stream
$TW(w, B, sid)\ S$ is defined as:

$(*\ fn\ :\ TIME\ *\ 'a\ EVENT\ list\ *\ STREAMID\ ->\ 'a\ EVENT\ list$
$->\ 'a\ list\ EVENT\ list\ *)$

$fun\ TW(w, B, sid)\ \ S\ =\ Window(w, B, sid)\ \ TWindowScope\ \ S;$

FIGURE 3.12: The definition of the input-driven time window

Stream operation: Clock-driven time window - $TW_t(w, B, sid)$
Let S be a stream of type $'a\ EVENT\ list$
Let T be a stream of type $'a\ EVENT\ list$ used as clock ticks of the time window
Let w be a duration of the time window
Let B be a data buffer used to temporarily store events
Let sid be a stream identifier of an output stream
$TW_t(w, B, sid)\ T\ S$ is defined as:

$(*\ fn\ :\ TIME\ *\ 'a\ EVENT\ list\ *\ STREAMID\ ->\ 'b\ EVENT\ list$
$->\ 'a\ EVENT\ list\ ->\ 'a\ list\ EVENT\ list\ *)$

$fun\ TW_t(w, B, sid)\ \ T\ \ S\ =\ Window_t(w, B, sid)\ \ TWindowScope\ \ T\ \ S;$

FIGURE 3.13: The definition of the clock-driven time window

As shown in Figure 3.12 and 3.13, the two time window operations utilize internal values of a stream component - a duration of the time window w, a data buffer B and a stream ID sid - as input parameters at registration time. When each individual input event flows into each operation it produces an output event, which combines the most recent events of an input stream during a particular time period w. The difference between these two operations is that outputs are input driven for one and clock driven for the other. For the clock-driven time window, the tick stream T is utilized by the operation to determine when each output event has to be produced. In addition, as shown in the definitions, the input-driven time window utilizes the $Window$ function (abstract function) for its core business logic and the $Window_t$ function is used as the base function of the clock-driven time window.

3.3.5.3 Length windows

Another type of data window is the length window (or count-based window). A length window is a data window where the extent of the window is defined in terms of the number of events. At any point in time a length window covers the most recent N events (size of window) of a stream. A length window of size 3 events, for example, keeps the last 3 events for a stream. Figure 3.14 and 3.15 show the length window operation in action.

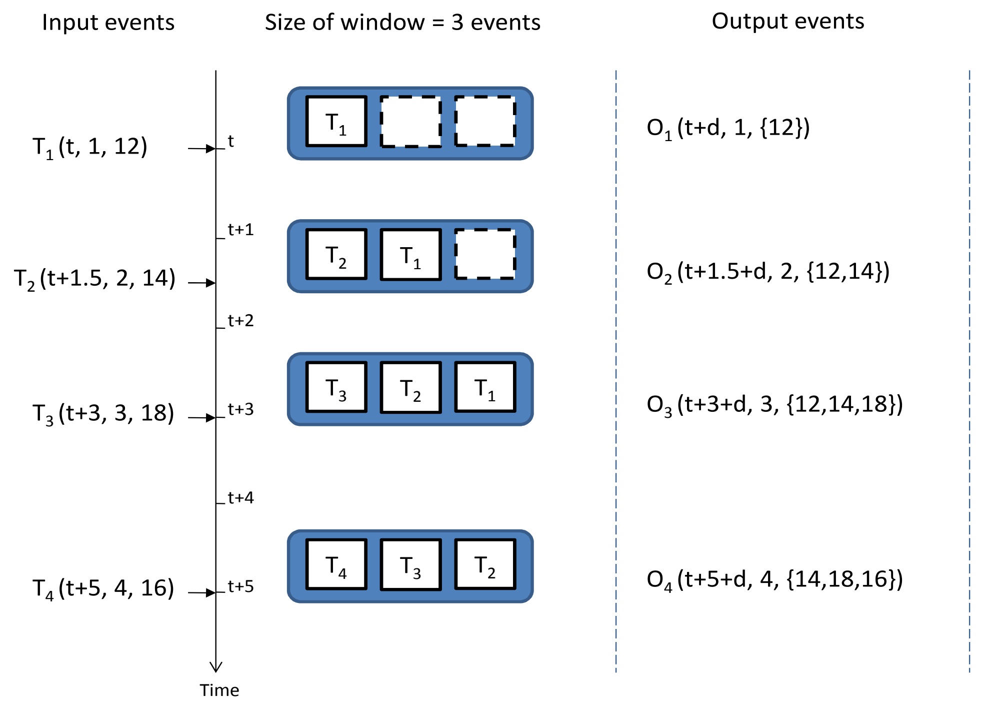


FIGURE 3.14: Output example for the input-driven length window

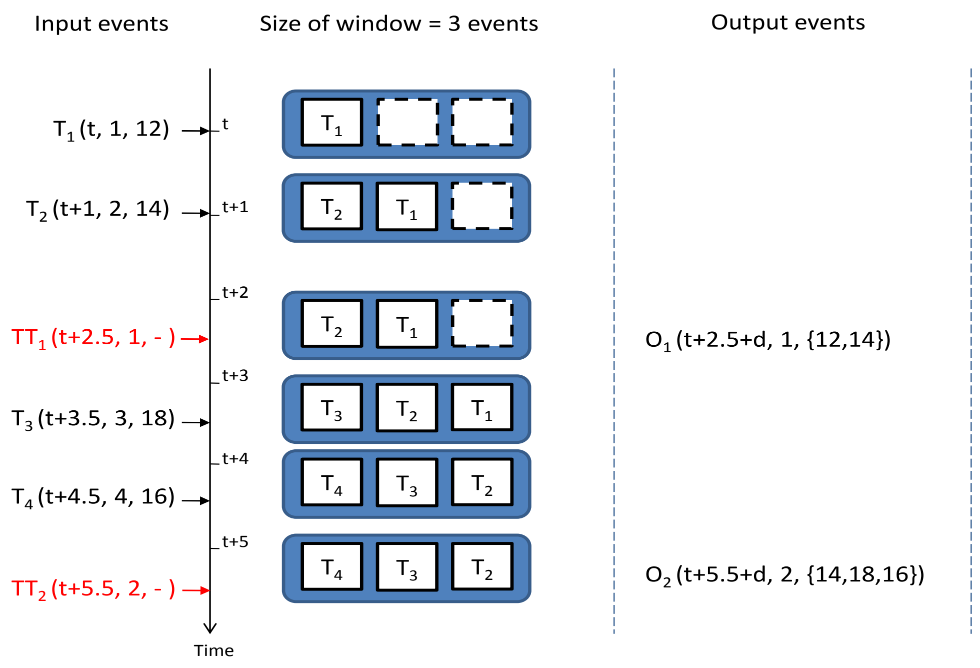


FIGURE 3.15: Output example for the clock-driven length window

The diagram in Figure 3.14 illustrates how the input-driven length window contents change as events arrive and shows the output events produced in term of the number of input events. The diagram starts at a given time t and input events arrive in the length window operation at time t and t + 1.5 seconds and so on. The diagram in Figure 3.15 illustrates the functioning of another type of length window operation - the clock-driven length window - in which each output event is produced based on the presence of elements in a tick stream. The clock-driven length window is used as an internal function of the length-window join operation that will be presented later in this chapter. The diagram starts at a given time t and input events (T events) arrive in the length window at time t and t + 1 seconds and so on. Furthermore, tick stream events (TT events) arrive the length window at time t + 2.5 and t + 5.5 seconds. The definitions of both length window operations are defined in Figure 3.16 and 3.17.

Stream operation: Input-driven length window - $LW(l, B, sid)$
Let S be a stream of type $'a\ EVENT\ list$
Let l be size of the length window
Let B be a data buffer used to temporarily store events
Let sid be a stream identifier of an output stream
$LW(l, B, sid)$ S is defined as:

$(*\ fn\ :\ int\ *\ 'a\ EVENT\ list\ *\ STREAMID -> 'a\ EVENT\ list$
$-> 'a\ list\ EVENT\ list\ *)$

$fun\ LW(l, B, sid)\ \ S\ \ =\ \ Window(l, B, sid)\ \ LWindowScope\ \ S;$

FIGURE 3.16: The definition of the input-driven length window

Stream operation: Clock-driven length window - $LW_t(l, B, sid)$
Let S be a stream of type $'a\ EVENT\ list$
Let T be a stream of type $'a\ EVENT\ list$ used as clock ticks of the length window
Let l be size of the length window
Let B be a data buffer used to temporarily store events
Let sid be a stream identifier of an output stream
$LW_t(l, B, sid)\ T\ S$ is defined as:

$(*\ fn\ :\ int\ *\ 'a\ EVENT\ list\ *\ STREAMID\ ->\ 'b\ EVENT\ list\ ->$
$'a\ EVENT\ list\ ->\ 'a\ list\ EVENT\ list\ *)$

$fun\ LW_t(l, B, sid)\ \ T\ \ S\ \ =\ \ Window_t(l, B, sid)\ \ LWindowScope\ \ T\ \ S;$

FIGURE 3.17: The definition of the clock-driven length window

As shown in Figure 3.16 and 3.17, the two length window operations utilize internal values of a stream component - size of length window l, a data buffer B and a stream ID sid - as input parameters at registration time. When each individual input event enters the operation it produces an output event that combines the most recent l events of an input stream together. Instead of using timestamps, the size of the data window l is used together with the scope function $LWindowScope$ for checking which events of an input stream are in the window extent. The difference between these two operations is that outputs are input driven for one and clock driven for the other. For the clock-driven length window the tick stream T is used by the operation to determine when each output event has to be generated.

3.3.6 Join operations

As in a traditional relational database system, a join operation is an important operation in stream processing systems. In this section, two common window-based join operations are described: the time-window join and the length-window join. We will describe the abstract functions for join operations first and then the definitions of two window-based join operations will be presented.

3.3.6.1 Abstract functions for the window-based join operations

A window-based join is a binary operation that pairs stream events from two input streams. Stream events from two time-based windows or count-based windows (length windows) are combined and each output event is produced according to a join condition. In order to clearly describe how the window-based join operation works, an output example for the join operation (sliding time-window join) is presented in Figure 3.18.

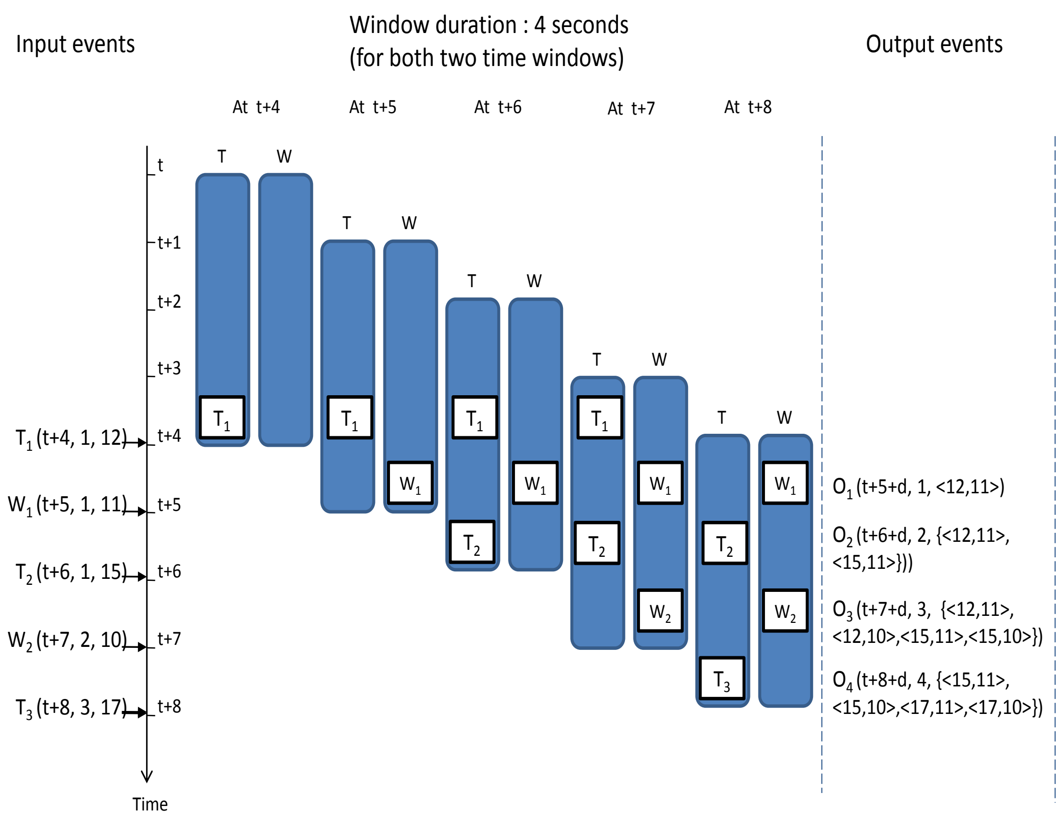

Input events
Window duration : 4 seconds
(for both two time windows)
Output events
At t+4
At t+5
At t+6
At t+7
At t+8
T
W
t
t+1
t+2
t+3
t+4
t+5
t+6
t+7
t+8
Time
T1 (t+4, 1, 12)
W1 (t+5, 1, 11)
T2 (t+6, 1, 15)
W2 (t+7, 2, 10)
T3 (t+8, 3, 17)
O1 (t+5+d, 1, <12,11>)
O2 (t+6+d, 2, {<12,11>, <15,11>}))
O3 (t+7+d, 3, {<12,11>, <12,10>,<15,11>,<15,10>})
O4 (t+8+d, 4, {<15,11>, <15,10>,<17,11>,<17,10>})

Figure 3.18: Output example for the time-window join operation

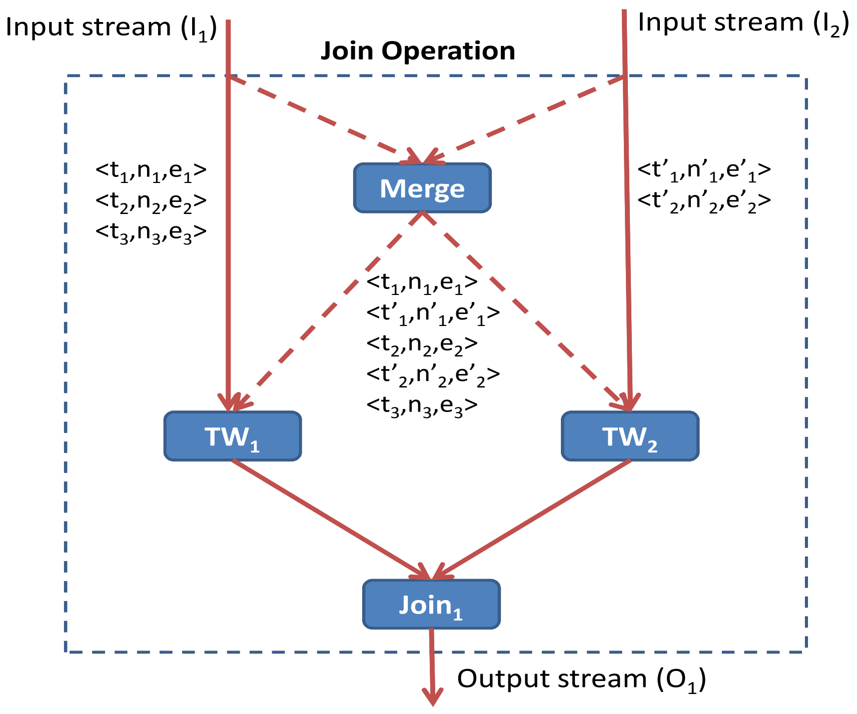


Figure 3.19: The internal functions of the time-window join operation

As shown in Figure 3.18, the diagram starts at a given time t and the input events from two streams arrive at time t+4 and t+5 seconds and so on. In this example, both streams have the same window duration (4 seconds). Furthermore, in Figure 3.19, the detail of the internal functions inside the join operation is presented. This can be used to explain how the operation works.

We now present the abstract function for the window-based join operations. This abstract function is defined to separate common routines contained in both time-window join and length-window join operations. The definition of the abstract function is presented as follows.

```
$(* fn\ :\ 'a\ *\ 'a\ *\ ('b\ *\ 'b\ *\ 'c\ list\ ->\ ''d\ list)$
$*\ int\ *\ int\ *\ STREAMID\ ->\ ('a\ *\ 'e\ list\ *\ STREAMID$
$->\ 'f\ EVENT\ list\ ->\ 'f\ EVENT\ list\ ->\ 'b\ EVENT\ list)$
$->\ 'f\ EVENT\ list\ ->\ 'f\ EVENT\ list\ ->\ ''d\ list\ EVENT\ list\ *)$
$fun\ Join(w1, w2, J, nj, sn, sid)\ \ ClockDrivenWin\ \ [\,]\ \ \_\ \ =\ [\,]$
$|\ Join(w1, w2, J, nj, sn, sid)\ \ ClockDrivenWin\ \ \_\ \ [\,]\ =\ [\,]$
$|\ Join(w1, w2, J, nj, sn, sid)\ \ ClockDrivenWin\ \ (evt1 :: S1)\ \ (evt2 :: S2)\ =$
$let$
    $val\ Tick\ =\ Merge(sn, sid)\ \ (evt1 :: S1)\ \ (evt2 :: S2)$
    $val\ win1\ =\ ClockDrivenWin(w1, [\,], sid)\ \ Tick\ \ (evt1 :: S1)$
    $val\ win2\ =\ ClockDrivenWin(w2, [\,], sid)\ \ Tick\ \ (evt2 :: S2)$
$in$
    $join1(J, nj, sid)\ \ win1\ \ win2$
$end$

$fun\ join1(J, nj, sid)\ \ \_\ \ [\,]\ =\ [\,]$
$|\ join1(J, nj, sid)\ \ [\,]\ \ \_\ =\ [\,]$
$|\ join1(J, nj, sid)\ \ (Event(Key(t, \_, \_, \_), e1) :: win1)\ \ (Event(\_, e2) :: win2)\ =$
$let$
    $val\ (jOutput, d_n)\ =\ withTime\ (fn\ ()\ =>\ J(e1, e2, [\,]));$
$in$
    $if(jOutput\ =\ [\,])\ then$
        $join1(J, nj, sid)\ \ win1\ \ win2$
    $else$
        $(Event(Key(t + +d_n, nj, sid, d_n), jOutput)) :: join1(J, nj + 1, sid)\ \ win1\ \ win2$
$end$
```

As presented in the definition of the abstract function, the window-based join consists of four internal functions: $Merge$, clock-driven windows ($win1$ and $win2$) and $join1$. The operation takes events from two streams ($S1$ and $S2$) as input. Each time that a new event from either streams arrives the operation returns a set of output events that joins events taken from the current state of windows ($win1$ and $win2$). From the definition, the scope (a duration of the data window) for each window applied for each stream can be different. Furthermore both data windows need to be synchronized, which means they have to produce their output events at the same time. To deal with this requirement a merge operation ($Merge$) is introduced as an internal operation inside the join operation. The merge operation aggregates events from two streams into a new stream in order to create a tick stream - a stream that is used as a trigger for execution of stream operations - for both data windows. The definition of the merge operation is defined as follows.

```
$(* fn : int * STREAMID -> 'a\ EVENT\ list -> 'a\ EVENT\ list$
  $-> 'a\ EVENT\ list *)$
$fun\ Merge(sn, sid)\ [\,]\ [\,] = [\,]$
$|\ Merge(sn, sid)\ (evt1 :: S1)\ [\,] = evt1 :: Merge(sn+1, sid)\ S1\ [\,]$
$|\ Merge(sn, sid)\ [\,]\ (evt2 :: S2) = evt2 :: Merge(sn+1, sid)\ [\,]\ S2$
$|\ Merge(sn, sid)\ (evt1 :: S1)\ (evt2 :: S2) =$
        $if\ tStampGT\ evt1\ evt2\ then$
            $evt2 :: Merge(sn+1, sid)\ (evt1 :: S1)\ S2$
        $else$
            $if\ tStampEQ\ evt1\ evt2\ then$
                $evt1 :: Merge(sn+1, sid)\ S1\ S2$
            $else$
                $evt1 :: Merge(sn+1, sid)\ S1\ (evt2 :: S2)$
```

3.3.6.2 Time-window join

One of the most common window-based join operations is a sliding time-window join. A time-window join is a binary operation that combines stream events from two streams and returns a combination of stream events that satisfy a join condition. The definition of the time-window join operation is presented in Figure 3.20.

Stream operation: Time-window join - $JoinTW(w1, w2, J, nj, sn, sid)$
Let $S1, S2$ be streams of type $'a\ EVENT\ list$
Let $w1, w2$ be duration of the time windows
Let J be a join condition that joins two input streams
Let nj be a sequence number generated according to the number of output events
Let sn be a sequence number of the merge operation
Let sid be a stream identifier of an output stream
$JoinTW(w1, w2, J, nj, sn, sid)\ S1\ S2$ is defined as:

```
$(* fn : * TIME * TIME * ('a\ list * 'a\ list * 'b\ list -> ''c\ list) * int * int$
 $* STREAMID -> 'a\ EVENT\ list -> 'a\ EVENT\ list -> ''c\ list\ EVENT\ list *)$

$fun\ JoinTW(w1, w2, J, nj, sn, sid)\ S1\ S2 = Join(w1, w2, J, nj, sn, sid)\ TW_t\ S1\ S2;$
```

FIGURE 3.20: The definition of the time-window join

As presented in the definition of the time-window join (Figure 3.20), the $Join$ function is used as the base function of the time-window join operation. The time-window join utilizes window duration of the time windows ($w1$ and $w2$), a join condition (J), two sequence numbers (nj and sn) and an output stream identifier as the function parameters. This operation takes events from two streams ($S1$ and $S2$) as input. Each time that a new event from either streams arrives the operation returns a set of output events that joins events taken from the current state of time windows. Joining between two streams is performed according to a join condition J. Finally, the operation produces an output event which is the list of pairs of joining events that satisfy that join condition.

3.3.6.3 Length-window join

Another type of join operation is a length-window join. Instead of using time-based windows, count-based windows are used as internal data windows. Stream events from two length windows are combined and output events are produced according to a join condition. The definition of the length-window join operation is presented in Figure 3.21.

Stream operation: Length-window join - $JoinLW(l1, l2, J, nj, sn, sid)$
Let $S1, S2$ be streams of type $'a\ EVENT\ list$
Let $l1, l2$ be size of the length windows
Let J be a join condition that joins two input streams
Let nj be a sequence number generated according to the number of output events
Let sn be a sequence number of the merge operation
Let sid be a stream identifier of an output stream
$JoinLW(l1, l2, J, nj, sn, sid)\ S1\ S2$ is defined as:

$$(*\ fn\ :\ int\ *\ int\ *\ ('a\ list\ *\ 'a\ list\ *\ 'b\ list\ -> \ ''c\ list)\ *\ int\ *\ int\ *\ STREAMID$$
$$->\ 'a\ EVENT\ list\ ->\ 'a\ EVENT\ list\ ->\ ''c\ list\ EVENT\ list\ *)$$

$$fun\ JoinLW(l1, l2, J, nj, sn, sid)\ \ S1\ \ S2\ =\ Join(l1, l2, J, nj, sn, sid)\ \ LW_t\ \ S1\ \ S2;$$

FIGURE 3.21: The definition of the length-window join

As illustrated in Figure 3.21, the length-window join takes events from two streams ($S1$ and $S2$) as input. Each time that a new event from either streams arrives the operation returns a set of output events that combines events taken from the current state of length windows. Joining between two streams is performed according to a join condition J. With this result the content of each individual output event is the list of pairs of joining events that satisfy that join condition.

3.4 Stream ancestor functions

Having described the specifications of primitive stream operations, we now present a list of specifications of stream ancestor functions for primitive stream operations. The stream ancestor functions (SAFs) are utilized as our crucial mechanism to explicitly express dependency relationships between input and output elements of stream operations. In addition, in order to simplify the specifications, the stream ancestor functions that will be presented use event keys as inputs and outputs of the functions instead of provenance assertions.

3.4.1 Additional shared functions

In this section some additional shared functions defined especially for working with stream ancestor functions are presented. These shared functions include $TScope_A$, $NScope_A$, *delayedExactly*, $delayedExactly_A$ and $Trim$. The $TScope_A$ (the compact version of the $TScope$) and $NScope_A$ are used to screen elements of a list ($KEY\ list$). They are defined specifically for using with optimized stream ancestor functions. The *delayedExactly* and $delayedExactly_A$ (the compact version of the *delayedExactly*) are internal functions of stream ancestor functions utilized to compare timestamps between two stream elements.

Furthermore, the $Trim$ function is a function used to describe how to transform an original stream event to the stream event that only consists of an event key. The function is applied to each stream event before it is sent to the provenance service by a stream component. Taking an input event the function returns a compact stream event where the content of each event is discarded. The definitions of all additional shared functions for stream ancestor functions are presented as follows.

```
$(*\ fn\ :\ TIME\ *\ TIME\ *\ KEY\ list -> KEY\ list\ *)$
$fun\ TScope_A(ub, lb, [\,])\ =\ [\,]$
$|\ TScope_A(ub, lb, ((key\ as\ Key(t, \_, \_, \_)) :: B))\ =$
    $if\ (ub\ GTE\ t)\ andalso\ (t\ GTE\ lb)\ then$
        $key :: TScope_A(ub, lb, B)$
    $else\ \ TScope_A(ub, lb, B)$

$(*\ fn\ :\ int\ *\ int\ *\ KEY\ list -> KEY\ list\ *)$
$fun\ NScope_A(ub, lb, [\,])\ =\ [\,]$
$|\ NScope_A(ub, lb, ((key\ as\ Key(\_, n, \_, \_)) :: B))\ =$
    $if\ (ub\ >=\ n)\ andalso\ (n\ >=\ lb)\ then$
        $key :: NScope_A(ub, lb, B)$
    $else\ \ NScope_A(ub, lb, B)$
```

```
$(* fn\ :\ 'a\ EVENT -> 'b\ EVENT -> bool\ *)$
$fun\ delayedExactly\ (Event(Key(t_i, \_, \_, \_), \_))\ (Event(Key(t, \_, \_, d), \_))$
        $=\ ((t - -d)\ =\ t_i)$

$(* fn\ :\ KEY -> KEY -> bool\ *)$
$fun\ delayedExactly_A\ (Key(t_i, \_, \_, \_))\ (Key(t, \_, \_, d))\ =\ ((t - -d)\ =\ t_i)$

$(* fn\ :\ 'a\ EVENT\ list\ \rightarrow\ KEY\ list\ *)$
$fun\ Trim\ [\,]\ =\ [\,]$
$|\ Trim\ ((Event(Key(t, n, sid, d), e)) :: S)\ =\ (Key(t, n, sid, d)) :: Trim\ S;$
```

3.4.2 The stream ancestor function for a map operation

The stream ancestor function for a map operation is defined in Figure 3.22. For this stream ancestor function a list of input stream events S is defined as an input parameter. Because a map operation applies on a single event at a time, after taking an output event as an input, the function returns an input event that contributes to the output.

Stream ancestor function: SAF for a map operation - $Map_A(S)$
Let *event* be an output stream event $(Event(Key(t, n, sid, d), e))$ of type $'b\ EVENT$
Let S be a list of input stream events of type $'a\ EVENT\ list$
For a map operation, the stream ancestor function is defined as:

```
$(* fn\ :\ 'a\ EVENT\ list -> 'b\ EVENT -> 'a\ EVENT\ *)$
$fun\ Map_A(event_i :: S)\ event\ =$
        $if\ delayedExactly\ event_i\ \ event\ \ then\ \ event_i$
        $else\ Map_A(S)\ \ event$
```

FIGURE 3.22: The definition of the stream ancestor function for a map operation

To further enhance the function, an *optimized stream ancestor function* (optimized SAF) is introduced as illustrated in Figure 3.23. The optimized stream ancestor function for a map operation requires a list of event keys (S_m) to be stored in order to use it for processing. By using $delayedExactly_A$ as the internal function to determine time dependencies between event keys, the event key of the input event that contribute to a particular output event can be exactly identified.

Stream ancestor function: Optimized SAF for a map operation - $Map_{OA}(S_m)$
Let *key* be a key of an output stream event $(Key(t, n, sid, d))$ of type KEY
Let S_m be a list of input event keys - $S_m = Trim(S)$
The optimized stream ancestor function for a map operation can be defined as:

```
$(* fn\ :\ KEY\ list -> KEY -> KEY\ *)$
$fun\ Map_{OA}(key_i :: S_m)\ key\ =$
        $if\ delayedExactly_A\ \ key_i\ \ key\ \ then\ \ key_i$
        $else\ Map_{OA}(S_m)\ \ key$
```

FIGURE 3.23: The definition of the optimized SAF for a map operation

As shown in the definition of stream ancestor functions, there are many shared expressions and statements between the unoptimized and the optimized versions of stream ancestor functions. In order to avoid code repetitions caused by presenting both versions of stream ancestor functions, for the stream ancestor functions that will be presented later in this chapter, the optimized stream ancestor functions are only presented.

An example output produced by the optimized stream ancestor function is presented as follows. We first show the output for the map operation. This is followed by the example output for the optimized stream ancestor function. Suppose that the operation takes parameters: F = double (function) and the key of a particular output event used as the input for the stream ancestor function is $(Key(Time\ 32, 4, StreamID\ 2, Time\ 1))$. Figure 3.24 illustrates the processing flow for this example.

$(*\ input\ stream\ *)$
$-\ val\ S\ =\ [Event(Key(Time\ 1, 1, StreamID\ 1, Time\ 0), 55),$
$Event(Key(Time\ 11, 2, StreamID\ 1, Time\ 0), 60),$
$Event(Key(Time\ 21, 3, StreamID\ 1, Time\ 0), 65),$
$Event(Key(Time\ 31, 4, StreamID\ 1, Time\ 0), 70),$
$Event(Key(Time\ 41, 5, StreamID\ 1, Time\ 0), 75)]\ :\ int\ EVENT\ list;$

$(*fn\ :\ int\ \rightarrow\ int\ *)$
$-\ fun\ double(x)\ =\ x + x;\ (*\ double\ function\ *)$

$(*\ execute\ the\ operation\ *)$
$(*\ Map(F, sid) :\ ('a\ \rightarrow\ 'b)\ *\ STREAMID\ \rightarrow\ 'a\ EVENT\ list\ \rightarrow\ 'b\ EVENT\ list\ *)$
$-\ Map(double, StreamID(2))\ S;$

$(*\ output\ stream\ *)$
$>\ [Event(Key(Time\ 2, 1, StreamID\ 2, Time\ 1), 110),$
$Event(Key(Time\ 12, 2, StreamID\ 2, Time\ 1), 120),$
$Event(Key(Time\ 22, 3, StreamID\ 2, Time\ 1), 130),$
$Event(Key(Time\ 32, 4, StreamID\ 2, Time\ 1), 140),$
$Event(Key(Time\ 42, 5, StreamID\ 2, Time\ 1), 150)]\ :\ int\ EVENT\ list$

For the optimized stream ancestor function:

$(*\ input\ stream\ recorded*)$
$-\ val\ S_m\ =\ [Key(Time\ 1, 1, StreamID\ 1, Time\ 0),$
$Key(Time\ 11, 2, StreamID\ 1, Time\ 0),$
$Key(Time\ 21, 3, StreamID\ 1, Time\ 0),$
$Key(Time\ 31, 4, StreamID\ 1, Time\ 0),$
$Key(Time\ 41, 5, StreamID\ 1, Time\ 0)]\ :\ KEY\ list$

$(*\ execute\ the\ stream\ ancestor\ function\ *)$
$-\ Map_{OA}(S_m)\ Key(Time 32, 4, StreamID\ 2, Time\ 1)$
$(*\ ancestor\ events\ *)$
$>\ Key(Time 31, 4, StreamID\ 1, Time\ 0)\ :\ KEY$

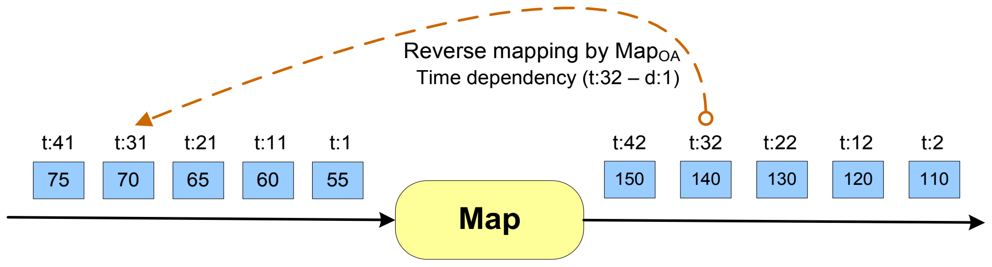


FIGURE 3.24: Example processing flow of a map operation and its SAF

3.4.3 The stream ancestor function for a filter operation

The optimized stream ancestor function for a filter operation is defined in Figure 3.25. This optimized stream ancestor function utilizes a list of input event keys (S_m) as the function parameter. The function takes a key of an output event of the filter operation as an input and returns a key of an input event that contributes to the output. Because the stream ancestor function for both map and filter operations use the same business logic, we utilize the optimized stream ancestor function for a map operation (Map_{OA}) as a core function for this optimized stream ancestor function.

Stream ancestor function: Optimized SAF for a filter operation - $Filter_{OA}(S_m)$
Let key be a key of an output stream event ($Key(t, n, sid, d)$) of type KEY
Let S_m be a list of input event keys - $S_m = Trim(S)$
The optimized stream ancestor function for a filter operation can be defined as:

$(*\ fn\ :\ KEY\ list\ ->\ KEY\ ->\ KEY\ *)$
$fun\ Filter_{OA}(S_m)\ key\ =\ Map_{OA}(S_m)\ key$

FIGURE 3.25: The definition of the optimized stream ancestor function for a filter

An example output produced by the optimized stream ancestor function is presented as follows. We first demonstrate the output for the filter operation. After that, the output for the optimized ancestor function is presented. Suppose that the operation takes input parameters: P = filterCond (function) and $sn = 1$, and the key of an output event used as the input for the stream ancestor function is ($Key(Time\ 22, 2, StreamID\ 2, Time\ 1)$). Figure 3.26 presents the processing flow for this example.

$(*\ input\ stream\ *)$
$-\ val\ S\ =\ [Event(Key(Time\ 1, 1, StreamID\ 1, Time\ 0), 55),$
$Event(Key(Time\ 11, 2, StreamID\ 1, Time\ 0), 60),$
$Event(Key(Time\ 21, 3, StreamID\ 1, Time\ 0), 65),$
$Event(Key(Time\ 31, 4, StreamID\ 1, Time\ 0), 70),$
$Event(Key(Time\ 41, 5, StreamID\ 1, Time\ 0), 75)]\ :\ int\ EVENT\ list$

$(*\ fn\ :\ int\ \rightarrow\ bool\ *)$
$-\ fun\ filterCond(e)\ =\ (e >= 60)\ (*\ filter\ condition\ *)$

$(*\ execute\ the\ operation\ *)$
$(*\ Filter(P, sn, sid) :\ ('a\ \rightarrow\ bool)\ *\ int\ *\ STREAMID\ \rightarrow$
$*\ 'a\ EVENT\ list\ \rightarrow\ 'a\ EVENT\ list\ *)$
$-\ Filter(filterCond, 1, StreamID(2))\ S;$

$(*\ output\ stream\ *)$
$>\ [Event(Key(Time\ 12, 1, StreamID\ 2, Time\ 1), 60),$
$Event(Key(Time\ 22, 2, StreamID\ 2, Time\ 1), 65),$
$Event(Key(Time\ 32, 3, StreamID\ 2, Time\ 1), 70),$
$Event(Key(Time\ 42, 4, StreamID\ 2, Time\ 1), 75)]\ :\ int\ EVENT\ list$

For the optimized stream ancestor function:

$(*\ input\ stream\ recorded*)$
$-\ val\ S_m\ =\ [Key(Time\ 1, 1, StreamID\ 1, Time\ 0),$
$Key(Time\ 11, 2, StreamID\ 1, Time\ 0),$
$Key(Time\ 21, 3, StreamID\ 1, Time\ 0),$
$Key(Time\ 31, 4, StreamID\ 1, Time\ 0),$
$Key(Time\ 41, 5, StreamID\ 1, Time\ 0)]\ :\ KEY\ list$

$(*\ execute\ the\ stream\ ancestor\ function\ *)$
$-\ Filter_{OA}(S_m)\ (Key(Time\ 22, 2, StreamID\ 2, Time\ 1))$
$(*\ ancestor\ events\ *)$
$>\ Key(Time\ 21, 3, StreamID\ 1, Time\ 0)\ :\ KEY$

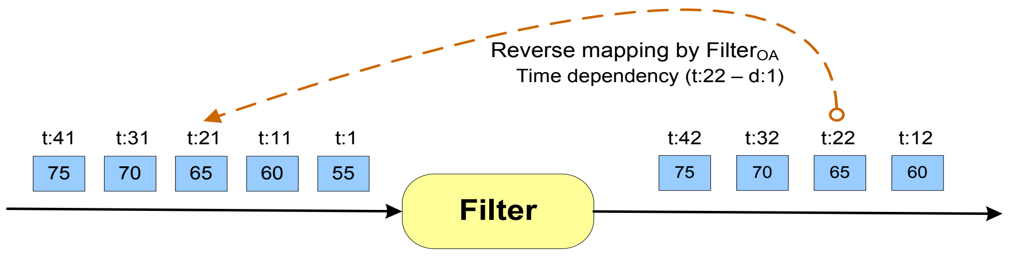

Reverse mapping by Filter$_{OA}$
Time dependency (t:22 – d:1)
t:41
75
t:31
70
t:21
65
t:11
60
t:1
55
Filter
t:42
75
t:32
70
t:22
65
t:12
60

FIGURE 3.26: Example processing flow of a filter operation and its SAF

3.4.4 The stream ancestor function for a sliding time window

The definition of the optimized stream ancestor function for a time window is presented in Figure 3.27. For this optimized stream ancestor function a list of input event keys S_m and the duration of the time window w are indicated as the function parameters. The function takes an event key of the output event generated from a sliding time window as an input and returns a set of event keys of the input events in the extent of the time window at the time that the operation produced the output. As shown in the definition, the optimized SAF for a time window utilizes parameters - the size of the data window, a timestamp and a delay time - in order to define the extent of a past data window which a particular output element is generated from. The interval of the past data window is between t - d - w (lower bound) and t - d (upper bound).

Stream ancestor function: Optimized SAF for a time window - $TW_{OA}(w, S_m)$
Let Key be a key of an output stream event $(Key(t, n, sid, d))$ of type KEY
Let S_m be a list of input event keys - $S_m = Trim(S)$
Let w be the duration of the time window
The optimized stream ancestor function for a sliding time window can be defined as:

$$(*\, fn\ :\ TIME * KEY\ list \rightarrow KEY \rightarrow KEY\ list\ *)$$

$$fun\ TW_{OA}(w, S_m)\ (Key(t, \_, \_, d))\ =\ TScope_A(t - -d, (t - -d) - -w, S_m)$$

FIGURE 3.27: The definition of the optimized SAF for a time window operation

An example output produced by the optimized stream ancestor function is now presented. We first show the output for the sliding time window. This is followed by the example output for the stream optimized ancestor function. Suppose that the operation takes parameters: w = Time(20) and B = [ ], and the key of a particular output event used as the input for the stream ancestor function is *(Key(Time 42,5,StreamID 2,Time 1))*. Figure 3.28 illustrates the processing flow for this example.

$$
\begin{aligned}
&(*\ input\ stream\ *)\\
&-\ val\ S\ =\ [Event(Key(Time\ 1, 1, StreamID\ 1, Time\ 0), 55),\\
&\qquad\qquad Event(Key(Time\ 11, 2, StreamID\ 1, Time\ 0), 60),\\
&\qquad\qquad Event(Key(Time\ 21, 3, StreamID\ 1, Time\ 0), 65),\\
&\qquad\qquad Event(Key(Time\ 31, 4, StreamID\ 1, Time\ 0), 70),\\
&\qquad\qquad Event(Key(Time\ 41, 5, StreamID\ `1, Time\ 0), 75)]\ :\ int\ EVENT\ list;
\end{aligned}
$$

$$
\begin{aligned}
&(*\ execute\ the\ operation\ *)\\
&(*\ TW(w, B, sid) : \ TIME *' a\ EVENT\ list\ *\ STREAMID \rightarrow\\
&\ 'a\ EVENT\ list \rightarrow' a\ list\ EVENT\ list\ *)\\
&-\ TW(Time(20), [\,], StreamID(2))\ S;
\end{aligned}
$$

$(*\ output\ stream\ *)$
$> [Event(Key(Time\ 2, 1, StreamID\ 2, Time\ 1), [55]),$
$\quad Event(Key(Time\ 12, 2, StreamID\ 2, Time\ 1), [60, 55]),$
$\quad Event(Key(Time\ 22, 3, StreamID\ 2, Time\ 1), [65, 60, 55]),$
$\quad Event(Key(Time\ 32, 4, StreamID\ 2, Time\ 1), [70, 65, 60]),$
$\quad Event(Key(Time\ 42, 5, StreamID\ 2, Time\ 1), [75, 70, 65])]\ :\ int\ list\ EVENT\ list$

For the optimized stream ancestor function:

$(*\ input\ stream\ recorded*)$
$-\ val\ S_m\ =\ [Key(Time\ 1, 1, StreamID\ 1, Time\ 0),$
$\qquad Key(Time\ 11, 2, StreamID\ 1, Time\ 0),$
$\qquad Key(Time\ 21, 3, StreamID\ 1, Time\ 0),$
$\qquad Key(Time\ 31, 4, StreamID\ 1, Time\ 0),$
$\qquad Key(Time\ 41, 5, StreamID\ 1, Time\ 0)]\ :\ KEY\ list$

$(*\ execute\ the\ stream\ ancestor\ function\ *)$
$-\ TW_{OA}(Time(20), S_m)\ \ (Key(Time\ 42, 5, StreamID\ 2, Time\ 1));$
$(*\ ancestor\ events\ *)$
$> [Key(Time\ 21, 3, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 31, 4, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 41, 5, StreamID\ 1, Time\ 0)]\ :\ KEY\ list$

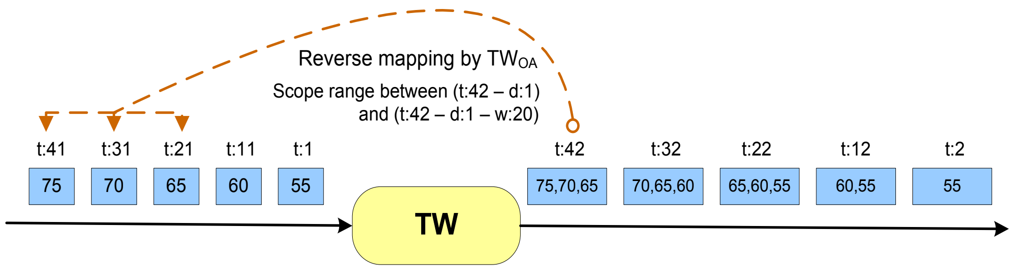


FIGURE 3.28: Example processing flow of a sliding time-window and its SAF

3.4.5 The stream ancestor function for a length window

The optimized stream ancestor function for a length window is presented in Figure 3.29. This optimized SAF is defined as a function that takes a list of event keys (S_m) and the size of the length window l as the function parameters. The function takes an event key of the output event of the length window and returns a set of event keys of the input events in the extent of the length window at the time that the window produced the output. The extent of the past length window is defined by using parameters - the size of the length window (l) and a sequence number (n)- and the interval of the window is between $(n - l) + 1$ (lower bound) and n (upper bound).

Stream ancestor function: Optimized SAF for a length window - $LW_{OA}(l, S_m)$
Let Key be a key of an output stream event $(Key(t, n, sid, d))$ of type KEY
Let S_m be a list of input event keys - $S_m = Trim(S)$
Let l be the size of the length window
The optimized stream ancestor function for a length window can be defined as:

$(*\ fn\ :\ int\ *\ KEY\ list\ \rightarrow\ KEY\ \rightarrow\ KEY\ list\ *)$

$fun\ LW_{OA}(l, S_m)\ (Key(\_, n, \_, \_))\ =\ NScope_A(n, (n - l) + 1, S_m);$

FIGURE 3.29: The definition of the optimized SAF for a length window operation

An example output produced by the optimized stream ancestor function is now presented. We first show the output for the length window and then the example output for the optimized stream ancestor function. Suppose that the operation takes parameters: $l = 4$ and $B = [\,]$, and the key of a particular output event used as the input for the stream ancestor function is *(Key(Time 42,5,StreamID 2,Time 1))*. Figure 3.30 illustrates the processing flow for this example.

$(*\ input\ stream\ *)$
$-\ val\ S\ =\ [Event(Key(Time\ 1, 1, StreamID\ 1, Time\ 0), 55),$
$\quad Event(Key(Time\ 11, 2, StreamID\ 1, Time\ 0), 60),$
$\quad Event(Key(Time\ 21, 3, StreamID\ 1, Time\ 0), 65),$
$\quad Event(Key(Time\ 31, 4, StreamID\ 1, Time\ 0), 70),$
$\quad Event(Key(Time\ 41, 5, StreamID\ 1, Time\ 0), 75)]\ :\ int\ EVENT\ list;$

$(*\ execute\ the\ operation\ *)$
$(*\ LW(l, B, sid) :\ int\ *\ 'a\ EVENT\ list\ *\ STREAMID\ \rightarrow$
$'a\ EVENT\ list\ \rightarrow\ 'a\ list\ EVENT\ list\ *)$
$-\ LW(4, [\,], StreamID(2))\ S;$
$(*\ output\ stream\ *)$
$>\ [Event(Key(Time\ 2, 1, StreamID\ 2, Time\ 1), [55]),$
$\quad Event(Key(Time\ 12, 2, StreamID\ 2, Time\ 1), [60, 55]),$
$\quad Event(Key(Time\ 22, 3, StreamID\ 2, Time\ 1), [65, 60, 55]),$
$\quad Event(Key(Time\ 32, 4, StreamID\ 2, Time\ 1), [70, 65, 60, 55]),$
$\quad Event(Key(Time\ 42, 5, StreamID\ 2, Time\ 1), [75, 70, 65, 60])]\ :\ int\ list\ EVENT\ list$

For the optimized stream ancestor function:

$(*\ input\ stream\ recorded *)$
$-\ val\ S_m\ =\ [Key(Time\ 1, 1, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 11, 2, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 21, 3, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 31, 4, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 41, 5, StreamID\ 1, Time\ 0)]\ :\ KEY\ list$

$(*\ execute\ the\ stream\ ancestor\ function\ *)$
$-\ LW_{OA}(4, S_m)\ (Key(Time\ 42, 5, StreamID\ 2, Time\ 1));$
$(*\ ancestor\ events\ *)$
$>\ [Key(Time\ 11, 2, StreamID\ 1, Time\ 0),$
$\quad [Key(Time\ 21, 3, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 31, 4, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 41, 5, StreamID\ 1, Time\ 0)]\ :\ KEY\ list$

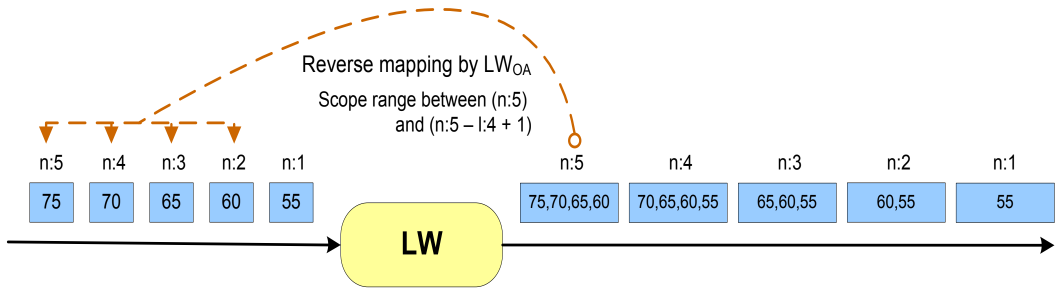


FIGURE 3.30: Example processing flow of a length-window and its SAF

3.4.6 The stream ancestor function for a time-window join operation

The definition of the optimized stream ancestor function for a time-window join operation is presented in Figure 3.31. We define the optimized stream ancestor function for a time-window join operation as a function that takes two lists of input event keys ($S1_m$ and $S2_m$) and the window duration of the time windows ($w1\ and\ w2$) as the function parameters. The function takes an event key of the output event of the join operation as an input and returns a set of event keys of the input events in the extent of the time windows at the time that the output is produced. This optimized stream ancestor function identifies all possible event keys of the input events that are involved in the processing of the join operation for each output event.

Stream ancestor function : Optimized SAF for a time-window join
- $JoinTW_{OA}(w1, w2, S1_m, S2_m)$
Let Key be a key of an output stream event ($Key(t, n, sid, d)$) of type KEY
Let $S1_m, S2_m$ be lists of input event keys - $S1_m = Trim(S1)$ and $S2_m = Trim(S2)$
Let $w1, w2$ be the window duration of the time windows
The optimized SAF for a time-window join operation can be defined as:

$(*\ fn\ :\ TIME\ *\ TIME\ *\ KEY\ list\ *\ KEY\ list\ ->\ KEY\ ->\ KEY\ list\ *)$

$fun\ JoinTW_{OA}(w1, w2, S1_m, S2_m)\ (Key(t, \_, \_, d))\ =$
$\quad TScope_A(t - -d, (t - -d) - -w1, S1_m)\ @\ TScope_A(t - -d, (t - -d) - -w2, S2_m)$

FIGURE 3.31: The definition of the optimized SAF for a time-window join operation

An example output produced by the optimized stream ancestor function is presented as follows. We first demonstrate the output for the time-window join operation. This is followed by the example output for the optimized stream ancestor function. Suppose that the operation takes input parameters: $w1$ *and* $w2$ = Time(10), J = cartesianJoin (function), $nj = 1$ and $sn = 1$, and the key of a particular output event used as the input for the stream ancestor function is $(Key(Time\ 44, 9, StreamID\ 3, Time\ 1))$. Figure 3.32 presents the processing flow for this example.

```
(* input streams *)
- val S1 = [Event(Key(Time 1, 1, StreamID 1, Time 0), 55),
            Event(Key(Time 11, 2, StreamID 1, Time 0), 60),
            Event(Key(Time 21, 3, StreamID 1, Time 0), 65),
            Event(Key(Time 31, 4, StreamID 1, Time 0), 70),
            Event(Key(Time 41, 5, StreamID 1, Time 0), 75)] : int EVENT list;

- val S2 = [Event(Key(Time 2, 1, StreamID 2, Time 0), 70),
            Event(Key(Time 12, 2, StreamID 2, Time 0), 75),
            Event(Key(Time 22, 3, StreamID 2, Time 0), 80),
            Event(Key(Time 32, 4, StreamID 2, Time 0), 65),
            Event(Key(Time 42, 5, StreamID 2, Time 0), 85)] : int EVENT list;

(* execute the operation *)
(* JoinTW(w1, w2, J, nj, sn, sid)) : TIME * TIME
* ('a list * 'a list * 'b list -> ''c list) * int * int * STREAMID
 -> 'a EVENT list -> 'a EVENT list -> ''c list EVENT list *)
- JoinTW(Time(10), Time(10), cartesianJoin, 1, 1, StreamID(3)) S1 S2;

(* output stream *)
> [Event(Key(Time 4, 1, StreamID 3, Time 2), [(55, 70)]),
  Event(Key(Time 13, 2, StreamID 3, Time 2), [(60, 70), (55, 70)]),
  Event(Key(Time 14, 3, StreamID 3, Time 2), [(60, 75), (60, 70)]),
  Event(Key(Time 23, 4, StreamID 3, Time 2), [(65, 75), (60, 75)]),
  Event(Key(Time 24, 5, StreamID 3, Time 2), [(65, 80), (65, 75)]),
  Event(Key(Time 33, 6, StreamID 3, Time 2), [(70, 80), (65, 80)]),
  Event(Key(Time 34, 7, StreamID 3, Time 2), [(70, 65), (70, 80)]),
  Event(Key(Time 43, 8, StreamID 3, Time 2), [(75, 65), (70, 65)]),
  Event(Key(Time 44, 9, StreamID 3, Time 2), [(75, 85), (75, 65)])]
  : (int * int) list EVENT list
```

For the optimized stream ancestor function:

$(*\ input\ streams\ recorded*)$
$-\ val\ S1_m\ =\ [Key(Time\ 1, 1, StreamID\ 1, Time\ 0),$
$Key(Time\ 11, 2, StreamID\ 1, Time\ 0),$
$Key(Time\ 21, 3, StreamID\ 1, Time\ 0),$
$Key(Time\ 31, 4, StreamID\ 1, Time\ 0),$
$Key(Time\ 41, 5, StreamID\ 1, Time\ 0)]\ :\ KEY\ list$

$-\ val\ S2_m\ =\ [Key(Time\ 2, 1, StreamID\ 2, Time\ 0),$
$Key(Time\ 12, 2, StreamID\ 2, Time\ 0),$
$Key(Time\ 22, 3, StreamID\ 2, Time\ 0),$
$Key(Time\ 32, 4, StreamID\ 2, Time\ 0),$
$Key(Time\ 42, 5, StreamID\ 2, Time\ 0)]\ :\ KEY\ list$

$(*\ execute\ the\ stream\ ancestor\ function\ *)$
$-\ JoinTW_{OA}(Time(10), Time(10), S1_m, S2_m)\ (Key(Time\ 44, 9, StreamID\ 3, Time\ 2));$

$(*\ ancestor\ events\ *)$
$>\ [Key(Time\ 41, 5, StreamID\ 1, Time\ 0),$
$Key(Time\ 32, 4, StreamID\ 2, Time\ 0),$
$Key(Time\ 42, 5, StreamID\ 2, Time\ 0)]\ :\ KEY$

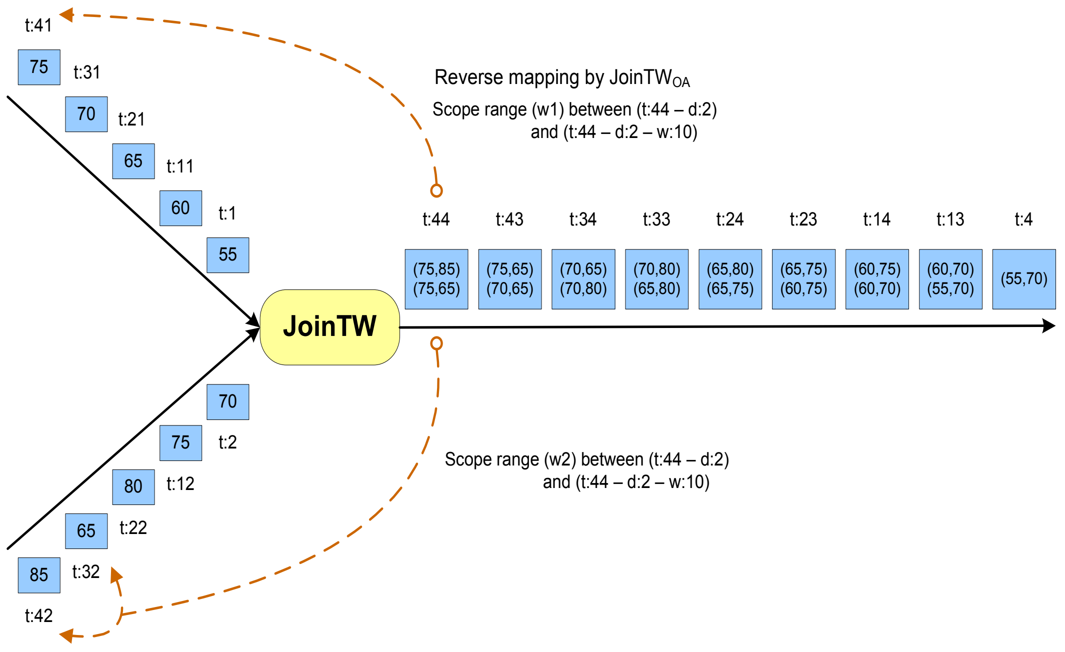

t:41
75
t:31
70
t:21
65
t:11
60
t:1
55
Reverse mapping by JoinTW$_{OA}$
Scope range (w1) between (t:44 – d:2) and (t:44 – d:2 – w:10)
t:44 t:43 t:34 t:33 t:24 t:23 t:14 t:13 t:4
(75,85) (75,65)
(75,65) (70,65)
(70,65) (70,80)
(70,80) (65,80)
(65,80) (65,75)
(65,75) (60,75)
(60,75) (60,70)
(60,70) (55,70)
(55,70)
JoinTW
70
75
t:2
80
t:12
65
t:22
85
t:32
t:42
Scope range (w2) between (t:44 – d:2) and (t:44 – d:2 – w:10)

FIGURE 3.32: Example processing flow of a time-window join operation and its SAF

3.4.7 The stream ancestor function for a length-window join operation

Stream ancestor function: Optimized SAF for a length-window join
- $JoinLW_{OA}(l1, l2, S1_m, S2_m)$
Let Key be a key of an output stream event $(Key(t, n, sid, d))$ of type KEY
Let $S1_m, S2_m$ be lists of input event keys - $S1_m = Trim(S1)$ and $S2_m = Trim(S2)$
Let $l1, l2$ be the size of the length windows
The optimized stream ancestor function for a length-window join can be defined as:

```
(* fn : int * int * KEY list * KEY list -> KEY -> KEY list *)
fun JoinLW_OA(l1, l2, S1_m, S2_m) key =
let
    val ub1 = upperBound_A(key, S1_m)
    val lb1 = (ub1 - l1) + 1
    val ub2 = upperBound_A(key, S2_m)
    val lb2 = (ub2 - l2) + 1
in
    NScope_A(ub1, lb1, S1_m) @ NScope_A(ub2, lb2, S2_m)
end

(* fn : KEY * KEY list -> int *)
fun upperBound_A((Key(t, _, _, d)), S_m) = getMaxSeqNo_A(0, timeFilter_A(t - -d, S_m))

(* fn : int * KEY list -> int *)
fun getMaxSeqNo_A(mx, [ ]) = mx
| getMaxSeqNo_A(mx, ((Key(_, n, _, _)) :: B)) =
        if(n > mx) then getMaxSeqNo_A(n, B)
        else getMaxSeqNo_A(mx, B);

(* fn : TIME * KEY list -> KEY list *)
fun timeFilter_A(ts, [ ]) = [ ]
| timeFilter_A(ts, ((Key(t, n, sid, d)) :: B)) =
        if(ts GTE t) then (Key(t, n, sid, d)) :: timeFilter_A(ts, B)
        else timeFilter_A(ts, B);
```

FIGURE 3.33: The definition of the optimized SAF for a length-window join operation

As illustrated in Figure 3.33, we define the optimized stream ancestor function for a length-window join as a function that takes two lists of input event keys ($S1_m$ *and* $S2_m$) and the size of the length windows ($l1$ *and* $l2$) as the function parameters. The function takes an event key of the output event of a length-window join operation as an input and returns a set of event keys of the input events in the extent of the length windows at the time that the output is produced. This optimized stream ancestor function identifies all possible event keys of the input events that are involved in the processing of the join operation for each output event. In addition, $getMaxSeqNo_A$ is an internal function responsible for getting the largest or the maximum sequence number in a given key list (a data buffer B). Another internal function is $timeFilter_A$ which is used to filter elements of a key list. This function only returns a list of elements (event keys) that their timestamp are less than or equal to a specific time (ts). The $upperBound_A$ is used by the optimized stream ancestor function to determine the upper bound of the past data window.

An example output produced by the optimized stream ancestor function is presented as follows. We first present the output for the length-window join and then the output for the optimized stream ancestor function. Suppose that the operation takes parameters: $l1$ and $l2 = 2$, $J =$ cartesianJoin (function), $nj = 1$ and $sn = 1$, and the key of a particular output event used as the input for the ancestor function is $(Key(Time\ 44, 9, StreamID\ 3, Time\ 1))$. Figure 3.34 illustrates the processing flow for this example.

$(*\ input\ streams\ *)$
$-\ val\ S1\ =\ [Event(Key(Time\ 1, 1, StreamID\ 1, Time\ 0), 55),$
$\quad Event(Key(Time\ 11, 2, StreamID\ 1, Time\ 0), 60),$
$\quad Event(Key(Time\ 21, 3, StreamID\ 1, Time\ 0), 65),$
$\quad Event(Key(Time\ 31, 4, StreamID\ 1, Time\ 0), 70),$
$\quad Event(Key(Time\ 41, 5, StreamID\ 1, Time\ 0), 75)]\ :\ int\ EVENT\ list;$

$-\ val\ S2\ =\ [Event(Key(Time\ 2, 1, StreamID\ 2, Time\ 0), 70),$
$\quad Event(Key(Time\ 12, 2, StreamID\ 2, Time\ 0), 75),$
$\quad Event(Key(Time\ 22, 3, StreamID\ 2, Time\ 0), 80),$
$\quad Event(Key(Time\ 32, 4, StreamID\ 2, Time\ 0), 85),$
$\quad Event(Key(Time\ 42, 5, StreamID\ 2, Time\ 0), 90)]\ :\ int\ EVENT\ list;$

$(*\ execute\ the\ operation\ *)$
$(*\ JoinLW(l1, l2, J, nj, sn, sid))\ :\ int\ *\ int$
$*\ ('a\ list\ *\ 'a\ list\ *\ 'b\ list\ \rightarrow\ ''c\ list)\ *\ int\ *\ int\ *\ STREAMID$
$\rightarrow\ 'a\ EVENT\ list\ \rightarrow\ 'a\ EVENT\ list\ \rightarrow\ ''c\ list\ EVENT\ list\ *)$
$-\ JoinLW(2, 2, cartesianJoin, 1, 1, StreamID(3))\ S1\ S2;$

$(*\ output\ stream\ *)$
$>\ [Event(Key(Time\ 3, 1, StreamID\ 3, Time\ 1), [(55, 70)]),$
$\quad Event(Key(Time\ 12, 2, StreamID\ 3, Time\ 1), [(60, 70), (55, 70)]),$
$\quad Event(Key(Time\ 13, 3, StreamID\ 3, Time\ 1), [(60, 75), (60, 70), (55, 75), (55, 70)]),$
$\quad Event(Key(Time\ 22, 4, StreamID\ 3, Time\ 1), [(65, 75), (65, 70), (60, 75), (60, 70)]),$
$\quad Event(Key(Time\ 23, 5, StreamID\ 3, Time\ 1), [(65, 80), (65, 75), (60, 80), (60, 75)]),$
$\quad Event(Key(Time\ 32, 6, StreamID\ 3, Time\ 1), [(70, 80), (70, 75), (65, 80), (65, 75)]),$
$\quad Event(Key(Time\ 33, 7, StreamID\ 3, Time\ 1), [(70, 85), (70, 80), (65, 85), (65, 80)]),$
$\quad Event(Key(Time\ 42, 8, StreamID\ 3, Time\ 1), [(75, 85), (75, 80), (70, 85), (70, 80)]),$
$\quad Event(Key(Time\ 43, 9, StreamID\ 3, Time\ 1), [(75, 90), (75, 85), (70, 90), (70, 85)])]$
$\quad :\ (int\ *\ int)\ list\ EVENT\ list$

For the optimized stream ancestor function:

$(*\ input\ streams\ recorded*)$
$-\ val\ S1_m\ =\ [Key(Time\ 1, 1, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 11, 2, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 21, 3, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 31, 4, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 41, 5, StreamID\ 1, Time\ 0)]\ :\ KEY\ list$

$-\ val\ S2_m\ =\ [Key(Time\ 2, 1, StreamID\ 2, Time\ 0),$
$\quad Key(Time\ 12, 2, StreamID\ 2, Time\ 0),$
$\quad Key(Time\ 22, 3, StreamID\ 2, Time\ 0),$
$\quad Key(Time\ 32, 4, StreamID\ 2, Time\ 0),$
$\quad Key(Time\ 42, 5, StreamID\ 2, Time\ 0)]\ :\ KEY\ list$

$(*\ execute\ the\ stream\ ancestor\ function\ *)$
$-\ JoinLW_{OA}(2, 2, S1_m, S2_m)\ (Key(Time\ 43, 9, StreamID\ 3, Time\ 1));$

$(*\ ancestor\ events\ *)$
$>\ [Key(Time\ 31, 4, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 41, 5, StreamID\ 1, Time\ 0),$
$\quad Key(Time\ 32, 4, StreamID\ 2, Time\ 0),$
$\quad Key(Time\ 42, 5, StreamID\ 2, Time\ 0)]\ :\ KEY$

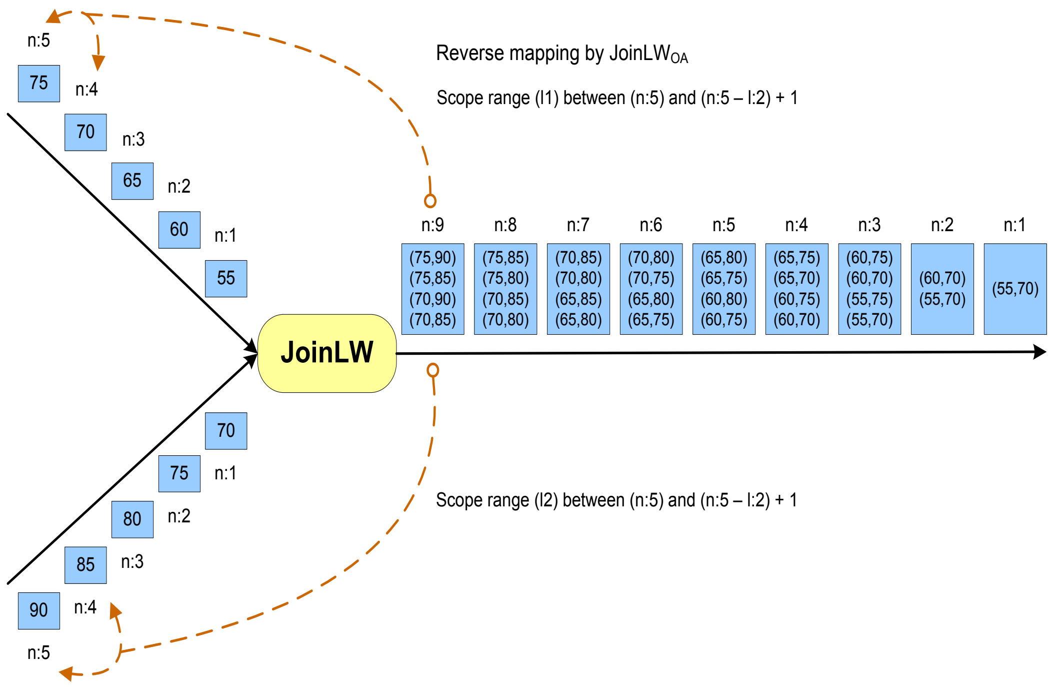


FIGURE 3.34: Example processing flow of a length-window join operation and its SAF

3.5 Summary

We began this chapter by presenting a stream provenance model which describes the fundamental concepts of provenance representation in stream processing systems. Several provenance related issues, for example the basic assumption for fine-grained provenance tracking in stream processing systems, how provenance of individual stream elements can be captured and how dependencies between stream elements can be expressed, were discussed. Based on the stream provenance model, we defined the provenance architecture for stream processing systems. With a logical provenance architecture and a generic fine-grained provenance data model, we can describe the structure of our stream provenance system and also demonstrate the structure of information used to represent the provenance of individual stream processing results.

After fully specifying the stream provenance model, we presented the programmatic specifications for stream operations in the form of recursive functions which were used to describe the processing logic of stream operations. These specifications allow us to precisely describe how the output elements for each stream operation are produced in terms of input elements. We then introduced the stream ancestor functions for stream operations. By using the stream ancestor function defined for each stream operation, dependency relationships between input and output stream events of a stream operations can be explained.

The contributions of this chapter were two fold: firstly, a fine-grained provenance model for stream processing systems and secondly, a set of primitive stream ancestor functions. The first contribution is supported by defining the provenance data model for streams. We not only presented the provenance model which allows the provenance of individual stream elements to be captured but also a provenance architecture for stream processing systems that is designed to comply with the provenance model. This logical provenance architecture allow us to describe internal components in our stream provenance system and interactions between these components in detail. To support the second contribution the programmatic specifications of stream ancestor functions for primitive stream operations were defined by using Standard ML. We demonstrated the use of reverse mapping techniques (in stream ancestor functions) to identify input stream elements that are involved in the production of a particular output stream element. Using a simple example we showed how stream ancestor functions work in practice. In the next chapter, we will demonstrate how to utilize stream ancestor functions together in order to address fine-grained provenance queries in stream processing systems.

Chapter 4

Provenance queries for streams

In the previous chapter, we presented the specifications of stream ancestor functions that can explicitly express dependency relationships between input and output elements of a stream operation. To provide a provenance query capability, or more particularly a mechanism that can capture the complete provenance of individual stream elements (all intermediate stream elements involved in the production of each individual stream element), it is necessary to express all input-output dependencies for all stream operations in a stream processing system. This chapter presents a solution that utilizes stream ancestor functions in order to address the fine-grained provenance queries in stream processing systems. The provenance query mechanism which allows for individual stream elements to be traced and computations to be verified at all processing steps is described. We demonstrate how precise our provenance query mechanism is by establishing that query results returned by our query mechanism can be used to reproduce the original stream processing results using a replay execution method. In addition, we provide a case study to demonstrate how the provenance query and the replay execution method work in practice.

The provenance query solution presented consists of two main contributions:

1. A stream provenance query mechanism which utilizes stream ancestor functions in order to capture the provenance of individual stream elements.
2. A replay execution method used to validate the accuracy of our provenance query mechanism. We also present a specification of the replay execution functionality.

The rest of the chapter is organized as follows. It begins by describing the concept of fine-grained provenance query for streams and how to compose stream ancestor functions together in order to obtain the provenance of stream processing results. Next, a replay execution method which utilizes provenance information (provenance assertions) recorded to address stream reproduction is presented. We then demonstrate an example case study for provenance queries. Finally, the chapter is summarized.

4.1 Fine-grained provenance queries

4.1.1 Composition of stream ancestor functions

This section describes how stream ancestor functions are exploited to address fine-grained provenance tracking in stream processing systems. To utilize stream ancestor functions, it is necessary to compose the stream ancestor functions for all stream operations in a stream system together. This concept has been inspired by function composition - a mechanism that combines simple functions to build more complicated ones. Like the usual composition of functions in mathematics, two or more functions can be composed in a new function that uses the output of one function as the input of another. For example, the functions $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ can be composed by applying f to an argument X to obtain $Y = f(X)$ and then applying g to Y to obtain $Z = g(Y)$. This composition can be described by using the notation: $g \circ f : X \rightarrow Z$.

The concept of function composition can be applied to the stream operations and their stream ancestor functions. In this context, the processing flow of a stream processing system is represented as a set of interconnected nodes where each node represents a stream operation. By composing all nodes (stream operations), the output of the stream processing system can be retrieved. To trace back a particular output stream event produced by a stream processing system, it is necessary to compose stream ancestor functions - reverse mapping functions for stream operations. For each stream ancestor function we can identify input events (provenance assertions that represent stream events) involved in the processing of a particular output stream event. By composing all stream ancestor functions in a stream processing system, we can capture the complete provenance of an individual stream element. This satisfies our primary use-case requirement regarding the tracking of the provenance of individual stream elements.

To demonstrate the concept of how to compose stream ancestor functions together, an example processing flow of a stream processing system is presented in Figure 4.1. The example processing flow (forward functions) begins with input events fed into a stream processing system. The first operation, the input-driven time window (TW), receives the input events and generates each output event from the most recent input events over 20 time units. Then, the event is sent to the map operation (Map) that is responsible for computing average values (using avg function). Finally, the event is submitted to the filter operation (Filter) in order to filter out events that do not meet the filter condition (filterCond - the condition of event value greater than or equal to 60).

Figure 4.1(a) illustrates an example of processing flow in a stream processing system.

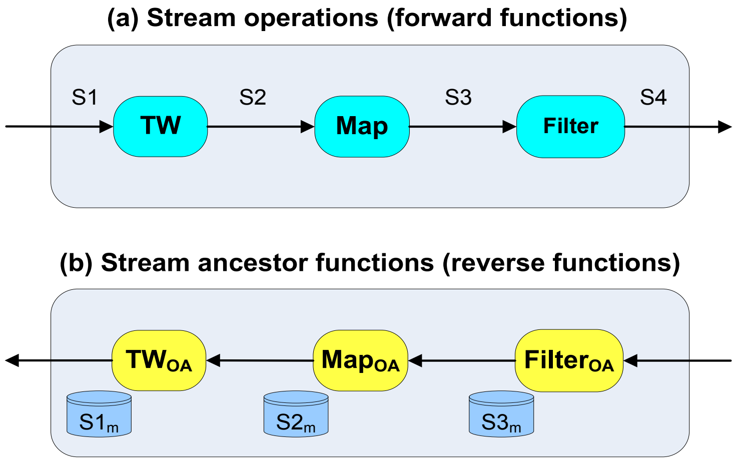

(a) Stream operations (forward functions)
S1
TW
S2
Map
S3
Filter
S4
(b) Stream ancestor functions (reverse functions)
TW_OA
Map_OA
Filter_OA
S1_m
S2_m
S3_m

FIGURE 4.1: An example of the composition of stream ancestor functions

In this example, the processing flow is constructed by composing stream operations including input-driven time window (TW), Map and Filter. For each operation input and output streams are labeled with unique IDs. For example, $S1$ and $S2$ are the input and output streams of the time window operation (TW). The composition of these stream operations to obtain the output stream $S4$ can be demonstrated as follows.

The input stream ($S1$) used in this example:

$$
\begin{aligned}
&(*\ input\ streams\ S1\ *)\\
&-\ val\ S1\ =\ [Event(Key(Time\ 1, 1, StreamID\ 1, Time\ 0), 58),\\
&\qquad\qquad Event(Key(Time\ 11, 2, StreamID\ 1, Time\ 0), 65),\\
&\qquad\qquad Event(Key(Time\ 21, 3, StreamID\ 1, Time\ 0), 70),\\
&\qquad\qquad Event(Key(Time\ 31, 4, StreamID\ 1, Time\ 0), 75),\\
&\qquad\qquad Event(Key(Time\ 41, 5, StreamID\ 1, Time\ 0), 72)]\ :\ int\ EVENT\ list;
\end{aligned}
$$

The predicates (filterCond) used by the filter operation and the input functions (avg) for the map operation:

$$
\begin{aligned}
&(*\ fn\ :\ int -> \ bool\ *)\\
&fun\ filterCond(e)\ =\ (e >= 60)\\
\\
&(*\ fn\ :\ int\ *\ int -> \ int\ list -> \ int\ *)\\
&fun\ avg(B : int, cnt : int)\ [\,]\ =\ round(real(B)\ /\ real(cnt))\\
&|\ avg(B : int, cnt : int)\ (e :: E)\ =\ avg(B + e, cnt + 1)\ E;
\end{aligned}
$$

The output stream ($S4$) can be produced by composing the stream operations as follows:

```
(* fn : int EVENT list -> int EVENT list *)
- val FwdFunction = Filter(filterCond, 1, StreamID(4))
        o Map(avg(0, 0), StreamID(3))
        o TW(Time(20), [ ], StreamID(2));

(* execute the composed stream operations *)
- FwdFunction S1;

(* the output stream (S4) *)
> [Event(Key(Time 14, 1, StreamID 4, Time 1), 62),
   Event(Key(Time 24, 2, StreamID 4, Time 1), 64),
   Event(Key(Time 34, 3, StreamID 4, Time 1), 70),
   Event(Key(Time 44, 4, StreamID 4, Time 1), 72)] : int EVENT list
```

To capture the provenance of the example processing flow, stream ancestor functions are composed as illustrated in Figure 4.1(b). The stream ancestor functions composed include $Filter_{OA}$, Map_{OA} and TW_{OA}. In addition, the information required for the processing of the optimized stream ancestor functions is also presented in Figure 4.1(b). Provenance assertions - representations of individual stream events - which contain only event keys need to be stored in a provenance store. These provenance assertions include $S1_m$ ($S1_m = Trim(S1)$), $S2_m$ ($S2_m = Trim(S2)$) and $S3_m$ ($S3_m = Trim(S3)$). By passing the provenance assertions (event keys) of output events from the output stream $S4$ to the composed stream ancestor functions, a set of provenance assertions (or ancestor assertions) from $S1$ can be identified. The composition of stream ancestor functions can be demonstrated as follows.

```
(* compose the stream ancestor functions *)
- val ancList = [mapf(TW_OA(Time(20), S1_m)), map(Map_OA(S1_m))
                ,map(Filter_OA(S3_m))];

(* fn : KEY list -> KEY list *)
- val RevFunction = AncFun(ancList);

(* utility functions for the composition of stream ancestor functions *)

(* fn : ('a -> 'a) list -> 'a -> 'a *)
fun AncFun [f] = f
| fun AncFun (f :: FL) = f o AncFun FL;

(* fn : ('a -> KEY list) -> 'a list -> KEY list *)
val mapf = fn(f) => flat o map(f);
```

```
(* fn : KEY list list -> KEY list *)
val flat = remdupl o concat;

(* fn : ''a list -> ''a list *)
fun remdupl [ ] = [ ]
| remdupl [x] = [x]
| remdupl (x :: xs) = if mem xs x then remdupl xs
                      else x :: remdupl xs;

(* fn : ''a list -> ''a -> bool *)
fun mem [ ] a = false
| mem (x :: xs) a = a = x orelse mem xs a;

(* fn : KEY list list -> KEY list *)
fun concat [ ] = [ ]
| concat ((x : KEY list) :: list) = x @ (concat list);
```

The example described above shows the straightforward case of the composition of stream ancestor functions. We compose the stream ancestor functions by passing a list of stream ancestor functions (*ancList*) to the function *AncFun* that is responsible for the recursive composition of all stream ancestor functions in the list. Note that *map* is the SML function used for applying a stream ancestor function to all elements in the results of the predecessor stream ancestor function. It is different from *Map* operation that we have defined as a stream operation in the previous chapter which applies an input function to the content of stream events only.

Suppose that we would like to identify which source events in $S1$ contributed to the output events in $S4$. We can pass the provenance assertions (event keys) of these output events to the composed stream ancestor functions *RevFunction*. Figure 4.2 demonstrates the reverse mapping process of the composition of stream ancestor functions in our example. The output of the composed stream ancestor function can be shown as follows:

```
(*execute the composed ancestor functions*)
- RevFunction [Key(Time 34, 3, StreamID 4, Time 1),
               Key(Time 44, 4, StreamID 4, Time 1)];

(*the output of the composed stream ancestor functions*)
> [Key(Time 11, 2, StreamID 1, Time 0),
   Key(Time 21, 3, StreamID 1, Time 0),
   Key(Time 31, 4, StreamID 1, , Time 0),
   Key(Time 41, 5, StreamID 1, Time 0)] : KEY list
```

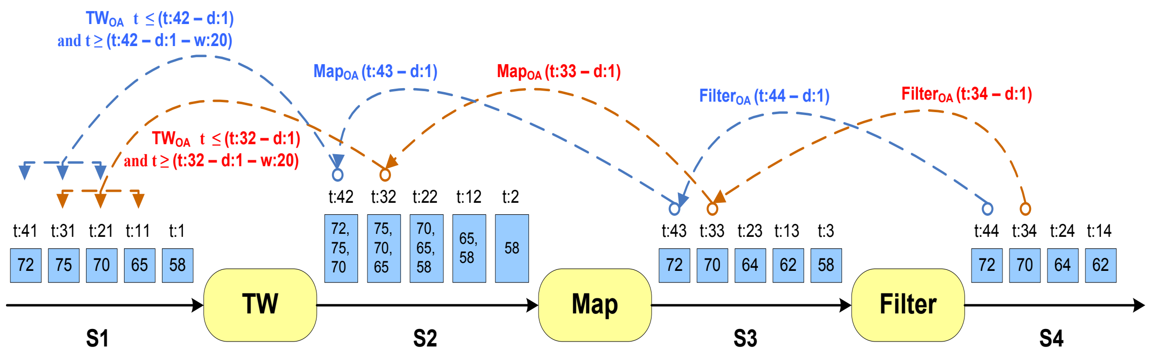


FIGURE 4.2: Mapping process of the composition of stream ancestor functions

4.1.2 Algorithm for a fine-grained provenance query

As demonstrated in the previous section, stream ancestor functions can be statically composed and then the provenance of individual stream elements can be retrieved by using a set of predefined stream ancestor functions. This kind of composition is suitable for stream-based applications or stream processing systems that have simple processing flows such as a linear or a chain topology. However, practical, real-life stream processing systems possess many different forms of stream topologies. In addition, there are some stream-based applications that possess complex internal processing flows. The processing flows of such stream-based applications are generally based on the current state of stream operations (e.g. a number of stream events processed, the occurrence of some types of stream events and the values of contents of previous stream events). It is almost impossible to prepare or predefine stream ancestor functions at system registration time before provenance queries are executed. Therefore, to offer provenance query capability that can be applied to various kinds of stream processing systems, it is important that a provenance query solution for streams satisfies two practical requirements:

1. The composition of stream ancestor functions needs to be performed dynamically at the time that provenance queries are executed. With this requirement we can offer a topology-independent query solution that can be used to precisely obtain the provenance of stream processing results.
2. The provenance query solution should provide an additional technique that allows for provenance queries to be scoped. With this requirement we can delineate information used to answer provenance queries and also we can provide only those pieces of information that users are particularly interested in.

To address the practical requirements mentioned above, we introduced a provenance query algorithm that can perform fine-grained provenance queries over provenance assertions of stream elements. The fundamental concept of this algorithm is that stream ancestor functions are composed dynamically during the execution of provenance queries.

The composition of stream ancestor functions, in this context, is based on the association of stream topology information and event keys contained in each individual provenance assertion. Using this information, our algorithm can reconstitute stream data flows by capturing all representations of interactions - provenance assertions - that take place between stream operations. Such data flows explicitly express data dependencies among intermediate stream elements involved in a stream processing system's execution and they generally form a direct acyclic graph (DAG) - a formal provenance representation. The composition of stream ancestor functions is performed like traversing a graph or more particularly a DAG in reverse order on a node by node basis. Following data dependencies in reverse allows us to understand how a particular stream processing result was generated. Data dependencies among the provenance assertions of stream elements are recursively resolved by the provenance query algorithm. This dependency resolution process is performed continuously until reaching the final node (provenance assertions of the first-input stream).

In addition, the provenance query algorithm also provides a mechanism that allows query users to specify the scope of provenance queries. The scope of a provenance query is generally used to specify which related data items should be included in the provenance query results [94]. More concretely, in the context of stream provenance queries, we can say that, given a particular output stream element as an input, instead of identifying all intermediate stream elements used in the production of that output, which we refer to as the complete provenance of that output, a scoped provenance query identifies the subset of the complete provenance of that output that satisfy a query scope condition. In the query algorithm, we identify the scope of a provenance query by specifying stream IDs of data streams (targeted stream IDs) used to terminate the dependency resolution process of the composition of stream ancestor functions. Figure 4.3 presents an example of how a provenance query is scoped.

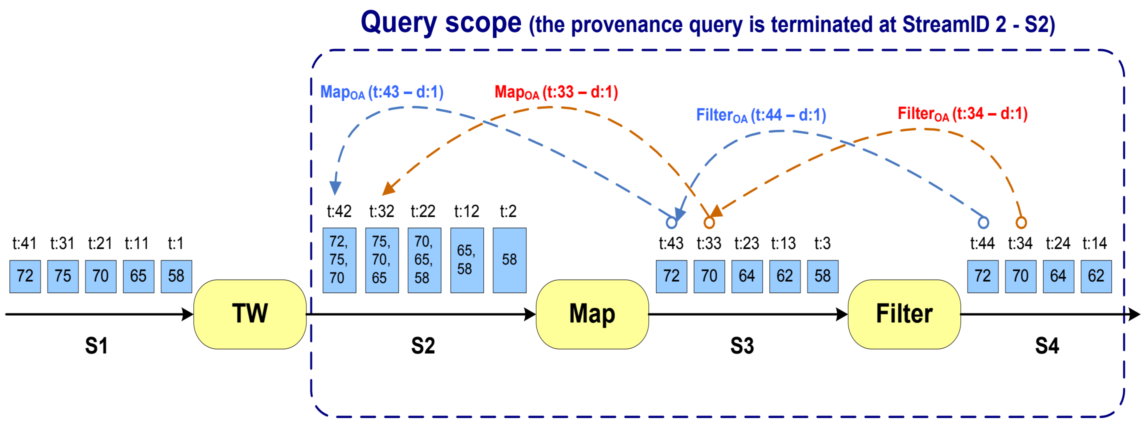


Figure 4.3: Example of how a provenance query is scoped

As illustrated in Figure 4.3, we consider the same example used in the previous section (Section 4.1.1). By specify the stream 2 as the targeted stream for a provenance query, we can terminate the processing of that provenance query and we can obtain provenance query results which are event keys: $Key(Time\ 32, 4, StreamID\ 2, Time\ 1)$ and $Key(Time\ 42, 5, StreamID\ 2, Time\ 1)$ that belong to the targeted stream 2. The algorithm for a stream provenance query is illustrated in Figure 4.4.

Algorithm: Stream provenance query - $retrieveAncestors(safList, sidList)$
Let KL be a list of event keys use as an input for the query
Let $safList$ be a look up table (list) for stream ancestor functions
Let $sidList$ be a list of targeted stream IDs

```
(* fn : (STREAMID * (KEY -> KEYlist)) list * STREAMID list
-> KEY list -> KEY list *)
fun retrieveAncestors(safList, sidList) [] = []
| retrieveAncestors(safList, sidList) KL =
                    composeSAF safList sidList KL []

(* fn : (STREAMID * (KEY -> KEY list)) list
 -> STREAMID list -> KEY list -> KEY list -> KEY list *)
fun composeSAF safList sidList [] R = remdupl R
| composeSAF safList sidList ((key as Key(_, _, sid, _)) :: KL) R =
   if member sid sidList then
      composeSAF safList sidList KL ([key] @ R)
   else
      composeSAF safList sidList (executeSAF safList key KL) R
```

FIGURE 4.4: Algorithm for a stream provenance query

Utility functions for using with composeSAF function

```
(* fn : ''a -> ''a list -> bool *)
fun member e [] = false
| member e (e1 :: l) = (e = e1) orelse member e l;

(* fn : ''a -> (''a * 'b) list -> 'b *)
fun assoc sid ((id, func) :: safList) =
        if (sid = id) then func
        else assoc sid safList;

(* fn : (STREAMID * (KEY -> ''a list)) list -> KEY
 -> ''a list -> ''a list *)
fun executeSAF safList (key as Key(_, _, sid, _)) B =
        B @ remdupl (assoc sid safList key);
```

As presented in Figure 4.4, the query algorithm consists of two internal functions: *retrieveAncestors* and *composeSAF* functions. The *retrieveAncestors* is the entry-point function that takes a list of event keys (KL), a look up table for stream ancestor functions ($safList$) and a list of targeted stream IDs (*sidList* - stream IDs used to terminate or scope the provenance query) as input parameters and returns a set of event keys (provenance assertions) which are the query results. Note that, the look up table for stream ancestor functions is created dynamically by using information from streams, operations and parameters tables described in Section 3.1.3 (the provenance data model). The other function, *composeSAF*, is the recursive function that contains the business logic of the provenance query. The process of the function consists of first receiving parameters passed by the retrieveAncestors function. Then the function iterates recursively over a list of event keys (KL) in order to execute stream ancestor functions on every input event key. For each event key, if its stream ID is not the IDs in the stream ID list (*sidList*), it will be processed by its associated stream ancestor function (using *executeSAF* function). Every output (event key) returned by the stream ancestor function is inserted back to the key list KL. Finally, if there are event keys in the key list KL (meaning there are some intermediate event keys still waiting to be processed), the composeSAF function will be called recursively until no event keys remain in the key list KL.

4.2 Replay execution

The main concept of the replay execution method is that it utilizes provenance assertions and auxiliary information (e.g. stream topology and stream operation parameters) which are stored in a provenance store to address stream reproduction or replay execution. Our replay method is based on the assumption that for a given execution of a stream processing system, output (or stream processing results) are generated in a deterministic way. This means all input events that flow through a processing graph of stream operations (stream processing flow) for a stream system are executed in the same order and non-deterministic behavior is therefore not introduced in stream reproduction. In addition, our replay method does not require a full input stream as an input for the processing of stream reproduction, instead it relies on provenance information (both static and dynamic information) recorded during past stream execution.

To replay stream execution, our replay method takes a set of provenance assertions representing individual stream events for a specific point of time as an input and produces replay execution output which are stream processing results originally produced by a stream processing system. Similar to the provenance query, our replay execution method applies the idea of function composition. As described earlier, we assume that the processing flow of a stream processing system is represented as a set of interconnected nodes and stream events flow through a directed graph of stream processing operations. Therefore, we can derive a particular output stream event in the processing flow by composing stream operations involved in the production of that output element. The algorithm for the replay execution method is presented in Figures 4.5. Note that, because the algorithm is fairly complex and it uses a lot of supporting functions (utility functions), to concisely illustrate our algorithm, we only present some important utility functions for the replay execution algorithm. All utility functions used for the replay execution algorithm are presented in Appendix A.

Algorithm: Replay execution - $replayExec(opList1, opList2, streamLut, sidList, paList)$
Let EM be an event map of type $(STREAMID * 'a\ EVENT\ list)\ list$
Let $opList1$ be a look up table (list) for unary stream operations
Let $opList2$ be a look up table (list) for binary stream operations
Let $streamLut$ be a look up table (list) for input and output streams
Let $sidList$ be a list of targeted stream IDs
Let $paList$ be a list of provenance assertions (event keys) recorded

```
$(* fn : (int * ('a\ EVENT\ list -> 'a\ list\ EVENT\ list))\ list *$
 $(int * ('a\ EVENT\ list -> 'a\ EVENT\ list -> ''c\ list\ EVENT\ list))$
 $list * (STREAMID * int * InOut)\ list * STREAMID\ list * KEY\ list ->$
 $(STREAMID * 'a\ EVENT\ list)\ list -> (STREAMID * 'a\ EVENT\ list)\ list *)$

$fun\ replayExec(opList1, opList2, streamLut, sidList, paList)\ [\,] = [\,]$
$|\ replayExec(opList1, opList2, streamLut, sidList, paList)\ EM =$
    $composeOp\ opList1\ opList2\ streamLut\ sidList\ paList\ EM\ [\,]$


$(* fn : (int * ('a\ EVENT\ list -> 'a\ list\ EVENT\ list))\ list$
 $-> (int * ('a\ EVENT\ list -> 'a\ EVENT\ list -> ''c\ list\ EVENT\ list))\ list$
 $-> (STREAMID * int * InOut)\ list -> STREAMID\ list -> KEY\ list$
 $-> (STREAMID * 'a\ EVENT\ list)\ list -> (STREAMID * 'a\ EVENT\ list)\ list$
 $-> (STREAMID * 'a\ EVENT\ list)\ list*)$

$fun\ composeOp\ opList1\ opList2\ streamLut\ sidList\ paList\ [\,]\ R = remdupl\ R$
$|\ composeOp\ opList1\ opList2\ streamLut\ sidList\ paList\ ((elm\ as\ (sid, \_)) :: EM)\ R =$
$let$
  $val\ opID = getOpID\ sid\ streamLut\ I$
  $val\ sid_{in} = getSID\ opID\ streamLut\ I$
  $val\ sid_{out} = getSID\ opID\ streamLut\ O$
$in$
   $if\ member\ sid\ sidList\ then$
     $composeOp\ opList1\ opList2\ streamLut\ sidList\ paList\ EM\ ([elm]\ @\ R)$
   $else$
     $if\ containsKeys\ sid_{in}\ (elm :: EM)\ then$
       $let$
         $val\ EM' = executeOp\ opList1\ opList2\ opID\ sid_{in}\ sid_{out}\ paList\ (elm :: EM);$
       $in$
         $composeOp\ opList1\ opList2\ streamLut\ sidList\ paList\ EM'\ R$
       $end$
     $else$
         $composeOp\ opList1\ opList2\ streamLut\ sidList\ paList\ (EM\ @\ [elm])\ R$
$end$
```

FIGURE 4.5: Algorithm for the replay execution method

Utility functions for the composeOp function

```
$datatype\ InOut\ =\ I \mid O$

$(*\ fn\ :\ ''a -> (''a * int * ''b)\ list -> ''b -> int\ *)$
$fun\ getOpID\ \ sid_{in}\ \ ((sid, opid, io) :: streamLut)\ \ io_{in}\ =$
    $if\ \ sid_{in}\ =\ sid\ \ andalso\ \ io_{in}\ =\ io\ \ then\ \ opid$
    $else\ \ getOpID\ \ sid_{in}\ \ streamLut\ \ io_{in};$

$(*\ fn\ :\ ''a -> ('b * ''a * ''c)\ list -> ''c -> 'b\ list\ *)$
$fun\ getSID\ \ opID_{in}\ \ [\,]\ \ io_{in}\ =\ [\,]$
$\mid getSID\ \ opID_{in}\ \ ((sid, opid, io) :: streamLut)\ \ io_{in}\ =$
    $if\ \ opID_{in}\ =\ opid\ \ andalso\ \ io_{in}\ =\ io\ \ then$
        $sid :: (getSID\ \ opID_{in}\ \ streamLut\ \ io_{in})$
    $else$
        $getSID\ \ opID_{in}\ \ streamLut\ \ io_{in};$

$(*\ fn\ :\ ''a\ list -> (''a * 'b)\ list -> bool\ *)$
$fun\ containsKeys\ \ [\,]\ \ HM\ =\ true$
$\mid containsKeys\ \ (k :: KL)\ \ HM\ =$
    $if\ \ containsKeys_1\ \ k\ \ HM\ \ then$
        $containsKeys\ \ KL\ \ HM$
    $else\ \ false$


$(*\ fn\ :\ ''a -> (''a * 'b)\ list -> bool\ *)$
$fun\ containsKeys_1\ \ k\ \ [\,]\ =\ false$
$\mid containsKeys_1\ \ k\ \ ((k1, \_) :: HM)\ =$
        $(k\ =\ k1)\ \ orelse\ \ containsKeys_1\ \ k\ \ HM;$

$(*\ fn\ :\ (int * ('a\ EVENT\ list -> 'a\ list\ EVENT\ list))\ list$
 $-> (int * ('a\ EVENT\ list -> 'a\ EVENT\ list -> ''c\ list\ EVENT\ list))\ list$
 $-> int -> STREAMID\ list -> STREAMID\ list -> KEY\ list$
 $-> (STREAMID * 'a\ EVENT\ list)\ list$
 $-> (STREAMID * 'a\ EVENT\ list)\ list\ *)$
$fun\ executeOp\ \ opList1\ \ opList2\ \ opID\ \ sid_{in}\ \ sid_{out}\ \ paList\ \ EM\ =$
$let$
    $val\ ((\_, eList) :: IM)\ =\ getElements\ \ sid_{in}\ \ EM;$
    $val\ output\ =\ (executeOp_1\ \ opList1\ \ opList2\ \ opID\ \ sid_{out}\ \ paList\ \ eList\ \ IM)$
    $val\ EM'\ =\ removeElm\ \ sid_{in}\ \ EM$
$in$
    $EM'\ @\ remdupl\ \ output$
$end$
```

As shown in Figures 4.5, a map or an associative array ($EM\ :\ (STREAMID\ *\ ('a\ EVENT\ list)\ )\ list$), which consists of a stream ID as a unique key and a list of stream events as its associated value, is mainly used to store input stream events and intermediate results. The replay execution algorithm consists of two main functions: *replayExec* and *composeOp*. The *replayExec* function is the entry-point function that takes an input map containing input stream events (EM), look up tables for stream operations ($opList1, opList2$), a look up table for input and output streams ($streamLut$), a list of target stream IDs (stream IDs used to terminate the replay execution - $sidList$) and a list of provenance assertions recorded ($paList$) as input parameters. After taking all parameters, the function returns a result map (R) which contains output events produced by the replay execution process. Like the provenance query algorithm, the *composeOp* is the recursive function that contains the business logic of the replay execution method. The process of the function begins by receiving parameters passed by the *replayExec* function. Then it iterates over the elements of an input map (EM) in order to execute stream operations on every input stream and every input event. For each input stream all parameters required for the processing of its associated stream operation are collected (the algorithm identifies each associated stream operation by using $getOpID$ function). Then a stream ID of each input stream needs to be checked (using *member* function). This is for collecting the final results of the replay execution. After that, the $containsKeys$ function is used to check whether all input streams required for the processing of the stream operation are available (in the case of binary operation e.g. time-window join). If the stream ID is not the IDs in the stream ID list (targeted stream - $sidList$) and all required input streams are available, all events of the input streams will be processed by its associated stream operation (using *executeOp* function). Every output stream returned by the stream operation is inserted back to the input map EM. Finally, if there are input streams in the input map EM (which means there are some intermediate streams still waiting to be processed), the *composeOp* function will be called recursively until no intermediate streams remain in the input map EM.

4.3 A case study for provenance queries

To demonstrate that our provenance solution - the stream ancestor functions and the provenance query algorithm - is expressive enough for precisely tracking provenance information at the level of individual stream events, we use a simple synthetic processing flow of a stream system as an example. A provenance query constructed by composing stream ancestor functions is presented to capture the provenance of the synthetic processing flow. The outputs of the provenance query are used as an input for our replay execution method in order to illustrate how the replay execution method works and how to validate query results that are produced by the provenance query. Note that, the example Standard ML code for this case study is also provided in Appendix B.

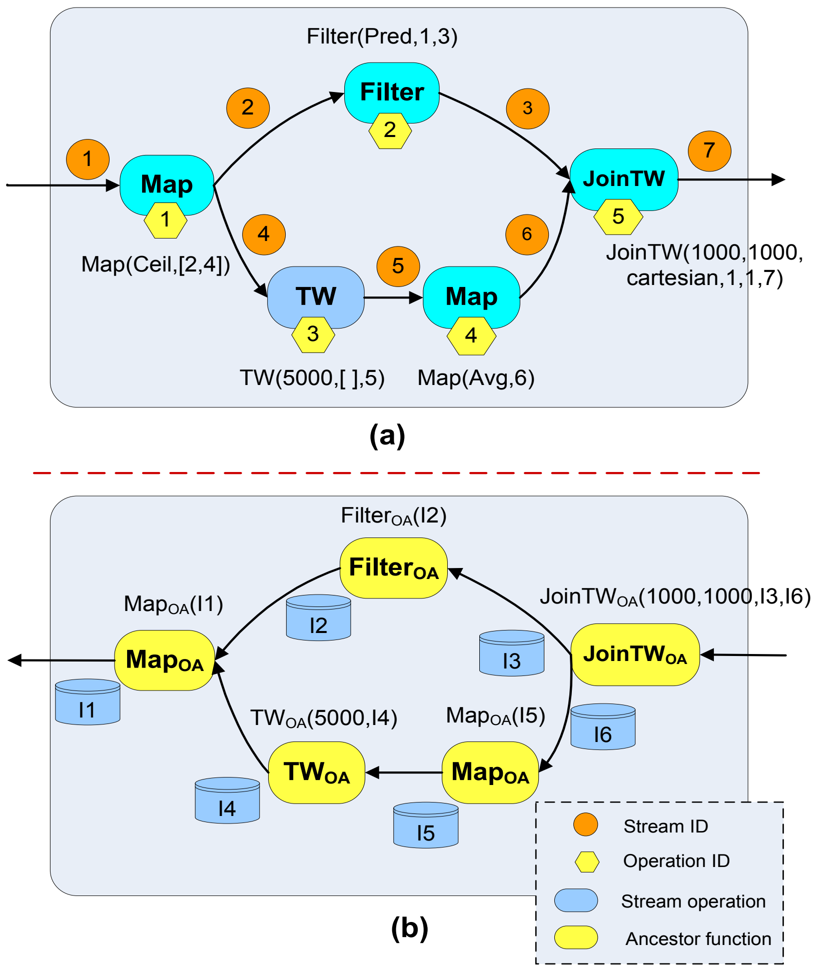


FIGURE 4.6: The processing flow of a stream processing system (a) vs the composition of stream ancestor functions (b)

The synthetic processing flow of our stream processing system is presented in Figure 4.6 (a). The processing flow is constructed by composing stream operations including *Map*, *Filter*, *Sliding time window (TW)* and *Time window join (JoinTW)*. For each operation input and output streams are labeled with unique IDs (stream identifiers). For example, the streams 4 and 5 are the input and output streams of the sliding time window operation (TW). In addition each stream operation is assigned a unique operation ID. For example, the filter operation is assigned no.2 as its operation ID.

The execution of our synthetic processing flow can be explained as follows: the processing flow begins with input events containing temperature values fed into a stream processing system at a rate of one event every two seconds. The first operation, map operation (*Map(Ceil,[2,4])*), applies the ceil function which returns the smallest following integer to the content of every input element. The events are then multicast to two stream operations. The filter operation (*Filter(P,1,3)*) is responsible for filtering out events that do not meet the filter condition (the condition of temperature greater than 60 degrees Fahrenheit). The time window operation (*TW(5000,[],5)*) is used to generate each output event from the most recent events over 5 seconds (5000 milliseconds). Output events generated by the time window operation are then passed to another map operation which applies the average function over the content of every input event. Finally, the events from the filter and the map operation are joined by the time-window join operation (*JoinTW(1000,1000,cartesian,1,1,7)*) and the final outputs are generated based on the most recent events over a second. Example execution of the case study's synthetic processing flow is illustrated in Figure 4.7. This example execution is based on an implementation of our stream provenance system with Esper stream engine described in Section 6.1 (the implementation design).

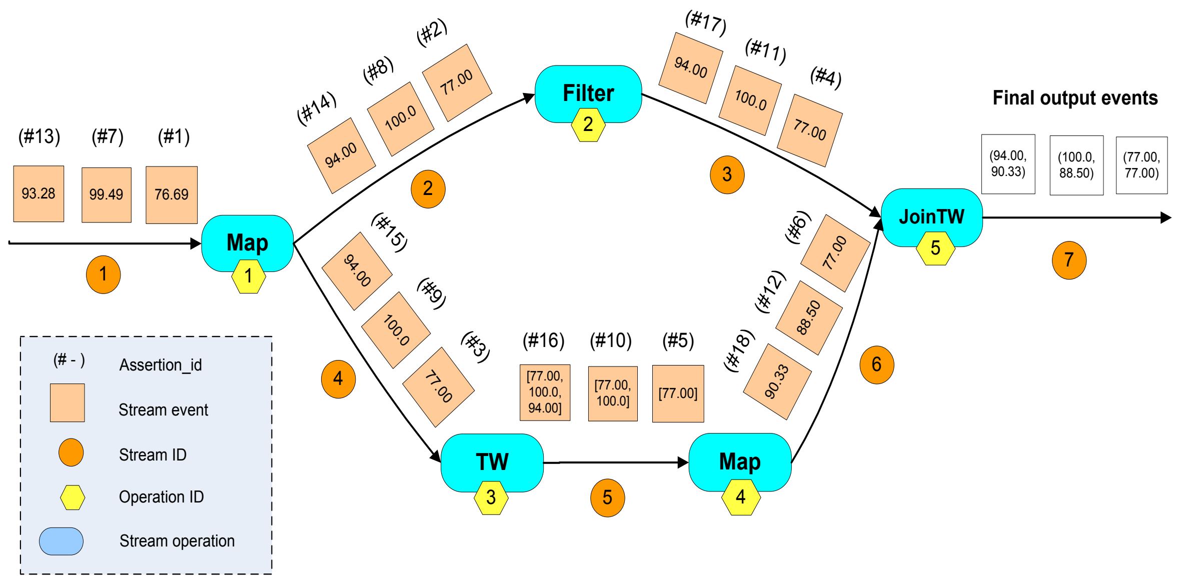


FIGURE 4.7: Execution of the case study's synthetic processing flow

assertion_id	timestamp	seqno	stream_id	delay	event_content	assertor	content discarded
1	1279398105675	1	1	0	76.69	1	
2	1279398105684	1	2	9		3	77.00
3	1279398105684	1	4	9		2	77.00
4	1279398105693	1	3	9		5	77.00
5	1279398105701	1	5	17		4	[77.00]
6	1279398105702	1	6	1		5	77.00
7	1279398107678	2	1	0	99.49	1	
8	1279398107678	2	2	0		3	100.00
9	1279398107678	2	4	0		2	100.00
10	1279398107680	2	5	2		4	[77.00,100.00]
11	1279398107681	2	3	3		5	100.00
12	1279398107682	2	6	2		5	88.50
13	1279398109678	3	1	0	93.28	1	
14	1279398109678	3	2	0		3	94.00
15	1279398109678	3	4	0		2	94.00
16	1279398109680	3	5	2		4	[77.00,100.00,94.00]
17	1279398109681	3	3	3		5	94.00
18	1279398109682	3	6	2		5	90.33

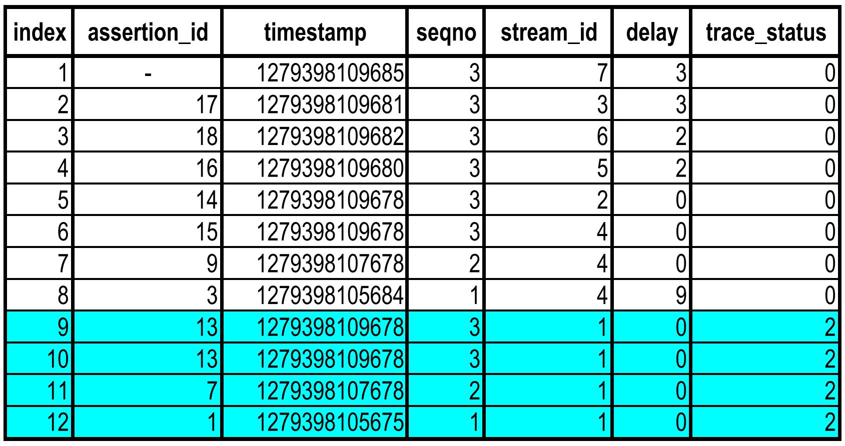
FIGURE 4.8: The assertions table

As illustrated in Figure 4.7, in this example, three input stream events are fed into the stream processing system. Figure 4.8 presents the *assertions* table related to the example execution of the synthetic processing flow. As described in the previous chapter (Section 3.1.3), this assertions table is used to store a set of provenance assertions recorded during the execution of the synthetic processing flow. The assertions table consists of the following fields: an assertion identifier (assertion_id), a set of fields representing an event key of a stream element (timestamp, seqno, stream_id and delay), event_content and an assertor (operation ID of a stream operation that records provenance assertions). In addition, as with the concept of our fine-grained provenance model for streams, only contents of the first input stream, stream 1, are recorded. Note that, for better understanding, each record presented in the assertions table can refer to an individual stream element in Figure 4.7 by using assertion_id. We also provide an extra field for event content that is discarded during the generation of provenance assertions.

To obtain the provenance of each stream event in the processing flow, stream ancestor functions are composed as illustrated in Figure 4.6 (b). The stream ancestor functions composed include Map_{OA}, $Filter_{OA}$, TW_{OA} and $JoinTW_{OA}$. All stream ancestor functions used in this case study are the optimized version of stream ancestor functions. Furthermore, the provenance assertions of all intermediate streams are required to be stored in a provenance store. For example, the stream ancestor function, $Filter_{OA}$, requires the provenance assertions of the input stream 2 to be stored in order to perform provenance queries. By passing the provenance assertions of output events from the output stream 7 to the composed stream ancestor functions, a set of provenance assertions of source events (ancestor events) from stream 1 can be identified.

We now demonstrate how to perform a fine-grained provenance query in order to exactly trace individual stream events. The fine-grained provenance query algorithm described in section 4.1.2 is applied in order to compose all stream ancestor functions dynamically. In Figure 4.9, the trace table, the internal table used by the provenance query algorithm for processing the provenance query, is presented. The trace table is used to temporarily store both query results and intermediate results produced during the execution of the provenance query. In the trace table, the top row (index:#1) represents the event key of the provenance assertion which is the input of the provenance query, and the last four rows (index:#9-#12) represent the query results.

In Figure 4.10, a provenance graph related to the trace table is presented. This provenance graph is used only for presentation clarity. It demonstrates how intermediate results produced during the execution of the provenance query (stored in the trace table) can be used to constitute a directed acyclic graph (DAG) - a core element of provenance representation. The provenance graph is also used to describe dependency relationships between each intermediate query result. In the provenance graph, each node labeled with an index number represents individual records in the trace table. Because the records #9 and #10 are the same provenance assertion, they are represented with a shared node. By traversing the provenance graph in reverse, we can exactly identify that the events from stream 1 (index:#9-#12) are the source events used in the production of the output event from stream 7 (index:#1).



index	assertion_id	timestamp	seqno	stream_id	delay	trace_status
1	-	1279398109685	3	7	3	0
2	17	1279398109681	3	3	3	0
3	18	1279398109682	3	6	2	0
4	16	1279398109680	3	5	2	0
5	14	1279398109678	3	2	0	0
6	15	1279398109678	3	4	0	0
7	9	1279398107678	2	4	0	0
8	3	1279398105684	1	4	9	0
9	13	1279398109678	3	1	0	2
10	13	1279398109678	3	1	0	2
11	7	1279398107678	2	1	0	2
12	1	1279398105675	1	1	0	2

FIGURE 4.9: The trace table

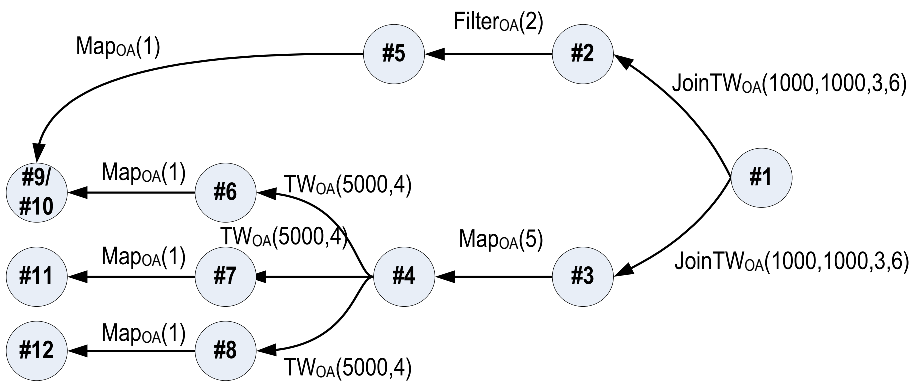
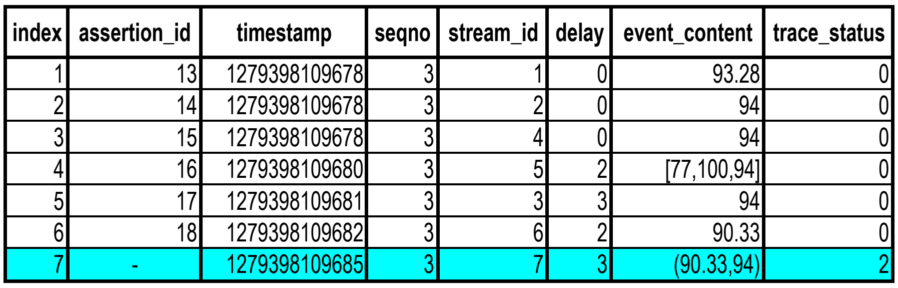


FIGURE 4.10: The provenance graph related to the trace table

To demonstrate stream reproduction and validate provenance query results, we present an example of replay execution. We apply the replay execution algorithm described in Section 4.2 in order to compose all stream operations dynamically. Figure 4.11 shows the replay table which is the internal table used by the replay execution algorithm for processing the stream replay execution. In the replay table the top row (index:#1) represents the provenance assertion which is the input of the replay execution, and the last row (index:#7) represents the replay result. We also present a processing graph related to the replay table in Figure 4.12 to describe the dependencies between each intermediate replay result. Similar to the provenance graph, each node labeled with an index number represents an individual record in the replay table. We start validating the query result produced by the previous provenance query by passing the query result (index:#1) to the replay execution algorithm. The intermediate results produced during the processing of the replay execution algorithm are stored in the replay table. Finally, we can derive the output event (index:#7). By comparing the output event derived from this replay execution and the event input to the previous query, we can demonstrate the precision of our provenance query solution and also we can validate the results produced by our fine-grained provenance query.



index	assertion_id	timestamp	seqno	stream_id	delay	event_content	trace_status
1	13	1279398109678	3	1	0	93.28	0
2	14	1279398109678	3	2	0	94	0
3	15	1279398109678	3	4	0	94	0
4	16	1279398109680	3	5	2	[77,100,94]	0
5	17	1279398109681	3	3	3	94	0
6	18	1279398109682	3	6	2	90.33	0
7	-	1279398109685	3	7	3	(90.33,94)	2

Figure 4.11: The replay table

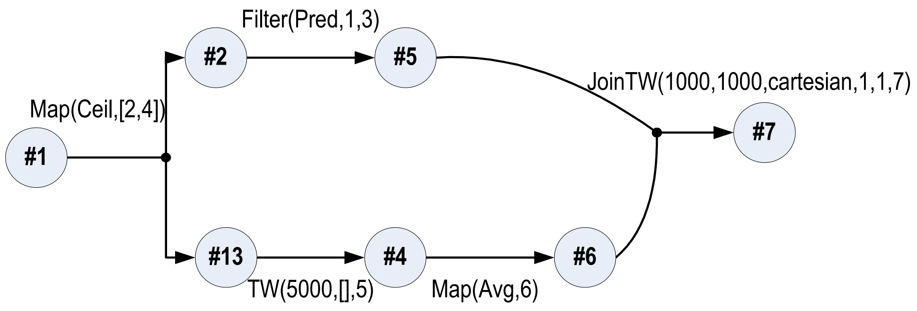


Figure 4.12: The processing graph related to the replay table

4.4 Summary

Once provenance assertions created by stream operations in a stream processing system are stored in a central provenance repository - provenance store, a number of provenance queries can be performed in order to determine the provenance of stream processing results. In this chapter we introduced the stream provenance query solution that resolves dependency relationships among stream elements and provides the complete provenance of individual stream elements (fine-grained provenance query). We provided the formal description of the provenance query mechanism for streams through the generic provenance query algorithm. In order to validate the results produced by the provenance query mechanism and demonstrate the precision of this query mechanism, we presented the replay execution method. Our replay execution method utilizes provenance assertions and auxiliary information (including topology configuration parameters and stream operations parameters) stored in a provenance store to derive an output stream element.

We now revisit the main contributions of this chapter: Firstly, a stream provenance query mechanism , and secondly, a replay execution method. To support the first contribution we presented the idea of function composition and detailed how this can be applied to our primitive stream operations and their stream ancestor functions. Based on the concept of function composition, we identified two key practical requirements that our provenance query mechanism needs to address: dynamic composition of stream ancestor functions and scoped provenance queries. By addressing these requirements we can offer a generic and topology-independent query solution which supports complex stream manipulation operations and can be used to obtain the provenance of individual stream element in a variety of stream processing flows. To support the second contribution, we proposed the replay execution algorithm. We defined the core function of replay execution method that contains business logic and also some necessary utility functions used to support the core function. Furthermore, we demonstrated how the provenance query works in practice and how the actual results generated by the provenance query are validated through the use of our replay execution method by presenting an example case study for provenance queries. Using this simple case study we can clarify how our provenance query and replay execution method work step-by-step. In addition, the impact of all the provenance solutions (presented in this chapter) on stream processing system performance will be evaluated in Chapter 6.

Although we have proposed the provenance query solution for streams that can perform provenance tracking at the level of individual stream elements (fine-grained stream provenance queries), there are some practical challenges related to the unique characteristics of data streams that are required to be addressed. One of the most significant practical challenges is the excessive storage requirement resulting from the persistence of provenance information for all intermediate stream elements. In the next chapter, a stream-specific provenance query mechanism that can perform provenance queries on-the-fly over streams of provenance assertions will be presented. This query mechanism exploits our stream ancestor functions and composes them together on-the-fly without requiring provenance assertions to be stored persistently. With the combination of both provenance query solutions we can properly tackle the problem of fine-grained provenance tracking in stream processing systems.

Chapter 5

Stream-specific provenance query

In the previous two chapters, although we have proposed a provenance model and provenance query solution for streams that can be used to track the provenance of individual stream elements, there are a number of practical challenges related to the unique characteristics of data streams that we still need to address. One of the most significant requirements that stream processing systems need to satisfy is that they must be able to process stream events on-the-fly without any requirement to store them. To address such a practical challenge we introduce a novel stream-specific provenance query mechanism in this chapter. The contribution of this chapter is a stream-specific provenance query mechanism that enables a collection of provenance queries to be performed on-the-fly over streams of provenance assertions without requiring the provenance assertions to be stored persistently in a provenance store. This mechanism is based on the idea of using the provenance service as a stream component. We also identify the important characteristics of this stream-specific provenance query mechanism. In addition, the specifications of the stream ancestor functions that are designed specifically to work with an on-the-fly provenance query mechanism are also defined.

The rest of the chapter is organized as follows. First, the fundamental concept of our stream-specific provenance query solution is detailed. After the stream-specific provenance query solution is conceptually introduced, the specifications of the new version of the stream ancestor functions designed for stream-specific provenance queries are described. This is followed by the presentation of an on-the-fly provenance query algorithm. Finally, an example case study for on-the-fly provenance queries is demonstrated and the chapter is concluded.

5.1 On-the-fly provenance query mechanism

We now present the fundamental concept of an on-the-fly provenance query mechanism that is the complementary solution for our fine-grained provenance tracking in stream processing systems. The purpose of the introduction of on-the-fly provenance queries is to present a novel stream-specific query solution that can address the practical requirement pertaining to "Keep the streaming data moving" rule [120] or more particularly to process incoming stream elements without any requirement to store them in a persistent storage. This mechanism is not intended to replace or supersede the existing "persistent provenance" mechanism that we presented in Chapters 3 and 4 because the persistent provenance mechanism offers several advantages.

- The persistent provenance mechanism allows us to track the provenance of individual stream elements and perform provenance queries at whatever time we need. Because provenance assertions are stored persistently in a provenance store which possesses immutable characteristics (it guarantees that provenance assertions that have been previously recorded will not be overwritten, deleted or modified), we can perform provenance queries multiple times after provenance assertions are recorded.
- With the persistence of provenance assertions, we can utilize the provenance assertions together with auxiliary information (topology information and configuration parameters) stored in a provenance store in order to reproduce or replay stream events if required. The reproduction capability enables the output of stream processing systems to be validated and processing steps can be checked and verified.

With the combination of persistent provenance query and on-the-fly provenance query mechanisms, we can offer all-round provenance query solution that can properly address the problem of fine-grained provenance tracking in stream processing systems.

5.1.1 Basic assumptions for on-the-fly provenance query

Before outlining the concept of on-the-fly provenance query in detail, the following is the basic assumption for our on-the-fly provenance query mechanism.

- The transmission order of provenance assertions streamed to a provenance service must be preserved. This order-preserving property allows for our stream-specific query mechanism to operate both time-based and order-based operations over streams of provenance assertions. It also allows us to control the size of data buffer used for the processing of on-the-fly provenance queries.

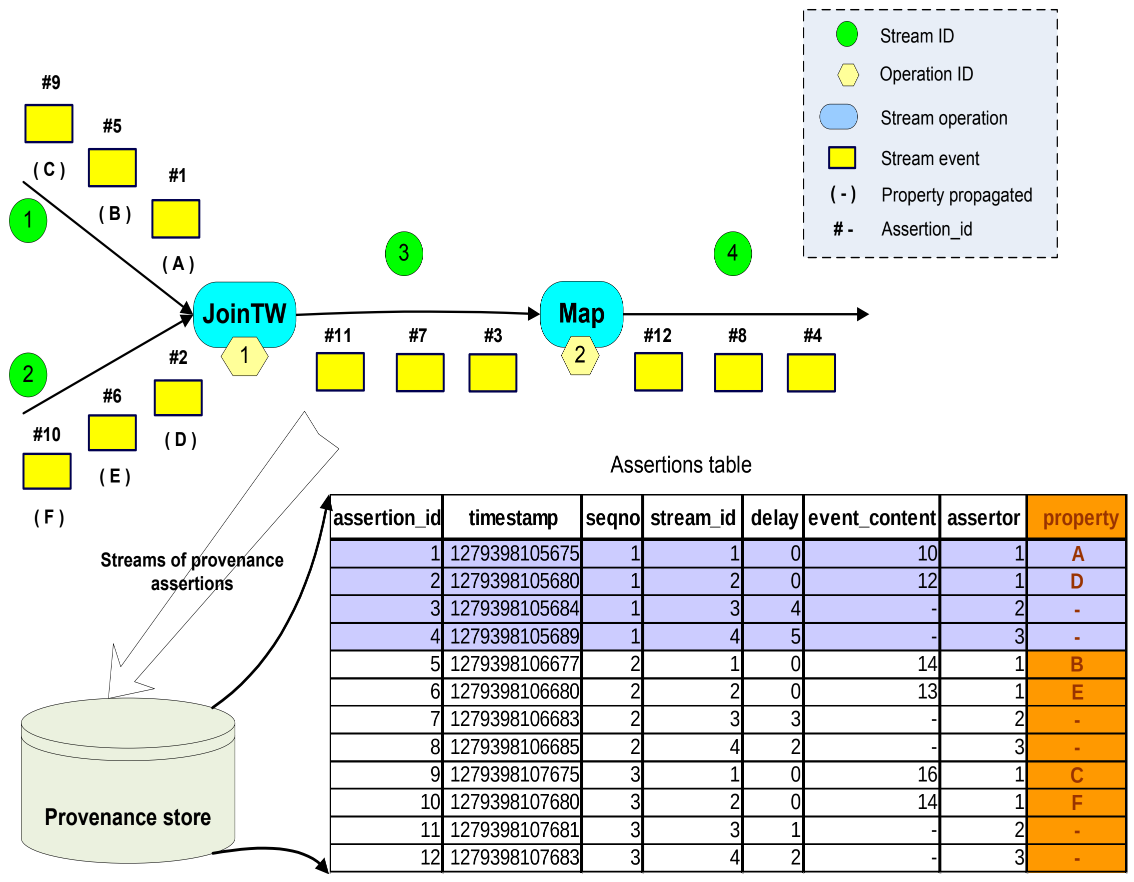
- Each individual stream element is associated with one or more properties. A property in our context is a piece of information describing a quality, characteristic or attribute that belongs to an individual stream element. Properties can range from a simple property (e.g. a single integer value) to a complex property (e.g. a tree, a graph or a collection of data items). For example, in a stream processing system that receives measurements from environmental sensors, a property associated with each stream element (raw sensor measurement) could be a sensor identifier or a sensor location (map coordinate).

- Static provenance information (e.g. stream topology and stream operation parameters) is only the part of our provenance model required to be stored in a provenance store (a compact provenance store demonstrated in Figure 3.5) for supporting on-the-fly provenance query processing. No persistent storage space is required for dynamic provenance information and provenance-related properties due to the fact that properties are computed on-the-fly over streams of provenance assertions (dynamic provenance information) without any requirement to store them persistently (we will discuss how on-the-fly provenance query works in detail in the next section).

- Storage requirements on properties (the amount of temporary memory space) depend on types of provenance-related properties utilized for on-the-fly query processing. For example, if a property used is a sensor id which is represented by an integer value, the storage requirement is only 4 bytes per property. On the other hand, if a sequence of processing steps (processing paths) used to generate an output event is utilized as a property, because a stream id for each processing step (represented by an integer value) is required to be accumulated to obtain the final result, so the storage requirement for this property can be calculated as the storage size for each stream id (4 bytes) multiplied by the number of processing steps used. In addition, in the case that a large number of properties is required to be temporarily stored for processing, the idea of using a unique identifier as a reference to a collection of properties for each individual stream element can be applied to reduce the amount of storage space required to temporarily store a large number of properties during the processing of on-the-fly provenance queries.

5.1.2 Fundamental concept of on-the-fly provenance query

Our on-the-fly provenance query mechanism adopts concepts from the persistent provenance query mechanism presented in the previous chapter. We apply the idea of function composition in order to utilize a stream ancestor function defined for each stream operation. All stream ancestor functions of stream operations in a stream processing system are dynamically composed and then their execution is interleaved continuously with the execution of the stream processing system. We extend the concept of the persistent provenance mechanism by adding stream-specific techniques designed specifically for addressing on-the-fly stream provenance queries. Our key concept is inspired by the idea of property propagation or annotation propagation. Based on the assumption that each individual stream element is associated with one or more properties, we can obtain the provenance of each individual output element by propagating provenance-related properties from the originating source (the stream elements of the first input streams) to the final destination (the stream element of the final output stream) of a stream processing system.



assertion_id	timestamp	seqno	stream_id	delay	event_content	assertor	property
1	1279398105675	1	1	0	10	1	A
2	1279398105680	1	2	0	12	1	D
3	1279398105684	1	3	4	-	2	-
4	1279398105689	1	4	5	-	3	-
5	1279398106677	2	1	0	14	1	B
6	1279398106680	2	2	0	13	1	E
7	1279398106683	2	3	3	-	2	-
8	1279398106685	2	4	2	-	3	-
9	1279398107675	3	1	0	16	1	C
10	1279398107680	3	2	0	14	1	F
11	1279398107681	3	3	1	-	2	-
12	1279398107683	3	4	2	-	3	-

FIGURE 5.1: An example of provenance-related property computed by the persistent provenance mechanism

To clearly describe the idea of property propagation, we first present an example of how we can compute provenance-related properties by using our existing persistent provenance mechanism, as illustrated in Figure 5.1. Suppose that in a stream system that receives measurements from environmental sensors, one of properties associated with each input stream event is the coverage area of each sensor (area A, B, C and so on). To determine what the coverage areas of sensors used as sources of the input events involved in the production of a particular output event are (or to capture provenance-related properties) in the persistent provenance mechanism, all provenance assertions for individual stream events together with their associated properties need to be stored persistently in a provenance store (using assertions table). An additional field (*property*) designed specifically for storing property information is required to be created. As in the example shown in Figure 5.1, to capture provenance-related properties of the output stream event (assertion #4), we need to perform a provenance query to obtain the provenance trail of that output event which consists of the provenance assertions of stream events id #3,#2 and #1, and finally the coverage area A and D (properties) from the assertions #2 and #1 can be identified as the query answer. As shown in this example, due to the fact that several stream processing systems do not have mass storage, property computing using the persistent provenance mechanism, that requires the persistence of all provenance assertions with an extra field for property computing, potentially causes a storage burden problem.

To overcome such a storage problem, we introduce an alternative approach where properties are computed on-the-fly, without any requirement to store provenance assertions persistently, by using accumulators. In this approach, we assume that each individual stream element contains a provenance-related property in its accumulator - a field used to accumulate property computing results. For each stream operation, provenance-related properties are computed and propagated automatically from input streams (ancestor streams) to output streams (descendant streams) based on input-output dependency relationships between stream elements. Each intermediate result produced by the processing of provenance-related properties is accumulated and temporarily stored in the accumulator field of intermediate stream elements. The processing of property propagation is performed continuously until reaching the final stream operation of a stream processing flow. Figure 5.2 illustrates an example of our property propagation approach in which properties are propagated through a stream processing flow.

In the example (illustrated in Figure 5.2), provenance-related properties are propagated from the stream elements of the first input streams (stream 1 and stream 2) to the stream element of the final output stream (stream 4). As shown in this example, there is no storage space required for recording provenance assertions because we compute provenance-related properties on-the-fly and accumulate them for further processing in accumulators. By using this property propagation approach, we can derive the same provenance-related properties as the persistent provenance mechanism for the output stream event (assertion #4) which are the coverage area A and D. However, it is important to note that we do not try to propagate provenance-related properties within a stream processing system layer because this would require the modification of the internal processing of stream processing operations, but instead provenance-related properties are computed and propagated through the use of provenance assertions inside a provenance service.

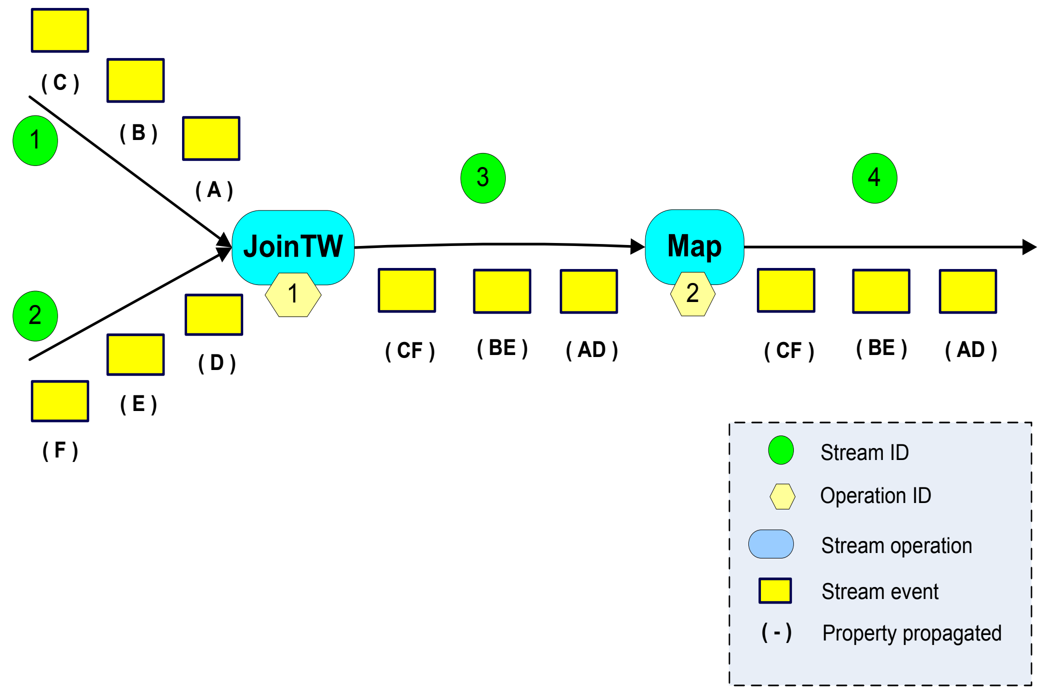


FIGURE 5.2: An example of property propagation in a stream processing system

The details of the on-the-fly provenance query mechanism are illustrated in Figure 5.3. The figure shows the internal processing of the on-the-fly provenance query which describes the dynamic execution of the on-the-fly provenance query and how the provenance query mechanism works inside the provenance service. Note that we use the same stream processing flow (stream topology) presented in the property propagation example (Figure 5.2).

Our on-the-fly provenance query mechanism is based on the idea of utilizing the provenance service as a stream component. The execution begins with a provenance service receiving streams of provenance assertions (AS) generated by stream operations in a stream processing system. Each provenance assertion is detected by an assertion separation unit. The assertion separation unit (ASU) is a component used to detect provenance assertions for a particular stream, create an extra field for storing property information (a property field used as a property accumulator) and direct the detected provenance assertions to a stream ancestor function that they are associated with. In the case that arrival assertions are provenance assertions of the first-input event - stream events that enter a stream processing system from data sources and are first processed by stream processing operations, it will be sent to a property computing unit (PCU) first in order to extract provenance-related properties. After that each individual provenance assertion is computed by its associated stream ancestor function. The stream ancestor function (SAF) used in our on-the-fly provenance query is the new version of the original stream ancestor function - called a property stream ancestor function (PSAF). It receives stream elements from both an assertion stream (AS) and result streams (RS)

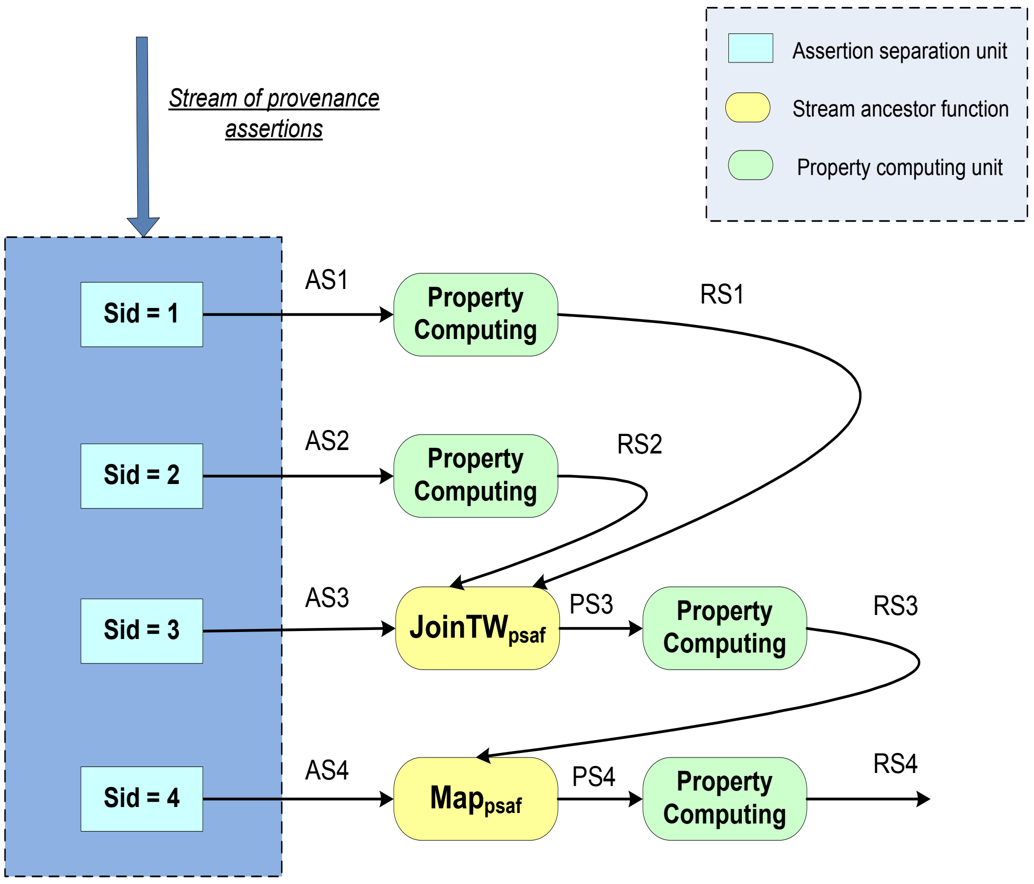


FIGURE 5.3: An internal process of on-the-fly provenance query mechanism

- the output from the previous step of property propagation - as an input and then it produces an output element belonging to a property stream (PS). Not only is the stream ancestor function for on-the-fly provenance queries used to identify the ancestors of a particular provenance assertion, it is also used to extract provenance-related properties from the ancestor provenance assertions (elements from RS). The output generated from the stream ancestor function is fed into a property computing unit in order to compute property propagation. Once the property propagation is processed, an output provenance assertion containing a new property is produced as an element of a result stream (RS). This process of on-the-fly provenance queries is executed continuously until it reaches the final stream ancestor function that executes provenance assertions of the final output stream. The process of how provenance-related properties are propagated inside a provenance service is demonstrated in Figure 5.4

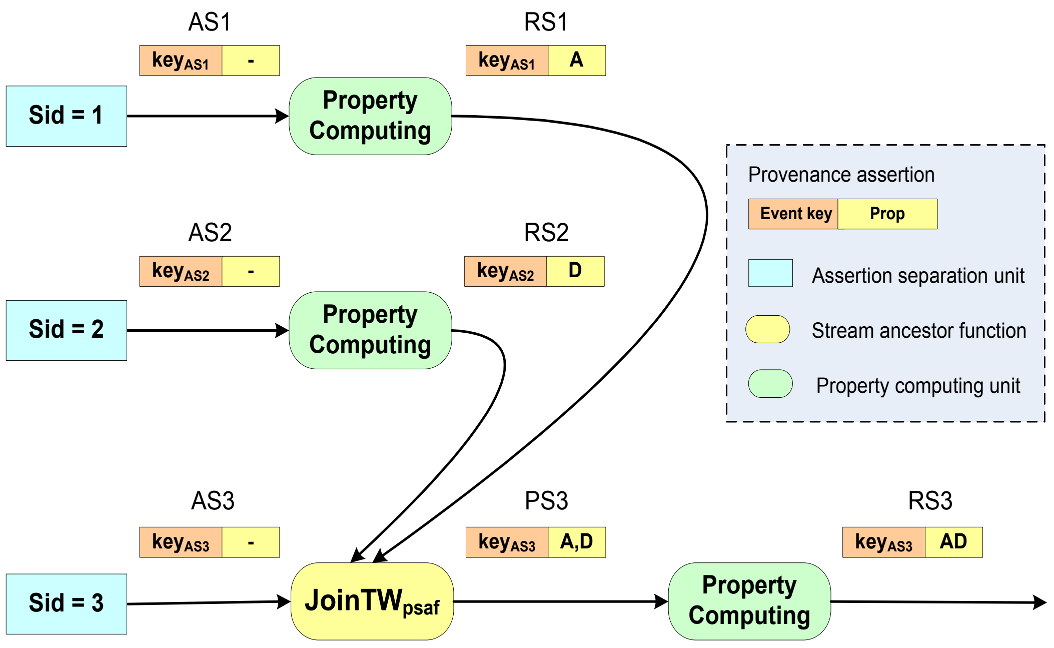


FIGURE 5.4: An example of how properties are propagated inside a provenance service

As illustrated in Figure 5.4, an extra property field (*prop*) are added by an assertion separation unit when a provenance assertion for each stream is detected. It is used as an accumulator for supporting property computing. Then the stream ancestor function (SAF) for the time-window join operation ($JoinTW_{psaf}$) is used to extract the properties (A and D) from the ancestor provenance assertions (elements in RS1 and RS2) and to put them into "prop" field. After that the properties identified by the SAF and the property of the current provenance assertion (AS3) are computed by the property computing unit. Finally, a new property value is replaced in "prop" field and the output assertion (RS3) is generated. It is necessary to note that we separate a property computing unit from a stream ancestor function component because we would like to make the SAF a generic function that can be used for any stream-based application independent from property calculating logic.

5.2 Property stream ancestor functions

In this section, the specifications of stream ancestor functions used for working with on-the-fly provenance queries are presented. We call this version of stream ancestor functions "Property stream ancestor function" (PSAF) because one of the most significant tasks for these stream ancestor functions is to extract provenance-related properties from ancestor provenance assertions. We will detail data types and shared functions first and then we will present the specifications of property stream ancestor functions.

5.2.1 Data types for representing a provenance assertion

As presented in the previous section, a provenance assertion used for an on-the-fly provenance query consists of an event key and an extra property field - prop (a list of properties). An event key contains a timestamp ($TIME$), a sequence number ($LargeInt$), a stream identifier ($STREAMID$) and a delay time for processing($TIME$). We represent a stream of provenance assertions as a list of assertions ($'a\ ASSERTION\ list$) where a stream can contain a varying number of elements and properties can be any type of content. Streams of provenance assertions are utilized in our on-the-fly provenance query mechanism as internal streams involved in the processing of on-the-fly provenance queries. The following are data types for representing a provenance assertion.

$$datatype\ KEY\ =\ Key\ of\ TIME\ *\ LargeInt.int\ *\ STREAMID\ *\ TIME;$$
$$datatype\ 'a\ ASSERTION\ =\ Assertion\ of\ KEY\ *\ 'a\ list;$$

The following is an example of a stream of provenance assertions represented by using a list of assertions.

$$[Assertion\ (Key\ (Time\ 11, 1, StreamID\ 1, Time\ 1), [9]),$$
$$Assertion\ (Key\ (Time\ 21, 2, StreamID\ 1, Time\ 2), [7]),$$
$$Assertion\ (Key\ (Time\ 31, 3, StreamID\ 1, Time\ 1), [10])] :\ int\ ASSERTION\ list$$

5.2.2 Shared functions

We now present shared functions defined for working with property stream ancestor functions. These shared functions include $ExtractPropT$ and $ExtractPropN$. $ExtractPropT$ and $ExtractPropN$ are functions used to extract properties from elements in an assertion list (Q) that are inside the scope of a data window (the scope ranges from lower bound (lb) to upper bound(ub)). The difference between these two functions is that elements of a list are scoped by using a timestamp (t) for one and a sequence number (n) for the other. The definitions of these functions are presented as follows.

```
(* $fn\ :\ TIME\ *\ TIME\ *\ 'a\ ASSERTION\ list\ ->\ 'a\ list$ *)

$fun\ ExtractPropT(ub, lb, [\,])\ =\ [\,]$
$|\ ExtractPropT(ub, lb, ((Assertion(Key(t, \_, \_, \_), p)) :: Q))$
              $=\ if\ (ub\ GTE\ t)\ andalso\ (t\ GTE\ lb)\ then$
                  $p\ @\ ExtractPropT(ub, lb, Q)$
              $else\ ExtractPropT(ub, lb, Q);$
```

```
(* fn : int * int * 'a ASSERTION list -> 'a list *)

fun ExtractPropN(ub, lb, [ ]) = [ ]
| ExtractPropN(ub, lb, ((Assertion(Key(_, n, _, _), p)) :: Q))
          = if (ub >= n) andalso (n >= lb) then
              p @ ExtractPropN(ub, lb, Q)
          else ExtractPropN(ub, lb, Q);
```

5.2.3 Abstract functions for property stream ancestor functions

Abstract functions are defined to separate common routines contained in several property stream ancestor functions (PSAFs) and provide base functions that a number of property stream ancestor functions can exploit. The benefit of defining abstract functions is that we can reduce code repetition in several property stream ancestor functions, the definitions of property stream ancestor functions are simplified and they can be easily understood. These abstract functions include $AbstractTW_{psaf}$ and $AbstractLW_{psaf}$.

$AbstractTW_{psaf}$ is the abstract function defined for property stream ancestor functions for a sliding time window, a map operation and a filter operation. The function takes a provenance assertion (AS) and a result stream (RS) - a stream generated from the previous property propagation step - as an input and generates an element of a property stream (PS) containing all provenance-related properties needed for further processing. This function extracts properties from RS by creating the extent of time window at the time that the operation produced the output using the duration of time window (w). The definition of the $AbstractTW_{psaf}$ function is described in Figure 5.5.

```
(* fn : TIME -> 'a ASSERTION list -> 'b ASSERTION
 -> 'a ASSERTION *)

fun AbstractTW_psaf(w) Q (Assertion(Key(t, n, sid, d), _)) =
let
    val pList = ExtractPropT(t - -d, t - -d - -w, Q);
in
   Assertion(Key(t, n, sid, d), pList)

end
```

FIGURE 5.5: The definition of $AbstractTW_{psaf}$ function

The definition of the other abstract function - $AbstractLW_{psaf}$ - is described in Figure 5.6. The $AbstractLW_{psaf}$ function is the abstract function defined for the property ancestor function for a length window. It utilizes a size of length window (l) and a queue (Q) as the function parameters. Similar to the $AbstractTW_{psaf}$ function, it takes a provenance assertion (AS) and a result stream (RS) as an input and returns a property stream (PS) containing all properties needed for further processing. To extract properties from RS, instead of using timestamps as variables, this function uses the size of length window (l) and a sequence number (n) for creating the extent of count-based window at the time that the operation produced the output.

```
(* fn : int -> 'a ASSERTION list -> 'b ASSERTION
-> 'a ASSERTION *)

fun AbstractLW_psaf(l) Q (Assertion(Key(t, n, sid, d), _)) =
let
   val pList = ExtractPropN(n, n - l + 1, Q);
in
   Assertion(Key(t, n, sid, d), pList)
end
```

FIGURE 5.6: The definition of $AbstractLW_{psaf}$ function

5.2.4 The property stream ancestor function for a map operation

Property stream ancestor function: PSAF for a map operation - Map_{psaf}
Let RS be a result stream of type $'a\ ASSERTION\ list$
Let AS be a provenance assertion of type $'b\ ASSERTION$
For a map operation, the property stream ancestor function is defined as:

```
(* fn : 'a ASSERTION list -> 'b ASSERTION -> 'a ASSERTION *)

fun Map_psaf RS AS = AbstractTW_psaf(Time(0)) RS AS;
```

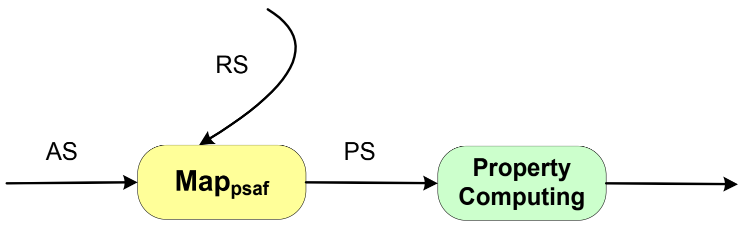


FIGURE 5.7: The definition of PSAF for a map operation

The property stream ancestor function for a map operation is defined in Figure 5.7. The function takes a provenance assertion (AS) and a result stream (RS) - a stream generated from the previous property propagation step - as an input and returns an element of a property stream (PS) containing all provenance-related properties needed for further processing as the function output. In the definition, this PSAF utilizes the $AbstractTW_{psaf}$ function for its core business logic. Note that, we pass $Time(0)$ as the parameter for the base function - $AbstractTW_{psaf}$ - because this PSAF is not the PSAF that requires to create the past extent of time window for extracting properties (it only needs to identify an element in RS that is the ancestors of a particular input assertion). By using the timestamp (t) together with a delay time (d) of an input assertion (AS) an associated element in the result stream (RS) can be identified and provenance-related properties can be obtained.

5.2.5 The property stream ancestor function for a filter operation

Property stream ancestor function: PSAF for a filter operation - $Filter_{psaf}$
Let RS be a result stream of type $'a\ ASSERTION\ list$
Let AS be a provenance assertion of type $'b\ ASSERTION$
For a filter operation, the property stream ancestor function is defined as:

$$(*\ fn\ :\ 'a\ ASSERTION\ list\ ->\ 'b\ ASSERTION\ ->\ 'a\ ASSERTION\ *)$$

$$fun\ Filter_{psaf}\ RS\ AS\ =\ AbstractTW_{psaf}(Time(0))\ RS\ AS;$$

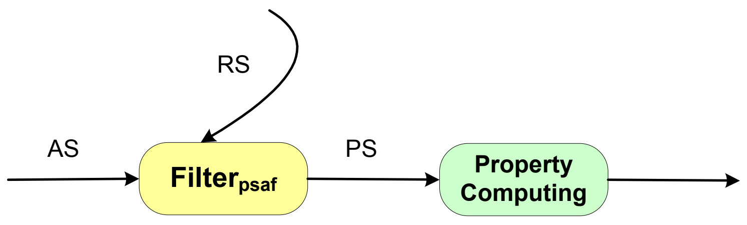


FIGURE 5.8: The definition of PSAF for a filter operation

As illustrated in Figure 5.8, the $AbstractTW_{psaf}$ function is used as the base function of the property stream ancestor function for a filter operation containing the function's business logic. The function takes a provenance assertion (AS) and a result stream (RS) as an input and generates an element of a property stream (PS) containing all provenance-related properties needed for further processing. Similar to the Map_{psaf} function, this PSAF extracts provenance-related properties from RS by utilizing the timestamp (t) together with a delay time (d) of an input provenance assertion (AS).

5.2.6 The property stream ancestor function for a sliding time window

The property stream ancestor function for a sliding time window is defined in Figure 5.9. For this function, only the duration of the time window (w) is indicated as the function parameter. The function takes a provenance assertion (AS) and a result stream (RS) as an input and produces an element of a property stream (PS) as a function output. As shown in the definition, this PSAF utilizes the $AbstractTW_{psaf}$ function for its core business logic. It extracts properties from RS by creating the extent of the sliding time window at the time that the operation produced the output using the function parameter - the duration of the time window (w).

Property stream ancestor function: PSAF for a sliding time window - $TW_{psaf}(w)$
Let RS be a result stream of type $'a\ ASSERTION\ list$
Let AS be a provenance assertion of type $'b\ ASSERTION$
Let w be the duration of the time window
For a sliding time window, the property stream ancestor function is defined as:

$$(*\ fn\ :\ TIME -> \ 'a\ ASSERTION\ list -> \ 'b\ ASSERTION$$
$$-> \ 'a\ ASSERTION\ *)$$

$$fun\ TW_{psaf}(w)\ RS\ AS\ =\ AbstractTW_{psaf}(w)\ RS\ AS;$$

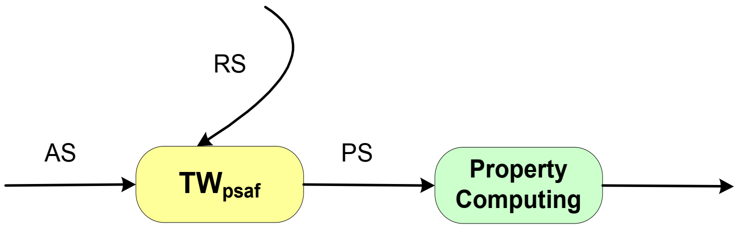


FIGURE 5.9: The definition of PSAF for a sliding time window

5.2.7 The property stream ancestor function for a length window

Property stream ancestor function: PSAF for a length window - $LW_{psaf}(l)$
Let RS be a result stream of type $'a\ ASSERTION\ list$
Let AS be a provenance assertion of type $'b\ ASSERTION$
Let l be the size of the length window
For a length window, the property stream ancestor function is defined as:

$$(*\ fn\ :\ int -> \ 'a\ ASSERTION\ list -> \ 'b\ ASSERTION$$
$$-> \ 'a\ ASSERTION\ *)$$

$$fun\ LW_{psaf}(l)\ RS\ AS\ =\ AbstractLW_{psaf}(l)\ RS\ AS;$$

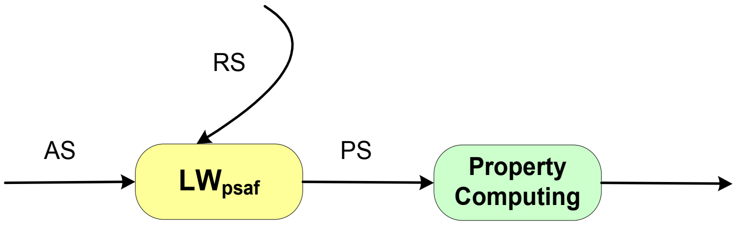


FIGURE 5.10: The definition of PSAF for a length window

The property stream ancestor function for a length window is defined in Figure 5.10. As presented in the function definition, this PSAF utilizes the size of the length window (l) as the function parameter. The $AbstractLW_{psaf}$ function is used as the base function of this PSAF containing the function's business logic. The function takes a provenance assertion (AS) and a result stream (RS) as an input. Then, provenance-related properties from RS are extracted by utilizing the size of the length window (l) and a sequence number (n). Finally, it returns an element of a property stream (PS) containing all provenance-related properties needed for further processing.

5.2.8 The property stream ancestor function for a time-window join

Property stream ancestor function: PSAF for a time-window join
- $JoinTW_{psaf}(w1, w2)$
Let $RS1, RS2$ be result streams of type $'a\ ASSERTION\ list$
Let $Assertion(Key(t, n, sid, d), \_)$ be a provenance assertion of type $'b\ ASSERTION$
Let $w1, w2$ be the duration of the time windows
For a time window join, the property stream ancestor function is defined as:

```
$(*\ fn\ :\ TIME\ *\ TIME\ ->\ 'a\ ASSERTION\ list\ ->\ 'a\ ASSERTION\ list$
  $->\ 'b\ ASSERTION\ ->\ 'a\ ASSERTION\ *)$

$fun\ JoinTW_{psaf}(w1, w2)\ \ RS1\ \ RS2\ \ (Assertion(Key(t, n, sid, d), \_))\ =$
$let$
    $val\ pl1\ =\ ExtractPropT(t--d, t--d--w1, RS1)$
    $val\ pl2\ =\ ExtractPropT(t--d, t--d--w2, RS2)$
    $val\ pList\ =\ pl1\ @\ pl2$
$in$
    $Assertion(Key(t, n, sid, d), pList)$
$end$
```

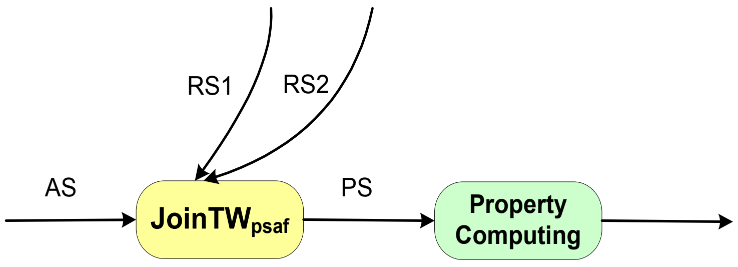


FIGURE 5.11: The definition of PSAF for a time-window join

As illustrated in Figure 5.11, we define the property stream ancestor function for a time-window join as a function that takes the duration of the time windows ($w1$ and $w2$) as the function parameters. The function takes a provenance assertion (AS) and two result streams ($RS1$ and $RS2$) as an input and generates an element of a property stream (PS) containing all provenance-related properties needed for further processing. To extract all related properties from two result streams ($RS1$ and $RS2$), the duration of the time windows ($w1$ and $w2$) is utilized to create the extent of the past time windows.

5.2.9 The property stream ancestor function for a length-window join

Property stream ancestor function: PSAF for a length-window join
- $JoinLW_{psaf}(l1, l2)$
Let $RS1, RS2$ be result streams of type $'a\ ASSERTION\ list$
Let $Assertion(Key(t, n, sid, d), \_)$ be a provenance assertion of type $'b\ ASSERTION$
Let $l1, l2$ be the size of the length windows
For a length-window join, the property stream ancestor function is defined as:

```
(* $fn : int * int -> 'a\ ASSERTION\ list -> 'a\ ASSERTION\ list$
  $-> 'b\ ASSERTION -> 'a\ ASSERTION$ *)

$fun\ JoinLW_{psaf}(l1, l2)\ RS1\ RS2\ (Assertion(Key(t, n, sid, d), \_)) =$
$let$
    $val\ pl1 = ExtractPropN(n, n - l1 + 1, RS1)$
    $val\ pl2 = ExtractPropN(n, n - l2 + 1, RS2)$
    $val\ pList = pl1\ @\ pl2$
$in$
    $Assertion(Key(t, n, sid, d), pList)$
$end$
```

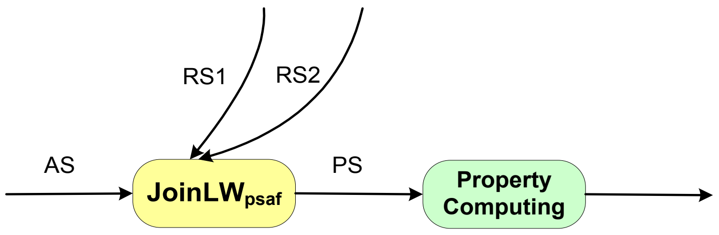


FIGURE 5.12: The definition of PSAF for a length-window join

The property stream ancestor function for a length-window join is defined in Figure 5.12. The function utilizes internal values of length windows ($l1$ and $l2$) as the function parameters. It takes a provenance assertion (AS) and two result streams ($RS1$ and $RS2$) as an input and produces an element of a property stream (PS) containing all properties needed for further processing. To extract properties from $RS1$ and $RS2$, it utilizes the size of the length windows ($l1$ and $l2$) in order to create the extent of the past count-based windows.

5.3 Algorithm for on-the-fly provenance queries

We now demonstrate how the on-the-fly provenance mechanism works and how property stream ancestor functions are composed together by means of an on-the-fly provenance query algorithm. The main concept of the on-the-fly provenance query algorithm is that it utilizes static provenance information (e.g. stream topology and stream operation parameters) stored in a provenance store to automatically create the internal processing of on-the-fly provenance queries. Similar to the persistent provenance query and the replay execution method, our on-the-fly query algorithm applies the idea of function composition. The algorithm takes a stream of provenance assertions generated during a stream system's execution as an input and produces query results which are provenance assertions containing provenance-related properties. The algorithm for on-the-fly provenance queries is presented in Figure 5.13.

```
Algorithm: On-the-fly provenance query
Let $KL$ be a list of event keys used as an input for the query
Let $psaf1, psaf2$ be look up tables (lists) for property stream ancestor functions
Let $sidList$ be a list of targeted stream IDs
Let $streamLut$ be a look up table (list) for input and output streams
Let $paramLut$ be a look up table (list) for stream operation parameters
Let $pFun$ be a property computing function

    (* fn : (STREAMID * ('a ASSERTION list -> 'b ASSERTION
     -> 'aASSERTION)) list * (STREAMID * ('a ASSERTION list
    -> 'a ASSERTION list -> 'b ASSERTION -> 'a ASSERTION)) list
    * STREAMID list * (STREAMID * int * InOut) list *
     (int * string * int * string) list * (KEY -> 'a list -> 'b)
    -> KEY list -> 'a ASSERTION list *)

    fun OTFpquery(psaf1, psaf2, sidList, streamLut, paramLut, pFun) KL =
      queryExec psaf1 psaf2 sidList streamLut paramLut pFun [ ] [ ] KL;

  (* fn : (STREAMID * ('a ASSERTION list -> 'b ASSERTION
  -> 'a ASSERTION)) list -> (STREAMID * ('a ASSERTION list
  -> 'a ASSERTION list -> 'b ASSERTION -> 'a ASSERTION)) list
  -> STREAMID list -> (STREAMID * int * InOut) list
  -> (int * string * int * string) list -> (KEY -> 'a list -> 'b)
  -> (STREAMID * 'a ASSERTION list) list
  -> 'a ASSERTION list -> KEY list -> 'a ASSERTION list *)

  fun queryExec psaf1 psaf2 sidList streamLut paramLut pFun B R [ ] = R
  | queryExec psaf1 psaf2 sidList streamLut paramLut pFun B R (k :: AS) =
  let
    val (pa as Assertion(Key(_, _, sid, d), _)) = ASU k;
    val (pa', B') = if (isFirstIStream sid streamLut) then
                        (PCU pFun pa, B)
                    else
                        (executePSAF psaf1 psaf2 streamLut pFun B pa,
                          Dequeue psaf1 streamLut paramLut B pa)
  in
    if (member sid sidList) then
      queryExec psaf1 psaf2 sidList streamLut paramLut pFun B' (pa' :: R) AS
    else
      queryExec psaf1 psaf2 sidList streamLut paramLut pFun (addElm B' pa') R AS
  end
```

FIGURE 5.13: Algorithm for on-the-fly provenance queries

Utility functions for the queryExec function

```
$datatype\ InOut\ =\ I \mid O$

$(*\ fn\ :\ STREAMID\ *\ (STREAMID\ *\ 'a\ *\ InOut)\ list\ ->\ bool\ *)$
$fun\ isFirstIStream\ \ sid\ \ streamLut\ =$
    $(containSID\ \ sid\ \ I\ \ streamLut)\ \ andalso\ \ not\ (containSID\ \ sid\ \ O\ \ streamLut);$


$(*\ fn\ :\ KEY\ ->\ 'a\ ASSERTION\ *)$
$fun\ ASU\ \ (key : KEY)\ =\ (Assertion(key, [\,]))$


$(*\ fn\ :\ (KEY\ ->\ 'a\ list\ ->\ 'b)\ ->\ 'a\ ASSERTION\ ->\ 'b\ ASSERTION\ *)$
$fun\ PCU\ \ func\ \ (Assertion(key, pl))\ =\ (Assertion(key, [func\ \ key\ \ pl]));$


$(*\ fn\ :\ (STREAMID\ *\ ('a\ ->\ 'b\ ASSERTION\ ->\ 'a\ ASSERTION))\ list$
$->\ (STREAMID\ *\ ('a\ ->\ 'a\ ->\ 'b\ ASSERTION\ ->\ 'a\ ASSERTION))\ list$
$->\ (STREAMID\ *\ int\ *\ InOut)\ list\ ->\ (KEY\ ->\ 'c\ list\ ->\ 'd)$
$->\ (STREAMID\ *\ 'a)\ list\ ->\ 'b\ ASSERTION\ ->\ 'a\ ASSERTION\ *)$
$fun\ executePSAF\ \ psaf1\ \ psaf2\ \ streamLut\ \ pFun\ \ B\ \ (pa\ as\ Assertion(Key(\_, \_, sid, \_), \_))\ =$
    $if\ (containsKeys\ \ [sid]\ \ psaf1)\ then$
        $let\ (*\ unary\ operation\ *)$
            $val\ rsID\ =\ hd\ \ (getRsID\ \ sid\ \ streamLut);$
            $val\ ((\_, rs) :: RS)\ =\ getElements\ \ [rsID]\ \ B;$
        $in$
            $PCU\ \ pFun\ \ ((assoc\ \ sid\ \ psaf1)\ \ rs\ \ pa)$
        $end$
    $else$
        $let\ (*\ binary\ operation\ *)$
            $val\ rsIDList\ =\ (getRsID\ \ sid\ \ streamLut);$
            $val\ ((\_, rs1) :: B1')\ =\ getElements\ \ [hd\ \ rsIDList]\ \ B;$
            $val\ ((\_, rs2) :: B2')\ =\ getElements\ \ (tl\ \ rsIDList)\ \ B;$
        $in$
            $PCU\ \ pFun\ \ ((assoc\ \ sid\ \ psaf2)\ \ rs1\ \ rs2\ \ pa)$
        $end$
```

Note that, because the on-the-fly provenance query algorithm utilizes several supporting functions (utility functions), to concisely illustrate our algorithm, we only present some significant utility functions. All utility functions used for the on-the-fly provenance query algorithm is detailed in Appendix C.

As illustrated in Figure 5.13, the on-the-fly provenance query algorithm consists of two main functions: $OTFpquery$ and $queryExec$. The $OTFpquery$ function is the entry-point function that takes look up tables for property stream ancestor functions ($psaf1$ - PSAFs for unary operations -, $psaf2$ - PSAFs for binary operations -), a list of target stream IDs (stream IDs used to terminate the query execution - $sidList$), a look up table for input and output streams ($streamLUT$) and a look up table for stream operation parameters ($paramLut$) and a property computing function ($pFun$) as input parameters. After taking all input parameters, the function returns a result list which is a list of provenance assertions containing provenance-related properties. The other function - $queryExec$ - is the recursive function that contains the business logic of the on-the-fly provenance query algorithm. The process of the function begins by receiving parameters passed by the $OTFpquery$ function. Then it iterates over the elements of a provenance assertion list ($k :: AS$) in order to execute queries (property propagation) on every input provenance assertion. For each provenance assertion, it is first processed by an assertion separation unit (ASU) to create an extra field for storing property information. Then, in the case that input provenance assertions are the assertions belonging to the first-input stream - a stream that enters a stream system from data sources and are first processed by stream operations - (each assertion needs to be checked by using $isFirstIStream$ function), they are executed by a property computing unit (PCU) in order to extract properties. After that, for each assertion, if its stream ID is not the IDs in the stream ID list ($sidList$), it will be processed by its associated property stream ancestor function (PSAF) and property computing unit (PCU) using the $executePSAF$ function. Every output of property computing (an element of a result stream RS) is inserted to a data buffer B. Finally, the $queryExec$ function will be called recursively until no provenance assertions remain in the provenance assertion list (AS).

As presented in the algorithm, to support the dynamic execution of on-the-fly provenance queries, a number of intermediate provenance assertions (elements of result streams RSs) need to be temporarily stored in a data buffer. This processing technique potentially increases the amount of memory consumed by our stream provenance system. However, as demonstrated in the on-the-fly query algorithm, the size of the temporary data buffer can be controlled by using $Dequeue$ function. The function utilizes stream operation parameters ($paramLut$) - especially a data window size for each stream operation - to eliminate assertions that is no longer need for further processing. Therefore, we expect that the amount of memory consumed by our stream provenance system should depend on types of stream operations and the size of the data window utilized in a stream system. We will present the memory consumption evaluation for our on-the-fly provenance query mechanism in Chapter 6.

An example output produced by the on-the-fly provenance query algorithm is now presented. We use the same example stream process flow presented in Figure 5.2.

A list of input provenance assertions (event keys) used in this example:

$$\begin{aligned}- val\ S\ =\ &[Key(Time\ 1,1,StreamID\ 1,Time\ 0),Key(Time\ 2,1,StreamID\ 2,Time\ 0),\\ &Key(Time\ 4,1,StreamID\ 3,Time\ 2),Key(Time\ 7,1,StreamID\ 4,Time\ 3),\\ &Key(Time\ 11,2,StreamID\ 1,Time\ 0),Key(Time\ 12,2,StreamID\ 2,Time\ 0),\\ &Key(Time\ 13,2,StreamID\ 3,Time\ 1),Key(Time\ 15,2,StreamID\ 4,Time\ 2),\\ &Key(Time\ 21,3,StreamID\ 1,Time\ 0),Key(Time\ 22,3,StreamID\ 2,Time\ 0),\\ &Key(Time\ 24,3,StreamID\ 3,Time\ 2),Key(Time\ 28,3,StreamID\ 4,Time\ 4)]\end{aligned}$$

Required parameters for the on-the-fly query algorithm:

```
$(*\ property\ stream\ ancestor\ functions\ *)$
$-\ val\ psafLut1\ =\ [(StreamID(4), Map_{psaf})];$
$-\ val\ psafLut2\ =\ [(StreamID(3), JoinTW_{psaf}(Time(5), Time(5)))];$

$(*\ a\ list\ of\ targeted\ stream\ ids\ *)$
$-\ val\ sidList\ =\ [StreamID(4)];$

$(*\ a\ look\ up\ table\ for\ input\ and\ output\ streams\ *)$
$-\ val\ streamLut\ =\ [(StreamID(1),1,I),(StreamID(2),1,I),(StreamID(3),1,O)$
                     $,(StreamID(3),2,I),(StreamID(4),2,O)];$

$(*\ a\ look\ up\ table\ for\ stream\ operation\ parameters\ *)$
$-\ val\ paramLut\ =\ [(1,\text{“}w1\text{”},5,\text{“}T\text{”}),(1,\text{“}w2\text{”},5,\text{“}T\text{”}),(2,\text{“}w\text{”},0,\text{“}T\text{”})];$

$(*\ property\ computing\ function\ and\ its\ corresponding\ look\ up\ table\ *)$
$fun\ CovArea\ \ (Key(t,n,sid,d))\ \ [\,]\ =\ getCovArea\ \ sid\ \ t\ \ covAreaLut$
$|\ CovArea\ \ (Key(t,n,sid,d))\ \ (p :: pList)\ =\ p\ \wedge\ CovArea\ \ (Key(t,n,sid,d))\ \ pList$

$fun\ getCovArea\ \ sid\ \ ts\ \ [\,]\ =\ \text{“”}$
$|\ getCovArea\ \ sid\ \ ts\ \ ((id,minT,maxT,area) :: aTable)\ =$
   $if(sid\ =\ id)\ \ andalso\ \ (maxT\ GTE\ ts)\ \ andalso\ \ (ts\ GTE\ minT)\ then\ \ area$
   $else\ \ getCovArea\ \ sid\ \ ts\ \ aTable$

$-\ val\ covAreaLut\ =\ [(StreamID(1),Time(1),Time(10),\text{“}A\text{”}),$
      $(StreamID(2),Time(1),Time(10),\text{“}D\text{”}),(StreamID(1),Time(11),Time(20),\text{“}B\text{”}),$
      $(StreamID(2),Time(11),Time(20),\text{“}E\text{”}),(StreamID(1),Time(21),Time(30),\text{“}C\text{”}),$
      $(StreamID(2),Time(21),Time(30),\text{“}F\text{”})];$
```

The on-the-fly provenance query for this example can be performed as follows:

```
$(*\ execute\ the\ on-the-fly\ provenance\ query\ function\ *)$
$-\ OTFpquery(psafLut1,psafLut2,sidList,streamLut,paramLut,CovArea)\ \ S;$
```

$(*\ \textit{the provenance query output}\ *)$
$> [Assertion(Key(Time\ 28, 3, StreamID\ 4, Time\ 4), [\text{“}CF\text{”}]),$
$\quad Assertion(Key(Time\ 15, 2, StreamID\ 4, Time\ 2), [\text{“}BE\text{”}]),$
$\quad Assertion(Key(Time\ 7, 1, StreamID\ 4, Time\ 3), [\text{“}AD\text{”}])]\ :\ string\ ASSERTION\ list$

5.4 A case study for on-the-fly provenance queries

The previous sections have presented a conceptual design, programmatic specifications of the property stream ancestor functions and an on-the-fly query algorithm for our stream-specific provenance query solution. The question now arises as to how to apply the design in practical stream-based applications. In this section we present our stream-specific provenance query solution - on-the-fly provenance queries - in action. A stream processing system containing five common primitive stream operations is used as a practical case study to demonstrate how our novel provenance solution can be applied in the example application and the accuracy of the provenance solution for obtaining the provenance of individual stream elements. Note that a processing flow used as an example in this case study is the same synthetic flow presented in the provenance query chapter. The example Standard ML code for this case study is also provided in Appendix D.

The synthetic processing flow of a stream processing system is presented in Figure 5.14. The processing flow contains five stream operations including two *Map* operations, *Filter*, *Sliding time window (TW)* and *Time window join (JoinTW)*. Each stream operation is assigned a unique operation identifier (operation ID). For example, the sliding time-window operation is assigned no.3 as its operation ID. Moreover, each data stream (either input or output stream) for each stream operation is labeled with a unique identifier. For example, stream 2 is the input stream of the filter operation.

From the processing flow presented in Figure 5.14, an internal process of on-the-fly queries can be derived, as shown in Figure 5.15. All components and internal streams illustrated in the figure are automatically generated by using topology configuration parameters and stream operation parameters stored in a provenance store. Seven assertion separation units are used as intercepting components for receiving provenance assertions created and streamed to a provenance service by the case study application. After receiving provenance assertions, each assertion separation unit redirects detected provenance assertions to their associated property stream ancestor functions (PSAF). Based on the stream operations in the stream processing flow, six PSAFs are utilized for on-the-fly provenance queries including two instances of Map_{psaf} (for operation ID 1), $Filter_{psaf}$, Map_{psaf} (for operation ID 4), TW_{psaf} and $JointTW_{psaf}$. Each PSAF extracts provenance-related properties and sends provenance assertions containing all required properties to a property computing unit (PCU) for property processing.

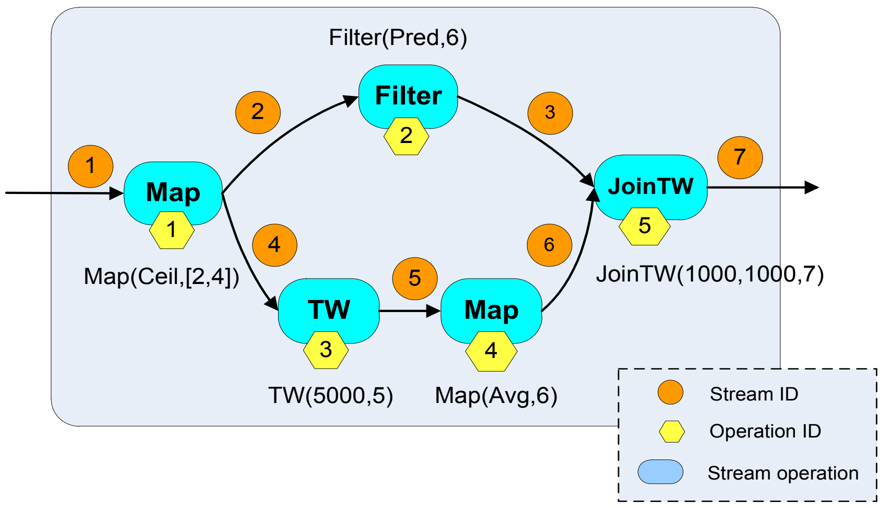


FIGURE 5.14: The processing flow of a stream processing system

Figure 5.16 presents a set of provenance assertions recorded as a stream during the execution of the synthetic processing flow of the case study application. The provenance assertions presented in the figure consists of the following fields: a set of fields representing an event key of a stream element (timestamp, seqno, stream id and delay) and an extra property field (prop) used to temporarily store provenance related properties for property propagation. Note that the index field - a reference for each provenance assertion - is used only for presentation clarity. It is not used in our practical implementation. In this example three input stream events of stream 1 are fed into the stream system. The provenance assertions for these input stream events are the assertions at the index #1, #8 and #15. Three final output stream events of the stream 7 are generated by the stream system as well. The provenance assertions for these output stream events includes the assertion at the index #7, #14 and #21.

We now demonstrate how to perform on-the-fly provenance queries in order to obtain the provenance of each individual stream element. In this case study a total delay time for event processing is used as an example provenance-related property that we intend to propagate through the execution of on-the-fly provenance queries. We use an example scenario of an anomaly or unusual event occurring at some point in time during stream processing. To detect the anomaly it is necessary to obtain the total delay time for each individual stream processing result and then compare it with a threshold that is defined as the maximum delay constraint for stream processing. With the use of query results (properties: total delay time) produced by on-the-fly queries, we can track that the problem occurred during the execution of a stream processing system and we can also determine how to solve this unusual problem in near real-time.

In Figure 5.17 a provenance graph related to the provenance assertions of stream elements generated in Figure 5.16 is presented. Each node labeled with an index number

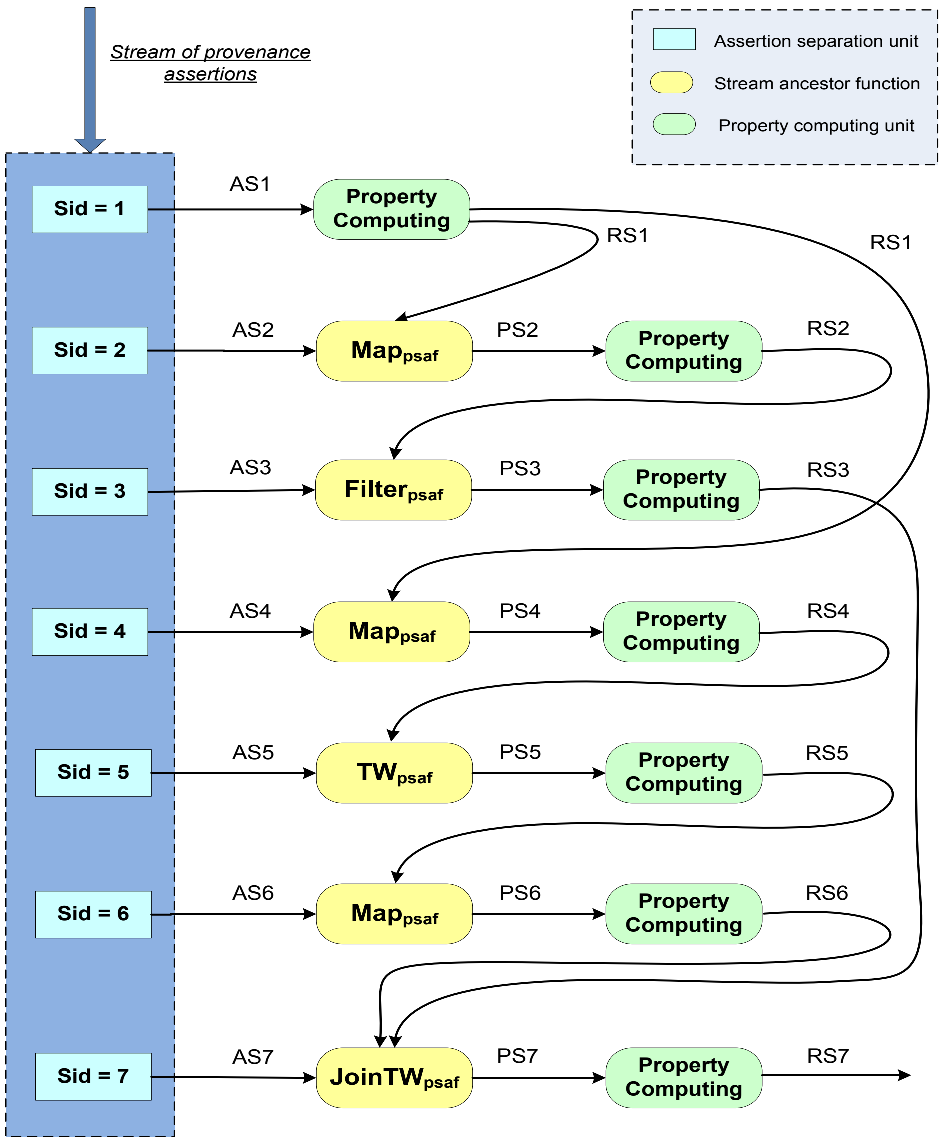


FIGURE 5.15: An internal process of on-the-fly provenance queries for the case study

represents individual provenance assertions. This provenance graph is used to describe dependencies among each individual provenance assertion. From the provenance graph we demonstrate how provenance-related properties can be propagated from ancestor provenance assertions to their descendant provenance assertions, traversing the provenance graph from left to right. For example, the provenance assertion at index #18 is utilized by the property stream ancestor function for the time-window (TW_{psaf}) to identify ancestor provenance assertions (index #3, #10 and #17). Then, provenance-related properties - delay time for processing - represented as integer values in brackets are propagated from these three ancestor provenance assertions to the assertions at the index #18.

Figure 5.18 illustrates an example of how provenance-related properties (delay time for processing) are propagated during the execution of on-the-fly provenance queries.

index	timestamp	seqno	stream_id	delay	prop
1	1279398105675	1	1	0	-
2	1279398105684	1	2	9	-
3	1279398105684	1	4	9	-
4	1279398105693	1	3	9	-
5	1279398105701	1	5	17	-
6	1279398105702	1	6	1	-
7	1279398105703	1	7	1	-
8	1279398107678	2	1	0	-
9	1279398107678	2	2	0	-
10	1279398107678	2	4	0	-
11	1279398107680	2	5	2	-
12	1279398107681	2	3	3	-
13	1279398107682	2	6	2	-
14	1279398107683	2	7	1	-
15	1279398109678	3	1	0	-
16	1279398109678	3	2	0	-
17	1279398109678	3	4	0	-
18	1279398109680	3	5	2	-
19	1279398109681	3	3	3	-
20	1279398109682	3	6	2	-
21	1279398109683	3	7	1	-

FIGURE 5.16: Provenance assertions generated during the execution of the case study application

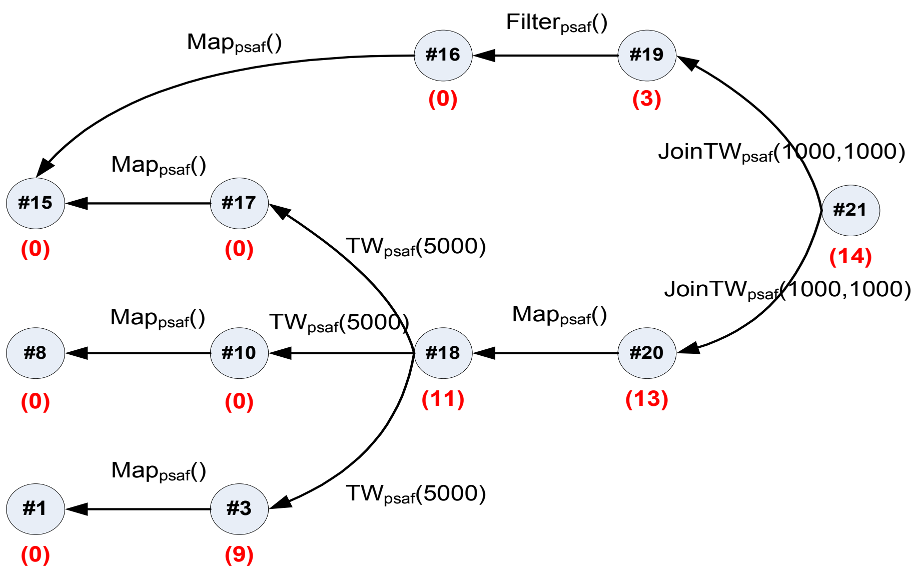


FIGURE 5.17: The provenance graph for the example on-the-fly provenance query

In this example we present the execution of on-the-fly provenance queries for obtaining the query result of the third input stream event (provenance assertion at the index #15). The process begins with each provenance assertion being detected by an assertion separation unit which adds an empty extra property field (prop) and then the provenance assertion is passed to its associated property stream ancestor function (PSAF) for processing. After receiving the provenance assertion, PSAF is executed (it identifies ancestor provenance assertions and extracts required provenance-related properties - delay time). After that PSAF sends the provenance assertion (element in a property stream - PS) containing all required properties (delay time of ancestor provenance assertions) to a property computing unit which finally computes the properties and produces a final provenance query result (element in a result stream - RS).

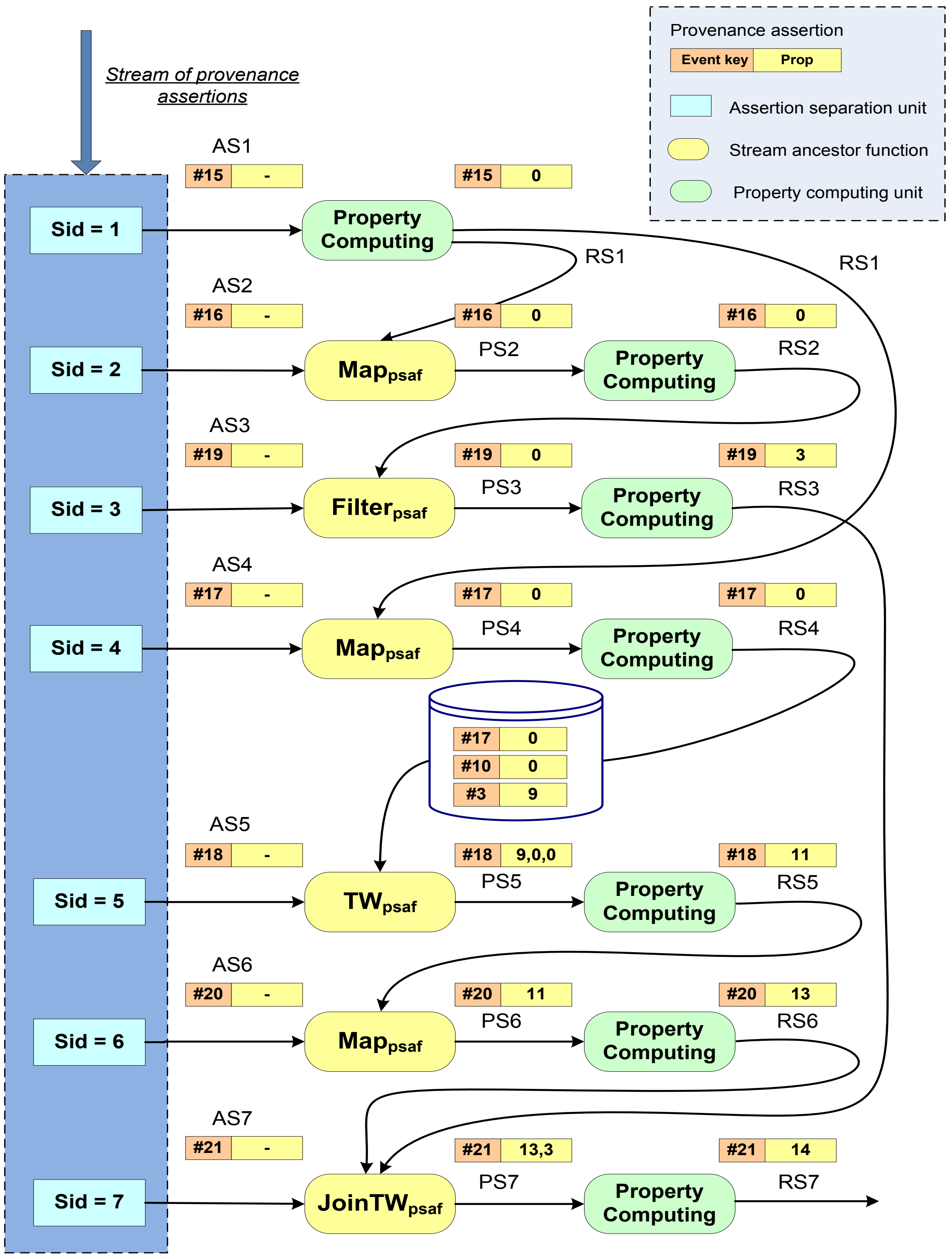


FIGURE 5.18: An internal process of on-the-fly provenance queries for the case study

As shown in Figure 5.18, we can describe the processing steps of on-the-fly provenance queries when the provenance assertion of the stream 5 arrives as follows. The provenance assertion (index #18) is detected by the assertion separation unit for stream 5 (sid = 5) and it is sent as an element of the assertion stream 5 (AS5) to PSAF (TW_{psaf}). Then, the provenance assertion is used by TW_{psaf} to identify ancestor provenance assertions (index #3, #10 and #17) temporarily stored in the buffering queue of the result stream 4 (AS4). After that, all required properties (delay time for processing: 9,0 and 0 milliseconds) from ancestor provenance assertions are extracted and they are contained in the element of the property stream 5 (PS5) that is sent to the property computing unit (PCU). Then, PCU computes the properties of the ancestor provenance assertions together with the property of the current assertion (delay time of the provenance assertion index #18: 2 milliseconds). Finally, PCU produces the output which is the provenance assertion containing the new property (accumulated total delay time: 11 milliseconds) as the element of the result stream 5 (RS5). In addition, as illustrated in Figure 5.18, the process of on-the-fly provenance queries starts with the processing of provenance assertions in the assertion stream 1 (AS1). Then, the process iterates through each successive internal stream until the final internal stream (assertion stream 7 - AS7) is reached and the final query output - element of the result stream 7 (RS7) containing total delay time: 14 milliseconds - is generated.

5.5 Summary

We started this chapter by outlining the characteristics that the stream-specific provenance query mechanism should have. Then, a novel on-the-fly provenance query mechanism for streams was presented. The key concepts of our on-the-fly provenance query mechanism is that we exploit a provenance service - a central component of the stream provenance architecture - as a stream component. Provenance assertions streamed to the provenance service are processed on-the-fly continuously without any requirement to store them persistently. We extend the persistent provenance query mechanism for streams presented in Chapter 4 by introducing the idea of property propagation. We do not propagate provenance-related properties through the processing flow of a stream processing system because this requires the modification of the internal processing of stream operations, instead properties are propagated inside the provenance service using accumulators. By utilizing a new version of stream ancestor functions (property stream ancestor functions), provenance-related properties can be propagated and provenance query results are produced as a stream.

We now revisit the main contribution of this chapter: a stream-specific provenance query mechanism that can create and compute provenance queries automatically without requiring provenance assertions to be stored persistently. To support this main contribution we presented the conceptual design of on-the-fly provenance queries which described the basic assumptions and how on-the-fly provenance queries can be performed inside a provenance service. We then introduced the programmatic specification of the stream ancestor functions designed specifically for working with stream-specific provenance query mechanism (property stream ancestor functions - PSAF) and the on-the-fly provenance query algorithm. Similar to the specifications of stream ancestor function previously described, we defined PSAFs and the on-the-fly query algorithm by using the general-purpose functional programming language - Standard ML (SML). Using SML allows us to validate our design and it also allows us to prove whether our functions produce correct outputs. In the next chapter, the evaluation of the implementation of our stream provenance solutions including the stream-specific provenance query mechanism will be presented. This evaluation will demonstrate the effectiveness and the impact of our stream provenance solution under different experimental conditions.

Chapter 6

Evaluation

In this chapter, we evaluate the implementation of our provenance solution presented in Chapters 3, 4 and 5 - our fine-grained provenance model for streams and both the persistent provenance query and the stream-specific provenance query mechanism. Our evaluation is conducted across four different aspects. Firstly, the storage overhead of the implementation when provenance information (provenance assertions) is stored persistently in a provenance store as the number of stream components increases. Two message payload sizes were considered to demonstrate the effect of different message sizes on the storage requirement for provenance recording. Secondly, the impact of provenance recording (throughput) in a controlled environment as the number of stream components increases. We also observed the impact of provenance recording on different implementations that have different time delays for processing. Thirdly, the memory consumption for a provenance service. Finally, the time latency of the on-the-fly provenance query approach as the number of stream components increases. Recommendations on the use of our system in applications are also given.

For this evaluation, there are four key contributions:

1. The storage overhead resulting from recording provenance information can be reduced significantly by using our storage reduction technique. The marginal cost of storage consumption for our provenance solution is constant and predictable (about 5 MB per additional stream component for 100,000 input stream events).

2. The provenance recording impact evaluation demonstrates that our provenance solution does not have a significant effect on the normal processing of stream systems. There are a 4% overhead for the store provenance assertions approach and a 7% overhead for the on-the-fly provenance query approach.

3. According to both the memory consumption and the time latency evaluations, it is shown that our on-the-fly provenance query approach offers low-latency processing (average time latency: 0.3 ms per additional component) with reasonable memory

consumption (the marginal cost of memory consumption for the map operation experiment is 0.5 MB and that for the time-window experiment is 1.8 MB).

4. A set of recommendations for application developers when designing and implementing provenance systems for streams.

The rest of this chapter is organized as follows: First, an implementation of our provenance system for streams is detailed. Next, the environment in which the experimental evaluations were performed is described. Then, four different sets of performance evaluation experiments, which include storage overhead, system throughput, memory consumption and time latency, and analyses of experimental results, are presented. After that, analysis conclusions and recommendations on the use of our system in stream processing applications are given. Finally, the chapter is concluded.

6.1 Implementation design

In this section, we discuss the design and implementation of our stream provenance system. We first describe the detailed design of the provenance service. Then, a justification of the technologies used in the implementation is discussed.

6.1.1 The provenance service

In our stream provenance system, the provenance service is utilized as a central component that provides provenance recording and querying functionalities. It is a wrapper for three important internal components including the provenance recording module, the on-the-fly provenance query module and the provenance store. The component diagram of the provenance service is shown in Figure 6.1

There are two operation modes of the provenance service that can be configured at system registration time: store provenance assertions mode, and on-the-fly provenance query mode. The *store provenance assertions mode* is used to support the persistent provenance query mechanism, which provenance assertions need to be stored persistently in a persistent storage (provenance store) before performing a variety of provenance queries. Another operation mode, *on-the-fly provenance query mode*, is utilized to support the on-the-fly provenance query mechanism that incoming provenance assertions are automatically processed in real-time without any requirement to store them in a persistent storage.

During a stream-based application's execution, incoming messages (provenance assertions) generated by all stream operations involved in the execution are received by the *assertion dispatcher*, which is responsible for routing the assertions to the appropriate

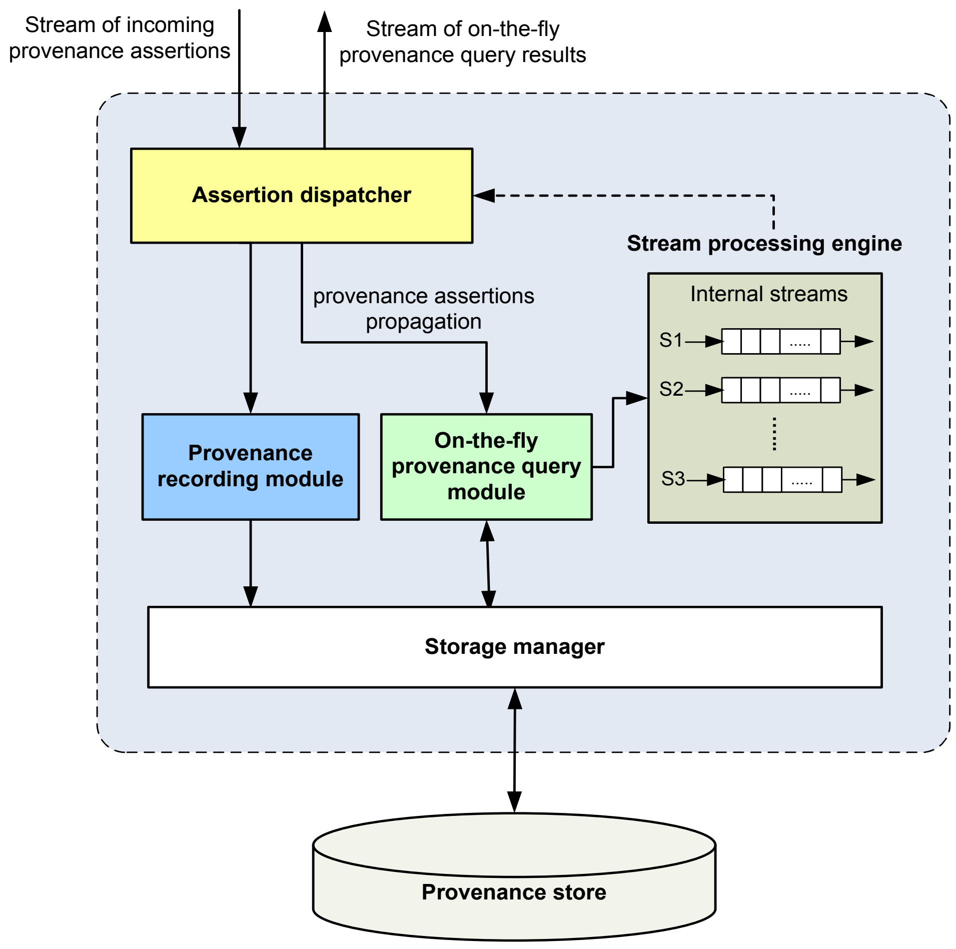

Stream of incoming provenance assertions
Stream of on-the-fly provenance query results
Assertion dispatcher
Stream processing engine
provenance assertions propagation
Internal streams
S1
S2
S3
Provenance recording module
On-the-fly provenance query module
Storage manager
Provenance store

FIGURE 6.1: The provenance service architecture

internal component. Depending on the operation modes of the provenance service configured, the provenance assertions can be sent to either the provenance recording module or the on-the-fly provenance query module. In the store provenance assertions mode, the *provenance recording module* takes each provenance assertion from the assertion dispatcher and converts the assertion, encoded in the communication medium's format (JMS format), into the internal format of the provenance service. Then, the converted provenance assertion is sent to the storage manager, which is the internal component responsible for recording the assertion into the provenance store. In the provenance service architecture, the *storage manager* is utilized as a storage abstraction layer. It allows for several internal components of the provenance service to access the provenance store (back-end data storage) through various provenance specific functions provided by the storage manager (e.g. provenance recording and static provenance retrieval functions). The use of the storage manager also enables different types of back-end data storage to be deployed without any changes to the provenance service implementation. So, after receiving messages (provenance assertions) from the provenance recording module, the storage manager transforms the provenance assertions into SQL statements. Then, the storage manager communicates with the provenance store via a back-end data storage connector (e.g. JDBC driver) and records the received provenance assertions to the provenance store.

The *on-the-fly provenance query module* plays an important role in the on-the-fly provenance query mode. Similar to the provenance recording module, it takes each provenance assertion from the assertion dispatcher and transforms the assertion, encoded in the communication medium's format (JMS format) into the internal format of the provenance service. Then, it puts each transformed provenance assertion into an inbound queue (internal queue) of the *stream processing engine* (Esper stream engine) in order to perform straight-through processing on provenance assertions without any requirement to store them. The on-the-fly provenance query module is not only responsible for routing provenance assertions to the stream processing engine but also for controlling the execution of stream processing engine. It queries the provenance store through the storage manager at system initialization time to obtain auxiliary information pertaining to the processing flow of a stream-base application (e.g. stream topology, stream operations used and parameters configured). This auxiliary information is used to automatically create an internal process (internal streams) of on-the-fly provenance queries inside the stream processing engine and it is also used to support the dynamic execution of on-the-fly provenance queries. The processing of on-the-fly provenance queries inside the stream engine is performed interleaved continuously with the execution of a stream-based application. Once each query result is generated by the stream processing engine, it is streamed back to a client application that subscribes to receive query results from the provenance service via the assertion dispatcher component.

One of the significant challenges for the implementation of on-the-fly provenance queries is how to create the internal process (internal streams) of on-the-fly provenance queries dynamically inside the stream processing engine (Esper) and support the continuous execution of on-the-fly queries over streams of provenance assertions. This challenge is addressed by utilizing EPL - Event Processing Language (sometime called StreamSQL [120]) - provided by Esper. At system initialization time, we query the provenance store to obtain stream topology information and this information is then used to create EPL statements representing internal components and internal streams of on-the-fly provenance queries. As discussed in Section 5.1, there are three types of internal components used in the processing of on-the-fly provenance queries: 1) an assertion separation unit for each stream, 2) a property stream ancestor function (PSAF) for each stream operation and 3) a property computing unit. All components of all types of internal components for on-the-fly provenance queries are created as EPL statements and they are registered with Esper stream processing engine in order to enable them to process streams of provenance assertions on-the-fly continuously at execution time. An example of EPL statements utilized in our implementation is illustrated in Figure 6.2.

```
(* Assertion separation unit *)

insert into AS2(timeStamp, seqNo, streamID, delay, property)
select timeStamp, seqNo, streamID, delay, null
from assertions
where streamID  =  2


(* Property stream ancestor function for time window operation  -  TW_psaf *)

insert into PS2(timeStamp, seqNo, streamID, delay, property)
select a.timeStamp, a.seqNo, a.streamID, a.delay,
              PropExtract.compute(
                     (select  *  from RS1),
                     a.timeStamp  -  a.delay,
                     a.timeStamp  -  a.delay  -  w
              )
from AS2  a


(* Property computing unit *)

insert into RS2(timeStamp, seqNo, streamID, delay, property)
select timeStamp, seqNo, streamID, delay,
       PropCal.compute(timeStamp, seqNo, streamID, delay, property)
from  PS2
```

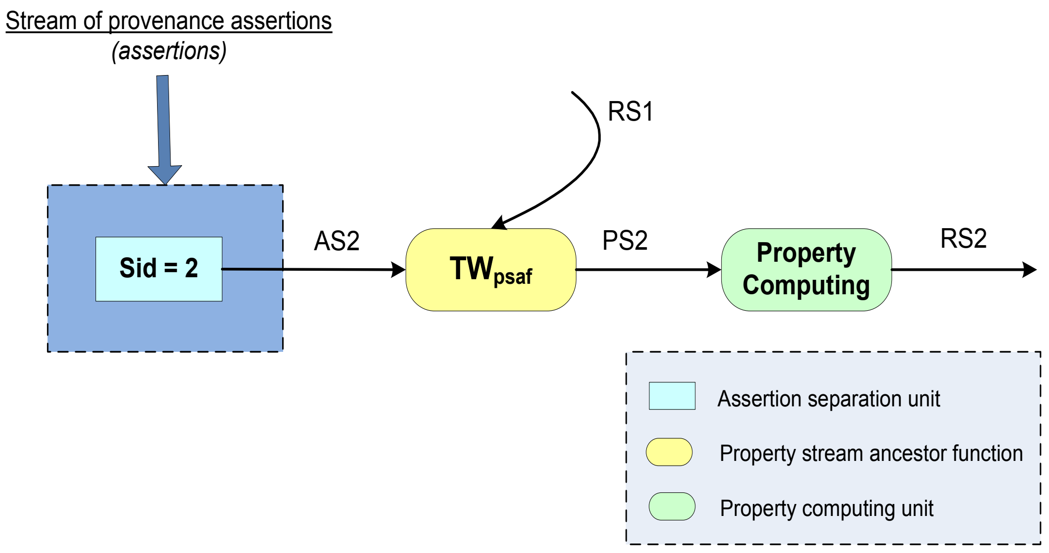

Stream of provenance assertions
(assertions)
RS1
Sid = 2
AS2
TW_{psaf}
PS2
Property Computing
RS2
Assertion separation unit
Property stream ancestor function
Property computing unit

FIGURE 6.2: An example of EPL statements utilized in our implementation

As shown in Figure 6.2, three example EPL statements representing three different types of internal components are presented. Parameters and internal stream IDs that are replaced by the on-the-fly provenance query module according to stream topology information are presented in bold font. In this example, the first statement - assertion separation unit - detects each provenance assertion of stream 2 from streams of provenance assertions ($assertions$) and then sends it as an element of the assertion stream $AS2$ to the property stream ancestor function statement (TW_{psaf}). The property stream ancestor function receives elements from both the assertion stream $AS2$ and the result stream $RS1$ and then produces an element of the property stream $PS2$ based on the element of $AS2$ and the provenance related properties extracted from the elements in $RS1$ (by using $PropExtract$ function). Finally, the property computing unit statement receives an element of the property stream $PS2$, performs property processing (by using $PropCal$ function) and produces an output containing a new property as an element of the result stream $RS2$.

6.1.2 Technologies used

We now discuss the technologies used in our implementation and the rationale behind their selection. Heterogeneity of platform is an important issue in distributed systems. This heterogeneity motivates us to choose Java as the implementation language. By using Java, our implementation can be run under several operating systems (that have Java Virtual Machines) such as Windows, Linux and Solaris without recompilation. The selection of Java satisfies the ease of installation requirement as well, due to the fact that implementation is not needed to re-compile the code for a specific platform. Architecture scalability is another significant issue. This scalability motivates us to select Apache ActiveMQ – an open source message broker software [128] – as our messaging framework. ActiveMQ provides a transport infrastructure that enables all components of the implementation to work together. It also provides a scalable environment in which we can easily deploy additional components. In addition, ActiveMQ allows us to seamlessly handle interactions with other applications using disparate technologies. For the transport protocol, the Java Message Service (JMS) Protocol [122] has been selected. JMS allows components of our systems to communicate between each other asynchronously via a point-to-point model - a model where senders and receivers exchange intermediate messages by using message queues. Moreover, JMS enables the implementation to stream messages preserving their order. This satisfies the required characteristics that a stream processing system should have.

To facilitate the stream processing environment, real-time processing and instantaneous response are the key features. Esper [46] - an open source event stream processing engine for event-driven architectures - is used as our stream processing engine because it satisfies these requirements. Esper also conforms to other unique characteristics of a stream processing system: straight-through processing, continuous and long-running queries and order based and time based operations. By implementing Esper inside ActiveMQ components, each stream processing unit can exploit stream-specific functions such as stream filters, continuous queries, and pattern matching. Finally, MySQL [106] is used as a back-end data storage that offers long-term persistence for provenance assertions recorded by stream-based applications. The selection was made for a number of reasons. First, MySQL supports multi-user access which allows for different internal components of provenance service to simultaneously access to a back-end database. Second, MySQL supports several different storage engines that allows us to choose the one that is most suitable for particular applications. In addition, MySQL provides the mysql JDBC type 4 driver which can communicate directly with databases. This JDBC driver is not required to translate the requests or to pass through middleware layers; so that it enhances performance considerably compared to other types of JDBC.

It is necessary to note that in all of our experiments, we choose MyISAM for MySQL as our storage engine. The selection of MyISAM is based on the fact that MyISAM is designed specifically for managing non-transactional tables. It provides high-speed storage and retrieval that satisfy the unique requirements for stream processing. It is also considered to be the storage engine for MySQL that offers the smallest disk space consumption. Therefore, the selection of MyISAM directly supports our experiment pertaining to the storage overhead of our implementation when provenance information is recorded persistently in the provenance store.

The following is a list of the software used in the implementation:

1. Apache ActiveMQ 5.x - Message broker software
2. Esper 3.x - Stream processing engine
3. MySQL 5.x and MyISAM storage engine - Back-end relational database management system (RDBMS)
4. Sun Java Developer Kit (JDK) 1.5.x

6.2 Evaluation environment

For all experiments used in this evaluation, the experimental set-up was as follows: Our provenance service and a stream processing system were hosted on a server computer with Intel Xeon Quad Core CPU running at 1.60GHz and 4 GB of memory (RAM). The server runs the Red Hat Linux operating system version 4.1.1-52 with Linux kernel version 2.6.18-8.1.4.el5. To store provenance information, our implementation used MySQL database 5.0.22 as a back-end data storage and MyISAM - the default storage engine of MySQL - is used as our storage engine.

All our application components, including the stream processing system and the provenance service, were implemented in Java and were run using the the Java 1.5.0_15-b04 Server Virtual Machine with Java HotSpot Just-In-Time compilation enabled. The minimum and maximum heap sizes that are set to be allocated for Java Virtual Machine at the initialization time are 1024 MB (1 GB). The provenance service uses Connector/J - the JDBC driver for MySQL (type 4) - as an API to communicate with the back-end data storage (provenance store). Furthermore, all components used in our implementation including the stream processing system, the provenance service and the message broker software (Apache ActiveMQ 5.3.1) were run within the same Java Virtual Machine.

In all our experiments, map operations are mainly used. This selection was made for several reasons. For the first two sets of experiments (the storage overheads and the system throughput), both of them mainly focus on the impact of provenance recording on system performance with respect to storage consumption and runtime overheads. Because we utilize the same provenance recording method for all stream operations - provenance assertions of individual elements of input streams have to be recorded by each stream operation during application's execution, map operations are considered as a representation for all different types of stream operations. In the case of the memory consumption and the time latency experiments, both of them aim to examine the effect of on-the-fly provenance query processing. Because of the unique characteristics of our on-the-fly provenance mechanism that require provenance assertions to be temporarily buffered in memory and utilize stream operation parameters (e.g. the size of data window) to control the size of data buffer, we therefore considered to use two different types of stream operations (map and time-window operations). The use of map and time-window operations allows us to demonstrate the effect of our on-the-fly provenance mechanism when a number of assertions buffered increases.

6.3 Storage overheads for provenance collection

We now evaluate our fine-grained provenance solution in terms of storage consumption on a synthetic processing flow of a stream processing system. This evaluation aims to establish that our storage reduction technique can significantly reduce the amount of storage space consumed when provenance is collected. We compare storage space consumed by the implementation of our provenance solution applying our storage reduction technique (optimized stream ancestor function) to another implementation that does not employ the reduction technique (unoptimized stream ancestor function).

The experiments were run with a stream producer submitting input stream events to our stream processing system. After stream events were fed into the stream system, provenance assertions for individual stream events were recorded to a provenance store. The synthetic processing flow used in our storage experiments is a linear stream processing flow where stream operations are chained together and each component takes input events from a previous component. In each set of experiments, we first measured the original storage cost of provenance recording (without applying the storage reduction technique). This experiment aims to demonstrate how much storage space the system requires to store every intermediate stream element. Then, we measured the storage cost resulting from the system that applies our reduction technique. By analyzing the storage measurements collected from these experiments, we can indicate the percentage of storage space we can save when applying our reduction technique.

To understand the variation of storage overheads incurred by the system, the number of stream components used in the experiments was increased from 2 up to 15. Two message payload sizes, 100 Bytes and 1 Kbytes, were considered to demonstrate the storage overheads for different message sizes. For each test, the number of stream events fed to the stream system was 100,000 stream events. The total number of provenance assertions recorded in a provenance store can be calculated as the number of events fed multiplied by the number of data streams used. This number of provenance assertions is indicative of the amount of data held within a provenance store.

Table 6.1: Mathematical Symbols for storage formulas

Symbol	**Definition**
$SC_{SAF-unopt}$	Storage cost of the unoptimized stream ancestor function
$SC_{SAF-opt}$	Storage cost of the optimized stream ancestor function
$MC_{SAF-unopt}$	Marginal cost of the unoptimized stream ancestor function
$MC_{SAF-opt}$	Marginal cost of the optimized stream ancestor function
SS	The percentage of storage saved
k	Size of an event key
e	Size of an event's content
m	No. of messages fed to a stream processing system
fs	No. of first-input streams - streams that enter a stream system from data sources and are first processed by stream operations
is	No. of intermediate streams - streams that are sent or received between stream operations in a stream processing system

The storage cost for provenance collection in our provenance solution for streams can be explained in terms of some straightforward mathematical formulas. We can use these formulas to predict the amount of storage consumed by our system when provenance is collected. Table 6.1 lists the mathematical symbols used in our storage formulas. For the unoptimized stream ancestor function approach (unoptimized SAF) which stores every intermediate stream element persistently in a provenance store, we can derive the storage cost from the following equation:

$$SC_{SAF-unopt} = ((k + e) * m) * (fs + is)$$

For the optimized stream ancestor function approach (optimized SAF), only contents of stream elements that act as the first input to a stream processing system are recorded. The content of each intermediate event is discarded and only its event key is stored. Hence, the storage cost for provenance collection can be calculated as follows:

$$SC_{SAF-opt} = (((k + e) * m) * fs) + ((k * m) * is)$$

By utilizing the storage formulas previously presented, we can derive the percentage of storage saved (SS) from the following equation:

$$SS = \left(\frac{SC_{SAF-unopt} - SC_{SAF-opt}}{SC_{SAF-unopt}} \right) * 100$$

In addition, we can further utilize the storage measurement information to predict the marginal cost of storage consumption for provenance collection. In our storage consumption context, we will consider the marginal cost of storage consumption as the amount of storage space that is required for adding an additional stream component to a stream processing system. This information can be used by application developers to ascertain the impact that provenance recording will have when integrated with their application. For the unoptimized stream ancestor function approach, we can derive the marginal cost of storage consumption for provenance collection from the following equation:

$$MC_{SAF-unopt} = (k + e) * m$$

For the optimized stream ancestor function approach, the marginal cost of storage consumption for provenance collection can be calculated as follows:

$$MC_{SAF-opt} = \frac{SC_{SAF-opt} - (((k + e) * m) * fs)}{is} \qquad or$$

$$MC_{SAF-opt} = (k * m)$$

Note that, the marginal cost only focuses on the cost of adding one more component (stream operation) to a stream system. Therefore, in the equation above ($MC_{SAF-opt}$), the fixed storage cost incurred by the recording of the contents of stream elements that act as the first input streams to a stream system is not used in the calculation.

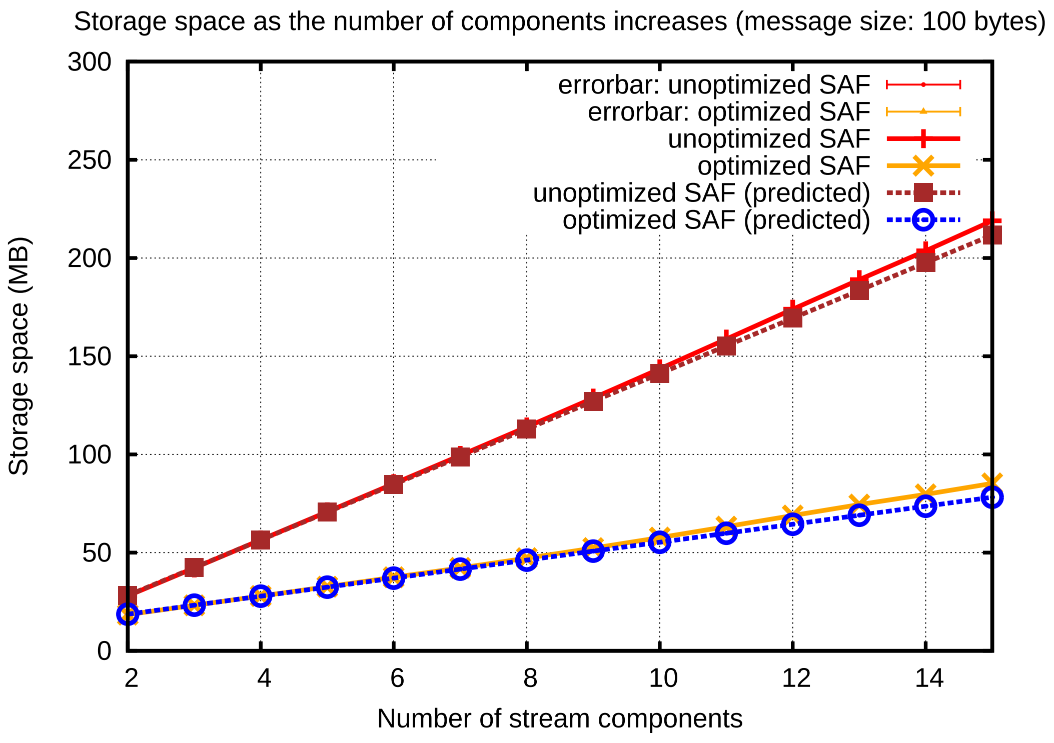


FIGURE 6.3: Provenance storage cost for 100 bytes stream events

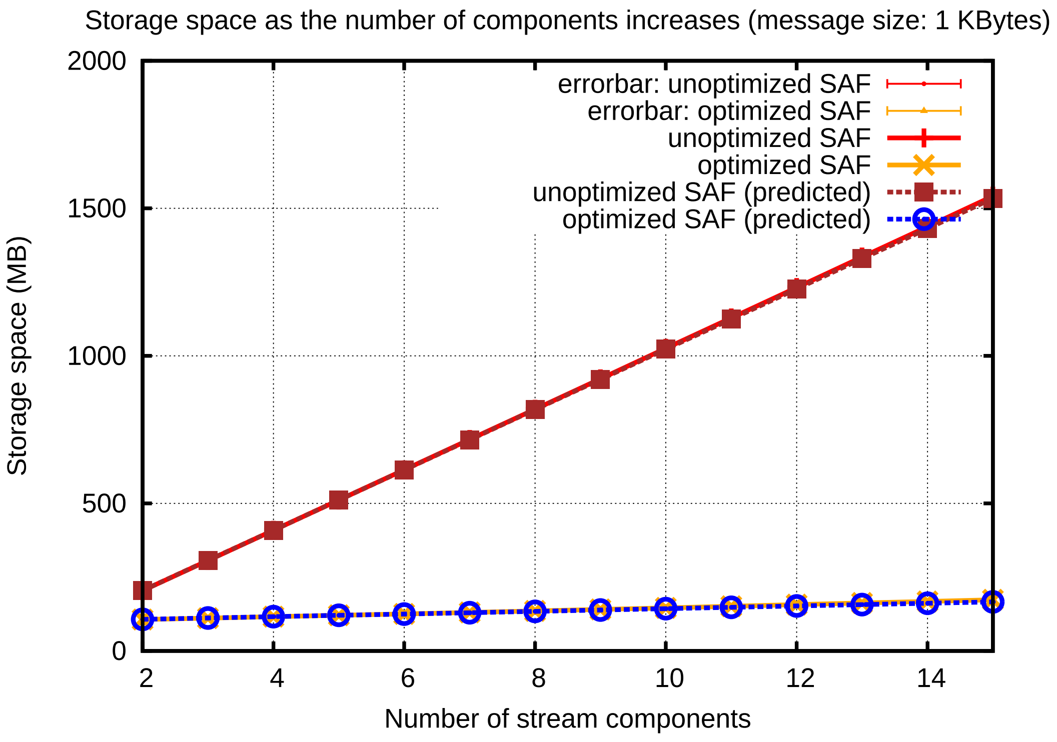


FIGURE 6.4: Provenance storage cost for 1 Kbytes stream events

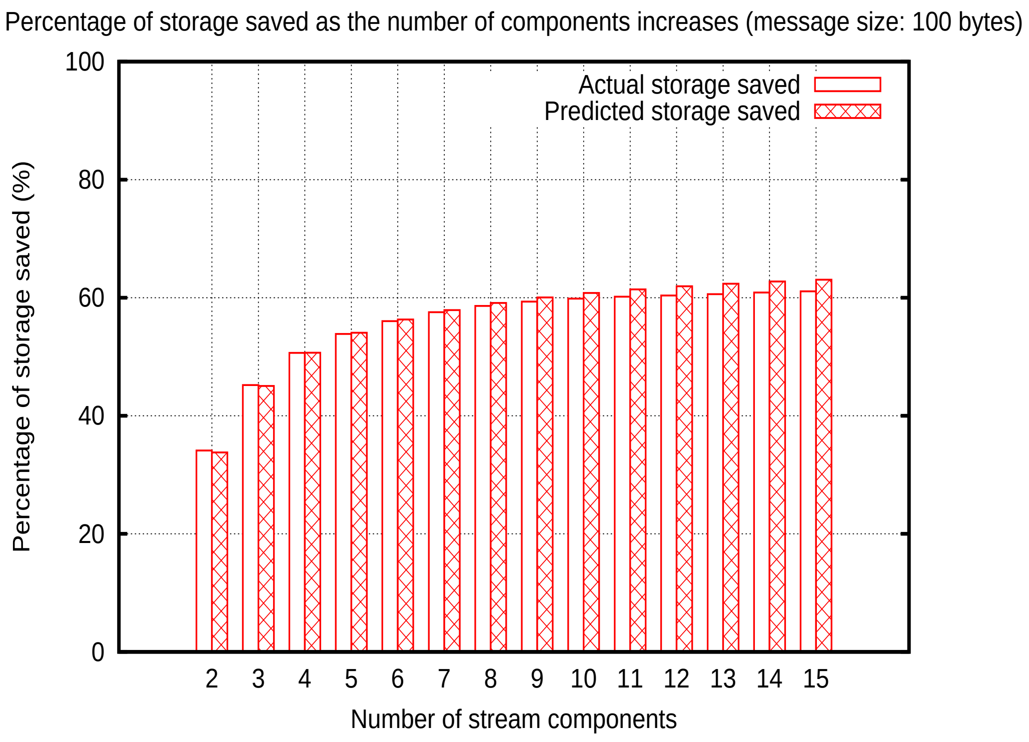


FIGURE 6.5: Percentage of storage saved for 100 bytes stream events

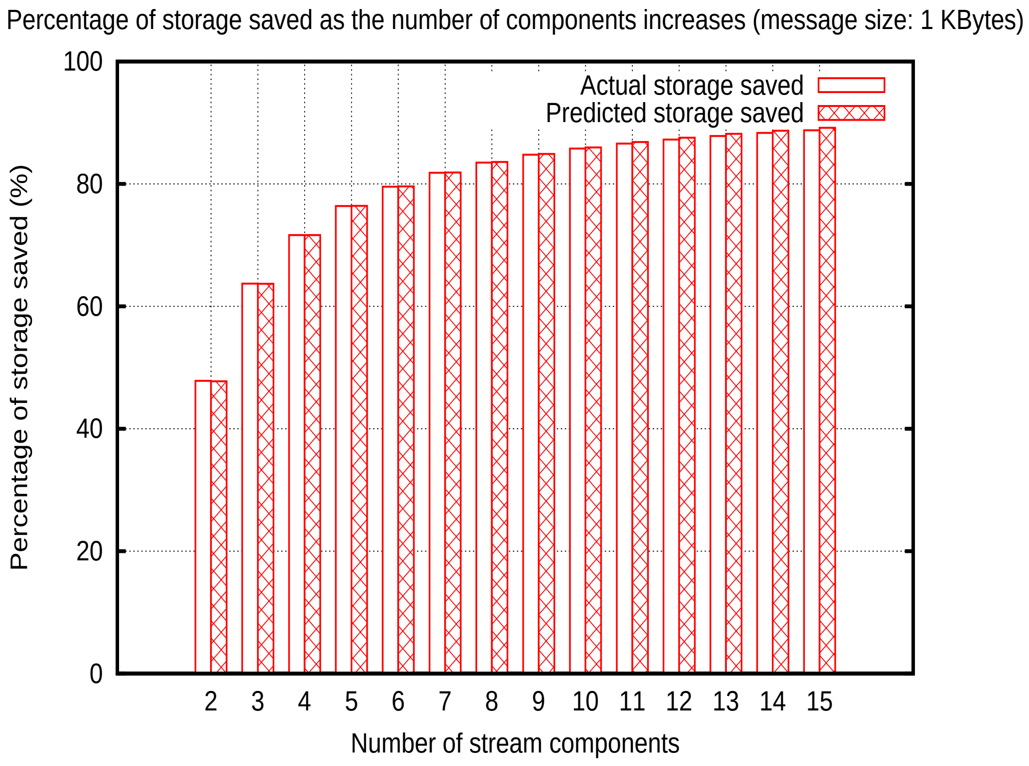


FIGURE 6.6: Percentage of storage saved for 1 Kbytes stream events

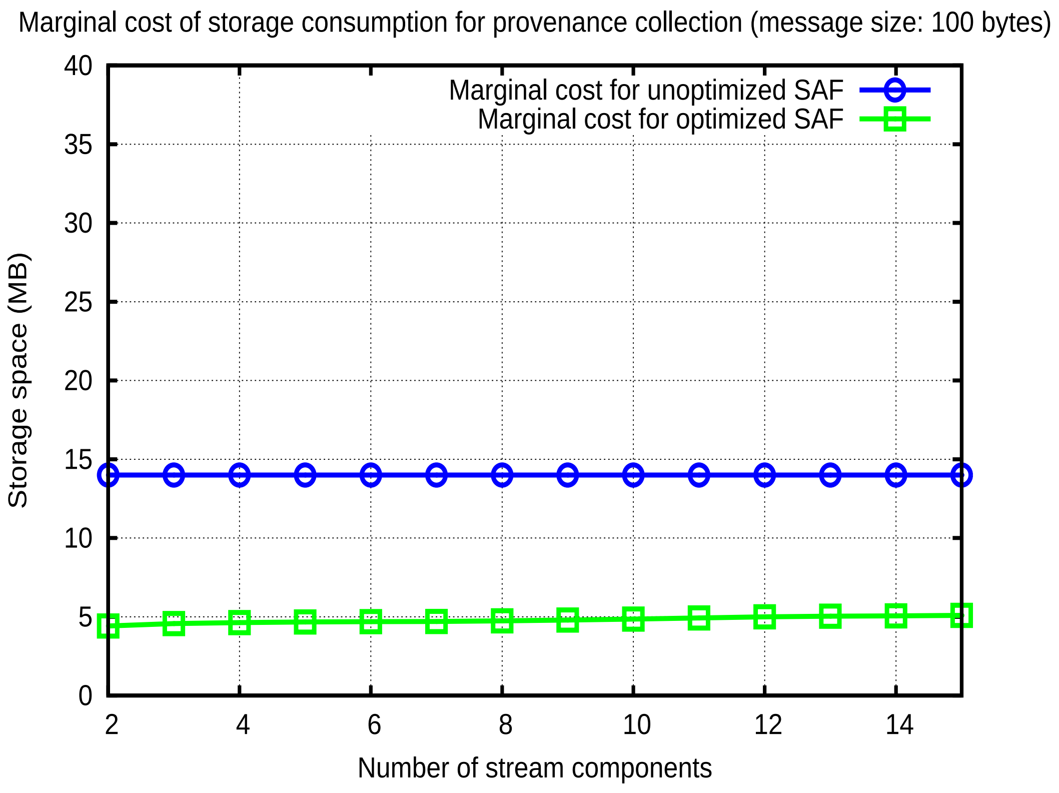


FIGURE 6.7: Marginal cost of storage consumption for 100 bytes stream events

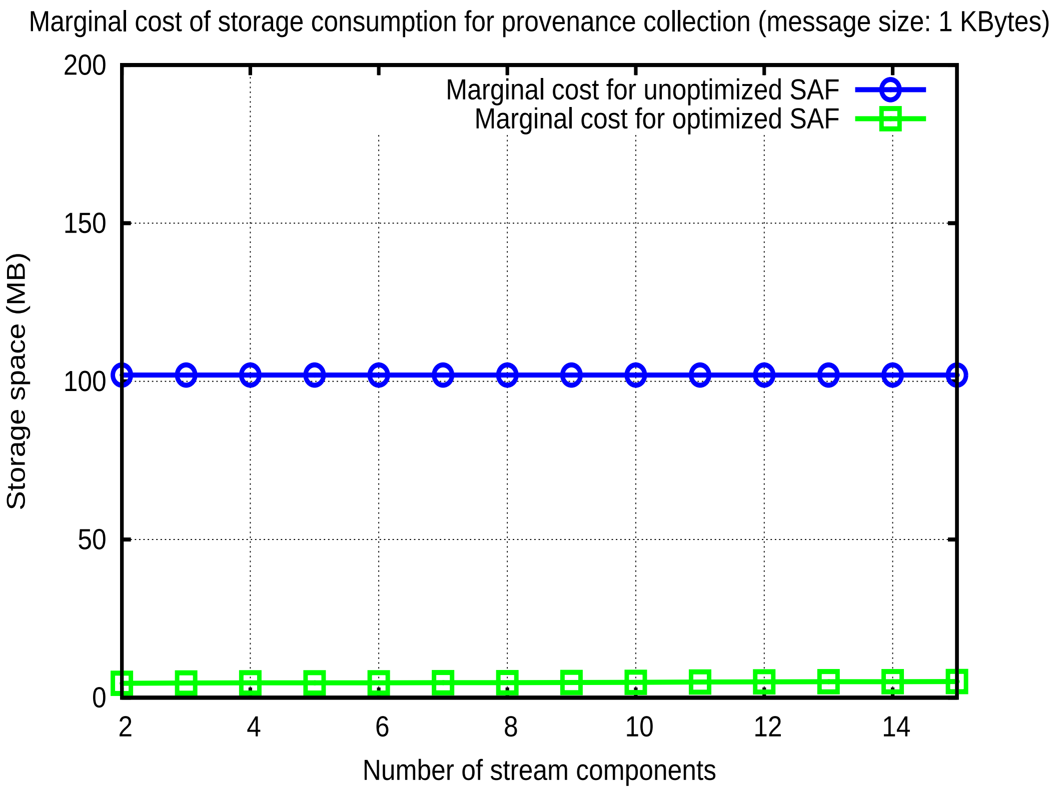


FIGURE 6.8: Marginal cost of storage consumption for 1 Kbytes stream events

Figure 6.3 and 6.4 show the storage cost needed to store provenance information for different provenance storage approaches and different payload sizes (message sizes). The storage cost presented in the both figures are an average from ten trials. We show the storage cost observed for the synthetic processing flow applying our storage reduction technique (optimized stream ancestor function - optimized SAF) and the other that does not employ the storage reduction technique (unoptimized stream ancestor function - unoptimized SAF). The predicted storage cost of both approaches derived from our storage formulas are also presented. In both figures, the storage cost of unoptimized SAF grows significantly because both event keys and event contents for every intermediate events need to be stored in a provenance store. On the other hand, compared to the optimized SAF which applies the storage reduction technique, the amount of storage consumed by the stream system is just slightly increased. This is because many event contents for every intermediate event are discarded (except the contents of stream events that are the first input to the system).

The storage saving rates shown in Figure 6.5 and 6.6 indicate that our storage reduction technique (optimized SAF) is extremely effective when dealing with large message sizes. For instance, at 10 stream components, the percentage of storage space we can save from discarding event contents of intermediate stream elements with 1 Kbytes payloads is almost 85 percent. It is much higher than the percentage of storage saved for 100 bytes payloads stream events at the same number of stream components which is about 60 percent. This finding shows that the bigger the message size that a stream processing system exploits, the greater the storage overheads can be saved by our storage reduction technique. The storage saving rates described here not only present the suitability of our storage reduction approach for dealing with large message sizes, but also demonstrate the efficiency of our storage reduction approach to reduce the storage overheads for provenance collection. For example, according to Figure 6.6, it is obvious that, at 15 stream components, the storage cost for provenance collection can be reduced by almost 90 percent by utilizing our storage reduction technique (optimized SAF). Therefore, it can be concluded that by applying our storage reduction technique, the storage overhead resulting from recording contents of stream events can be reduced significantly.

Furthermore, the marginal cost of storage consumption in Figure 6.7 and 6.8 indicates that our storage reduction solution can economize the storage cost for provenance collection when a stream processing system is scaled up. In the both figures, the marginal cost for unoptimized SAF and that for optimized SAF remain stable when the number of stream components increases. However, the fixed rate of the marginal cost for optimized SAF is considerably less than that for unoptimized SAF. Compared to unoptimized SAF, the percentage of the marginal cost reduced by the optimized SAF approach for 100 bytes payloads stream events is around 65 percent and that for 1 Kbytes payloads stream events is about 95 percent. These results show the substantial reduction in storage consumption when stream components increase. The results not only present the fixed rate of storage cost but also indicate that the marginal cost for the optimized SAF does not depend on the size of a stream event (size of event content). It is obvious that the marginal cost for optimized SAF is relatively low, about 5 MB for both 100 bytes stream events and 1 Kbytes stream events experiments (for 100,000 input stream events). Therefore, with the considerably smaller and fixed marginal cost (based on the number of input stream events), practical storage cost control can be maintained by application developers when a stream processing system needs to be scaled up.

6.4 Provenance recording impact

The impact of provenance recording on stream processing system performance is now evaluated. The purpose of this evaluation is to observe the impact of provenance recording on a stream processing system, and to understand how much our provenance solution penalizes stream processing in a controlled environment. In this evaluation, system throughput - the number of messages (stream events) processed by a stream processing system over a given interval of time - is used as our performance indicator. We compare the system throughputs obtained from the implementation of our provenance solution using different provenance processing modes to another implementation that does not process or record any provenance-related information (a stream processing system under normal processing).

The experiments were run with a stream producer submitting input stream events to our stream processing system. Similar to the storage experiments, after stream events were fed into the stream system, provenance assertions for individual stream events were recorded through the use of a provenance service. The synthetic processing flow used is a linear stream processing flow where stream components (stream operations) are chained together and each component takes input events from a previous component. In each set of experiments, we first measured the system throughput of the implementation of a stream processing system that does not record provenance information. This experiment aims to demonstrate how much is the system throughput that we can expect under normal processing (without provenance recording). Then, we measured the system throughput of a stream processing system that records provenance information for different provenance processing modes of the provenance service including: 1) "Just receive provenance assertions" (without any provenance processing by the provenance service), 2) "Store provenance assertions" into a provenance store, and 3) "Perform on-the-fly provenance queries" over provenance assertions received. This experiment aims to present the effect of provenance recording on the system throughput of a stream processing system applying our different provenance processing approaches.

To understand the impact of provenance recording when a stream processing system is scaled up, the number of stream components (stream operations) used in the experiments was increased from 2 up to 15. In addition, we note that widely varying delays in processing are common in real-life stream-based applications. The delays in processing are usually caused by high-volume data streams and complex calculation of stream processing operations. However, the processing delays, in some stream-based applications, result from the processing of stream operations that do not require intensive or complex calculation as well. Examples of these stream operations include the in-order stream processing operations [85] - operations that need to force order on their input stream elements during execution - and the blocking stream operations [55] - operations that are required to wait until all input stream elements are available for their computation. Furthermore, because we assume that our stream processing system is implemented as a distributed system where stream execution can be computed at different nodes in the system, so instead of focusing on the computational loads and the complexity of stream computation performed, we observe the impact of processing delays on system performance when provenance information is recorded. We consider the delays in stream processing as significant parameters that can represent the processing of the wide variety of stream-based applications. Four different time delays in stream processing - no delay, 1ms, 2ms and 3ms - were considered in the experiments. With the introduction of various time delays in stream processing, we can better understand the impact of provenance recording on a stream processing system's performance and we can also evaluate the effectiveness of our provenance system when dealing with real-life stream processing systems at different time delays for stream processing.

It is important to note that all components used in the provenance recording impact experiments, including all stream processing component (stream processing operations) in a stream processing system and a provenance service, were run in the same Java Virtual Machine (JVM). During the experiments, no other applications were running and using resources on the system under test. In addition, before starting each measurement, we introduced a warm-up or ramp-up period. This interval is utilized as the time for loading all necessary java classes and allocating all the necessary resources in order that the implementation of our stream provenance system can be run with full message handling capacities. System throughput information was collected and calculated after the warm-up period. Each test was run ten times, and measurements were averaged to obtain the final results.

Figure 6.9 displays the system throughputs of our implementations of stream processing systems with no time delays in processing, as a number of stream components increases. The figure shows a significant drop-off in the system throughput of the implementation that does not record provenance-related information (no provenance) from the maximum throughput (around 23,000 messages/second). Similar but significant lower trends in system throughput were observed for the other implementations that record provenance information. At the same number of stream components, the performance (system throughput) decreases more than 50 percent for all implementations compared to the

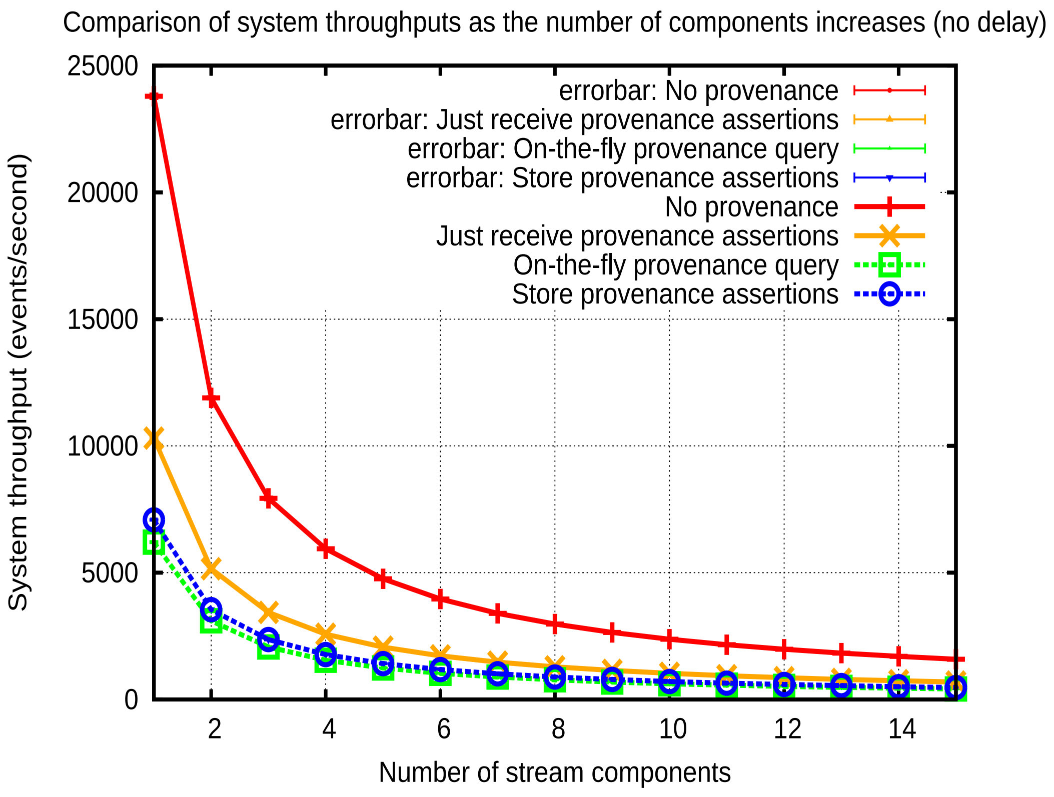


FIGURE 6.9: Comparison of system throughputs for no delay in stream processing

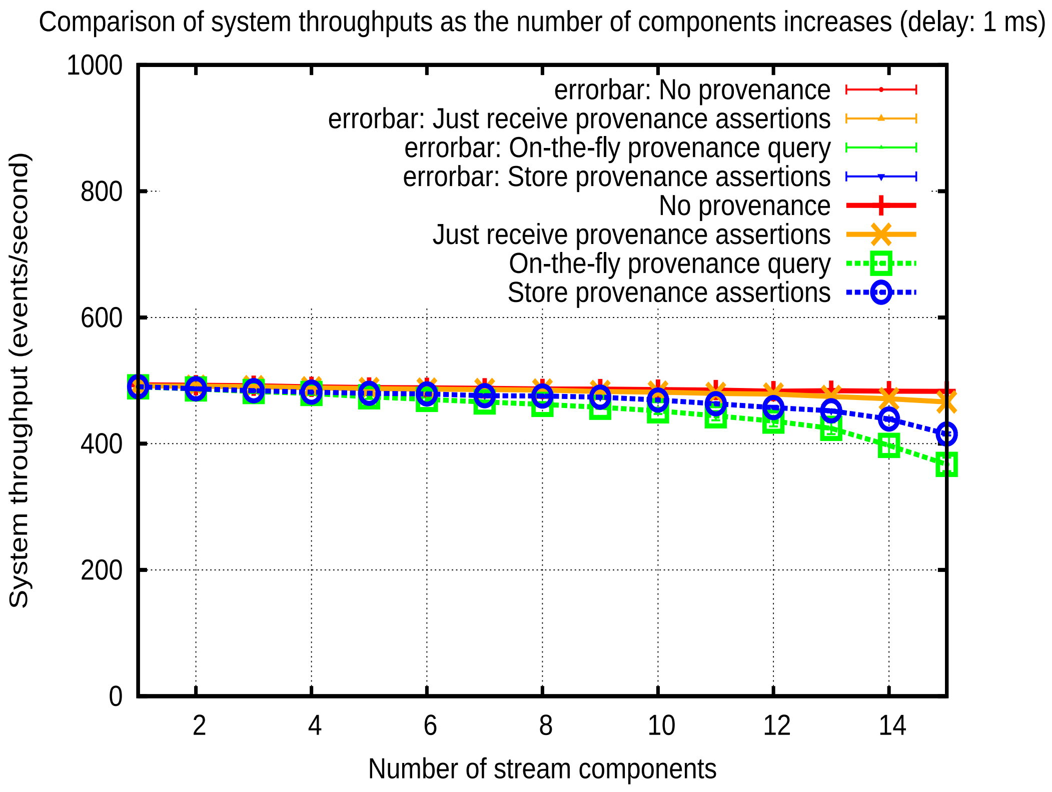


FIGURE 6.10: Comparison of system throughputs for 1 ms delay in stream processing

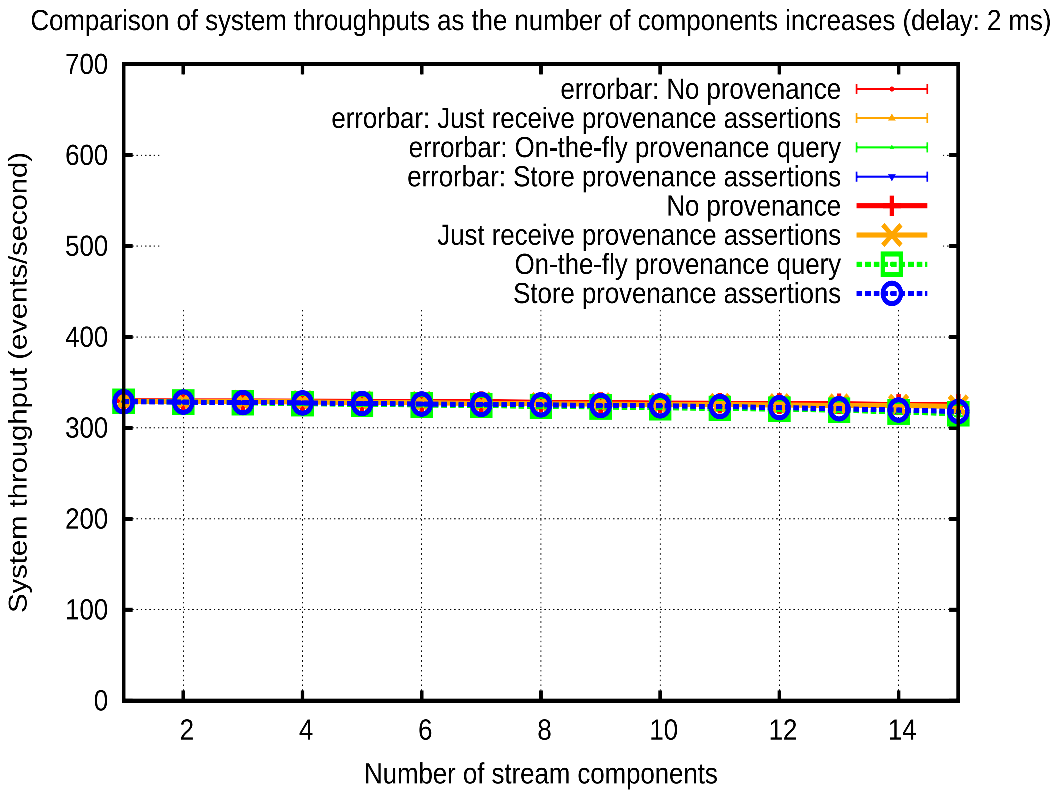


FIGURE 6.11: Comparison of system throughputs for 2 ms delay in stream processing

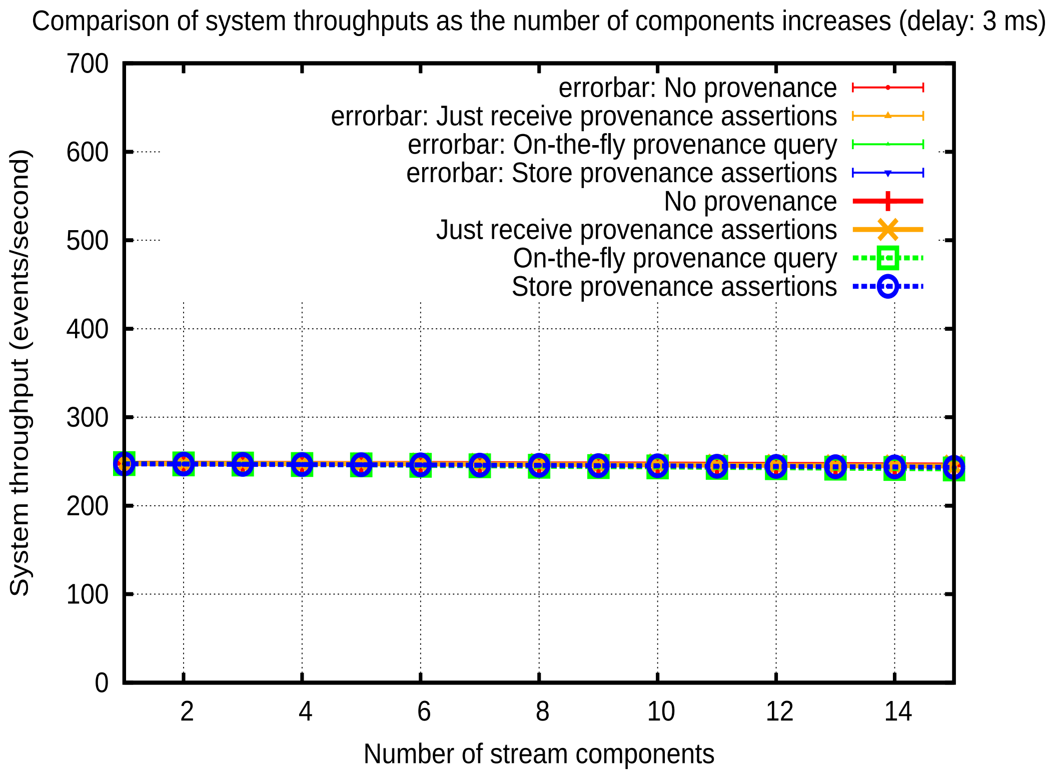


FIGURE 6.12: Comparison of system throughputs for 3 ms delay in stream processing

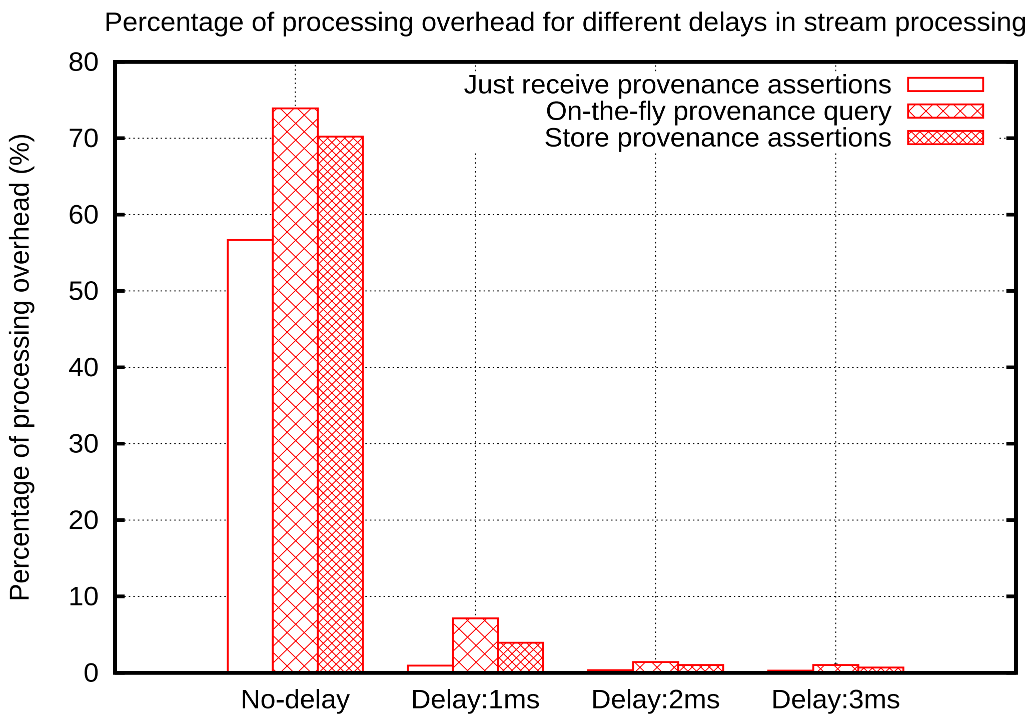


Figure 6.13: Percentage of overheads for different delays in stream processing

case of "no provenance" implementation. We determined that this degradation is due to the introduction of provenance recording functionality which doubles the number of data streams maintained by the message broker software (Apache ActiveMQ). Overall, the processing overhead of the on-the-fly provenance query implementation is slightly greater than that of the store provenance assertions. This can be explained by the characteristic of on-the-fly provenance query approach (stream-specific query approach),where provenance assertions are recorded as a stream and at the same time that incoming provenance assertions are received, provenance queries are executed continuously by a provenance service. This processing characteristic directly causes some processing overheads to the provenance system compared to the store provenance assertions approach (or the persistent provenance query mechanism that stores assertions persistently before performing queries later) which only receives provenance assertions and records them directly to a provenance store.

Figure 6.10, 6.11 and 6.12 demonstrate the system throughputs of our implementations that increase time delays for stream processing. The time delays are increased from no delay to 1ms, 2ms and 3ms respectively. In Figure 6.10, all system throughputs significantly drop from that in "no processing delay" experiment (shown in Figure 6.9) as expected due to the introduction of time delay 1ms. The overall trends of the system throughputs for our implementations using different provenance recording approaches has been changed by the introduction of time delay as well. The system throughputs of both on-the-fly provenance query and store provenance assertions implementation gradually decrease when a number of stream components increases. This degradation of the system throughputs is considerably smaller compared to the reduction of the system throughputs in “no processing delay” experiment. In addition, as shown in Figure 6.11 and 6.12, the trends of the system throughputs for all our implementations are almost flat and there are almost no significant difference between the system throughputs of “no provenance” implementation and that of both on-the-fly provenance query and store provenance assertions implementations. The more the time delay for stream processing increases the more the processing overheads for provenance recording can be reduced. As a result, we conclude that the overheads caused by our provenance solution can be greatly reduced when the time delay of a stream processing system is large.

In Figure 6.13, the percentage of the processing overheads incurred by our implementations for different provenance recording approach is summarized as the time delays for stream processing increase. For the ‘no processing delay” experiment, the average overheads when provenance information is recorded are excessively high - about 70 percent for store provenance assertions approach and nearly 75 percent for on-the-fly provenance query approach. On the other hand, when the time delays for stream processing are introduced - 1ms, 2ms and 3ms respectively -, the overheads are significantly reduced to be less than 10 percent for all our provenance recording approaches. For example, in “1ms processing delay” experiment, the percentage of overheads for store provenance assertions approach is about 4 percent and that for on-the-fly provenance query approach is nearly 7 percent, both of which are relatively low. Considering that, in real-life stream-based applications, delay time for stream processing is very common and it generally occurs in wide variety of stream-based applications. Therefore, with the experimental results, we can establish that the impact of provenance recording is relatively small or more particularly the processing of provenance recording in our provenance solution generally does not have a significant effect on the normal processing of stream processing systems. In addition, these performance figures also guarantee that our provenance solution - including both the on-the-fly provenance query and the store provenance assertions - offer reasonable and acceptable processing overheads.

6.5 Memory consumption for a provenance service

This evaluation aims to examine the effect of the provenance processing on the normal processing of a provenance service, particularly with respect to memory consumption. We compare the amount of memory consumed by the implementation of our provenance service applying on-the-fly provenance query approach to another implementation that does not perform any further processing on provenance-related information received (provenance assertions submitted by a stream processing system are just received and then discarded by the provenance service).

In our memory consumption experiments, a linear processing flow - synthetic stream processing flow where stream components are chained together and each component takes input events from a previous component - was utilized for a stream processing system. In each set of experiments, we first measured the actual memory space used by the provenance service when provenance assertions are just collected but not executed as the number of stream components increases. The result of this experiment is utilized as a performance baseline which indicates the normal amount of memory consumed by the provenance service without provenance processing. Then, we measured the actual memory space used by the provenance service that is operated on the on-the-fly provenance query mode. This experiment aims to present the effect of provenance processing on the memory consumption of the provenance service when provenance-related information is computed (performing on-the-fly provenance queries over provenance assertions).

It is important to note that the provenance service used in our memory consumption experiments was run in a separate JVM (different from the one used to run a stream system). The use of a separate JVM allows us to measure the amount of memory allocated only for the provenance service. The memory space we measured is the Java heap memory - the area of memory utilized by the JVM for dynamic memory allocation (store the new objects being created). In addition, for every run for each stream topology (each number of stream components used), a fresh JVM was used. This means that our provenance service is executed and run in a new JVM instance every time that a new measurement (a new run) for each stream topology is started. This experiment set-up is used to ensure that there is no effect from the previous run on the current measurement (no objects from the previous run that occupy Java heap spaces when the new run is started). In this evaluation, the input stream events were submitted to a stream processing system at a rate of an event per millisecond. Measurements were taken after receiving 1000 provenance assertions and measurements were averaged to obtain final results.

To investigate the change of the memory consumption of the provenance service when a stream processing system is scaled up, the number of stream components used in the experiments was increased from 2 up to 10. In addition, two different stream operations are considered: Map and Sliding time-based window operations. The reason behind the use of two stream operations is because for some stream operations (e.g. Time-based window), a number of provenance assertions has to be buffered (in memory) until it is used in the processing of on-the-fly provenance queries. This persistence of provenance assertions in memory potentially increases the amount of memory consumed by the provenance service. Therefore, by using different stream operations in our memory consumption experiments, we can demonstrate the transformation of the amount of memory consumed when some provenance assertions are buffered during the processing of on-the-fly provenance queries and we can also present how accurate is our memory prediction equation.

TABLE 6.2: Mathematical Symbols for memory prediction formulas

Symbol	**Definition**
MS	Memory space consumed for the on-the-fly provenance query mode
MC	Marginal cost of memory consumed for the on-the-fly provenance query
c	No. of stream components
EP	Memory size of additional classes for Esper stream engine
Q_{EP}	Size of an inbound queue of Esper stream engine
AS	Memory size of each provenance assertion
w	No. of provenance assertions stored in data windows (temporary buffers) of Esper engine for the on-the-fly provenance query mode

The memory consumption of a provenance service can be estimated by using a straightforward mathematical formula. We can utilize the formula to predict the amount of memory consumed by our system when provenance information is processed. Table 6.2 lists the mathematical symbols used in our memory prediction formula. For the on-the-fly provenance query, the amount of memory space consumed can be calculated as follows:

$$MS = Initial\ memory + EP + (Q_{EP} * AS) + (c * w * AS)$$

According to the memory prediction formula, *the initial memory* is defined as the amount of memory consumed by the provenance service when provenance assertions are just received but not executed (no provenance processing). Because Esper stream engine is added to the provenance service to provide straight-through processing over streams of provenance assertions supporting the on-the-fly provenance query (discussed in Section 6.1), EP - the memory size of additional classes for Esper - is a required parameter. Furthermore, due to the fact that all incoming events (provenance assertions) need to be placed in Esper's inbound queue for on-the-fly processing, the size of Esper's inbound queue Q_{EP} is another important parameter for memory calculation. The last term - $(c*w*AS)$ - describes a total number of assertions temporarily stored in the internal data buffers of Esper for each stream operations. Note that, in this formula, we particularly focus on the case that every stream operation has the same size of data windows (w). So, in the case that the difference sizes of data windows are used, the last term needs to be modified in order to accurately calculate the memory consumption. For example, in the case of two length-windows of size 10 and 20 events respectively, we can derive the memory prediction formula as follows:

$$MS = Initial\ memory + EP + (Q_{EP} * AS) + \mathbf{(10\ *\ AS)\ +\ (20\ *\ AS)}$$

In addition, we can further utilize the memory prediction formulas to predict the marginal cost of memory space consumed. In this context, the marginal cost is calculated by using the difference between the fixed memory space consumed for a single stream component and the memory space used for the actual number of stream components divided by the number of stream components increased. The marginal cost of memory space consumed is calculated by the following equations:

$$MC = \frac{MS - MS(for\ a\ single\ stream\ component)}{(c-1)}$$

Figures 6.14 and 6.15 present the actual memory space consumed by our implementation of a provenance service for the map operations and the sliding time window operations experiments respectively, as the number of stream components (stream operations) increases. The predicted memory space consumed for our on-th-fly provenance query approach is also presented. The memory size for the on-the-fly provenance query approach is slightly higher in Figure 6.14 (map operation experiment), and significantly higher in Figure 6.15 (time window operation experiment) as the number of stream components increases, compared to that for the baseline (Just receive provenance assertions). For instance, at 10 stream components, the memory size for the baseline and the on-the-fly provenance query in the map operation experiment are about 98 and 102 MB respectively (the difference of memory size is around 4 MB) compared to that in the time window operation experiment which are around 98 and 115 MB (the difference of memory size is around 17 MB). We determine that this increase of memory size is due to the processing of on-the-fly provenance queries that needs to store provenance assertions received in the memory (both in an inbound queue and in temporary data windows of a stream engine during the execution of provenance queries) and also the introduction of the Esper stream engine added to the provenance service to provide on-the-fly provenance query functionality.

In addition, considering the fact that when the type of stream operations is changed from Map operations (in Figure 6.14) to Sliding time window operations (in Figure 6.15), the memory size consumed for the on-the-fly provenance query approach goes up considerably. This can be explained because, for the time window experiment, about 1000 provenance assertions are required to be stored in the memory for each internal assertion stream of the stream engine during execution time (the size of data window used is 1 seconds and the stream rate is an event per millisecond). Furthermore, when the number of stream components increases, the number of internal assertion streams of the stream engine increases as well. The greater the number of provenance assertions needed to be stored in memory for processing, the greater the memory space consumed by our provenance solution. Therefore, from the experimental results, we can conclude that the amount of memory space consumed for the on-the-fly provenance query approach

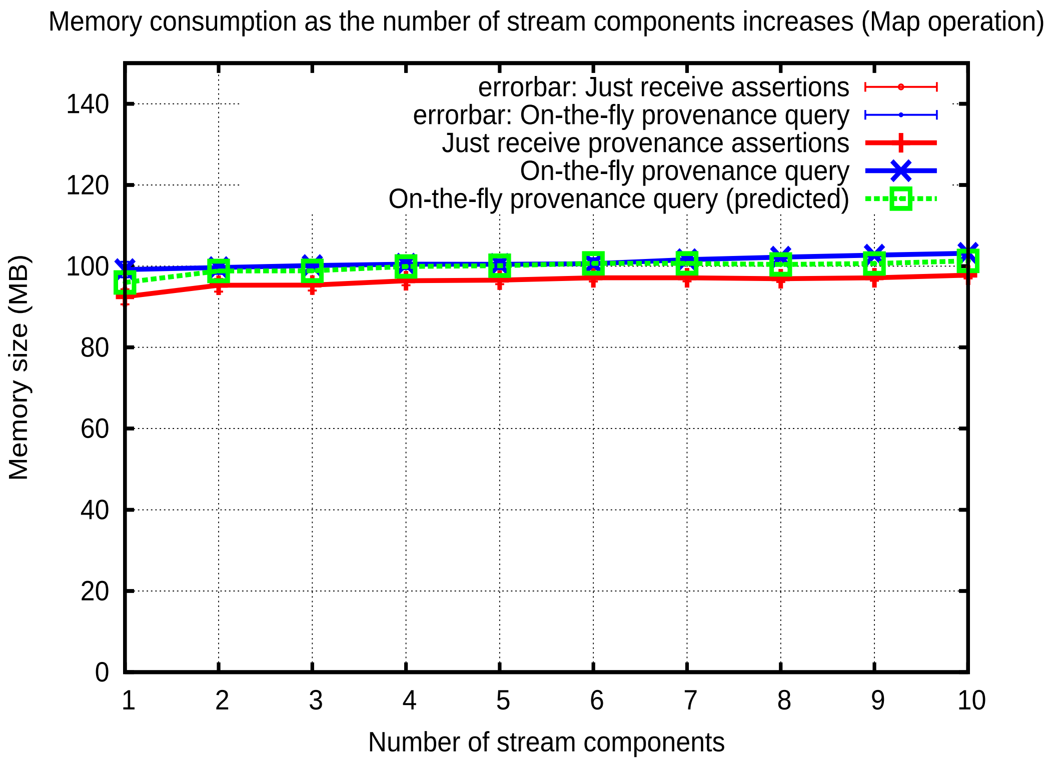


FIGURE 6.14: Memory consumption for a provenance service (Map operation)

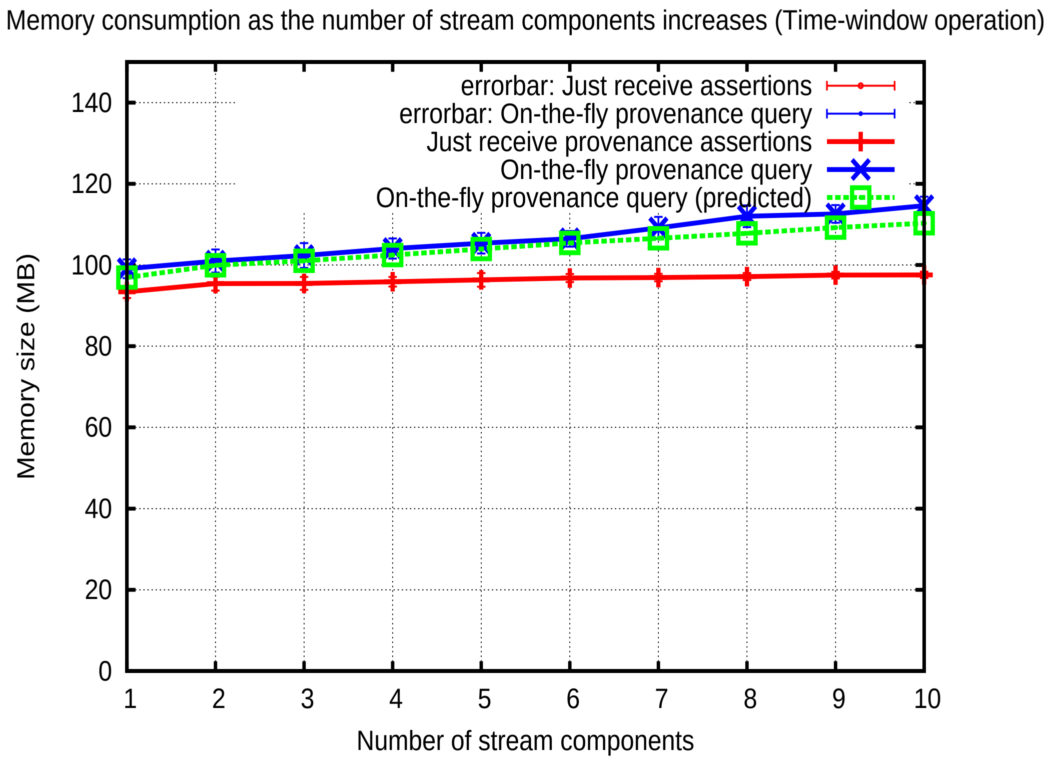


FIGURE 6.15: Memory consumption for a provenance service (Time-window operation)

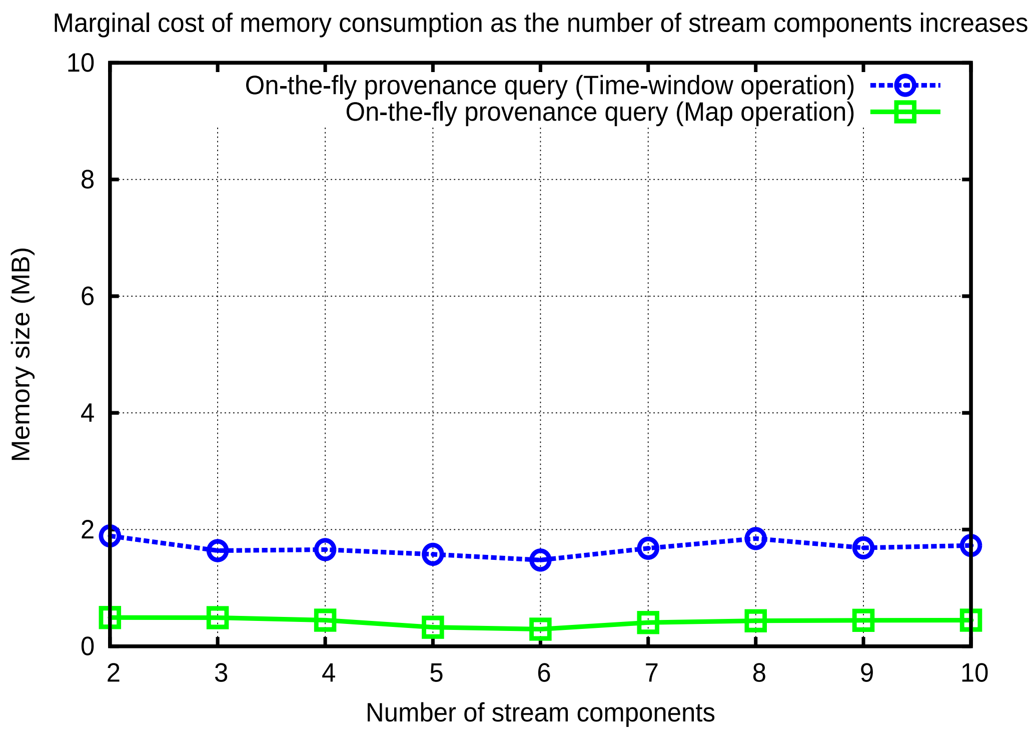


Figure 6.16: Marginal cost of memory space consumed

depends upon the types of stream operations used and the size of data windows specified for each stream operation (the size of data windows indicates the amount of assertions required to be buffered in memory).

Furthermore, as shown in Figures 6.16, the overall trend of the marginal cost of memory space consumed remains stable for both the map operation experiment and the time-window operation experiment as the number of stream components increases. The result also shows that the marginal cost is relatively low compared to the total memory space consumed. For example, the average marginal cost of memory size for the on-the-fly provenance query in the map operation experiment is slightly less than 0.5 MB and also that in the time window operation experiment is about 1.8 MB. With these experimental results, we can conclude that the memory consumption for a provenance service may vary slightly based on the types of stream operations and the size of data windows utilized in a stream processing system, but the marginal cost of memory space consumed for our provenance solution is relatively low and reasonable.

6.6 Time latency for on-the-fly query processing

We now examine the overall runtime overheads for our on-the-fly provenance query approach by measuring the time latency while a stream processing system records provenance assertions for individual stream elements and provenance query results are continuously generated as a stream by on-the-fly provenance queries. In this context, the time latency for on-the-fly provenance queries is defined as a period of time difference between a stream system produces a stream processing result and a corresponding provenance-related property is computed. This evaluation aims to establish that our on-the-fly provenance query approach offers low-latency query processing and can provide real-time or near real-time query responses.

In this experiment, we use a linear stream processing flow where stream components (stream operations) are chained together and each component takes input events from a previous component. Three different chains of stream operations are considered: Map operations, Time-windows of size 10ms and Time-windows of size 100ms. The reason behind the use of different stream operations is because for some stream operations (e.g. Time-based window operation), more than one provenance-related properties have to be calculated in the processing of provenance queries. This may cause some runtime overheads and potentially results in delayed query results. Furthermore, because the size of data windows indicates the amount of provenance assertions required to be buffered in memory, therefore we consider the size of data windows as another factor that potentially has an impact on time latency. To understand the overhead cost of our on-the-fly query solution when a stream processing system is scaled up, we increased the number of stream components used in the experiment from 2 up to 10. With the use of three different chains of stream operations and the increase of the number of stream components utilized in the experiment, we can better understand the performance characteristics of our on-the-fly query approach and we can also evaluate the effectiveness of the on-the-fly provenance query when dealing with difference scales of stream processing systems.

In order to obtain the time latency for each stream processing result, we first measured the time that a stream processing system produces an output. Then, we compared it with the time that the corresponding provenance-related property is generated by our provenance service. For each test, 100,000 input stream events were submitted to a stream processing system (at a rate of an event per millisecond). Measurements were averaged to obtain the final results (average time latency). Note that, all components used in the time latency evaluation, including all stream component in a stream system and a provenance service, were run in the same Java Virtual Machine (JVM). During the experiment, no other applications were running and using resources on the system under test. We measured the time latency for each stream processing result after the warm-up period (to ensure that all necessary classes and resources are loaded).

Figure 6.17 displays the average time latency of the implementation of our on-the-fly provenance query solution, as a number of stream components increases. We show the time latency observed for the chain of map operations, the chain of time-windows of size 10 ms and the chain of time-windows of size 100 ms. In the figure, as the number of stream components increases, the time latency for the chain of map operations remains stable (at around 0.3 ms). On the other hand, the time latency for both chains of time-

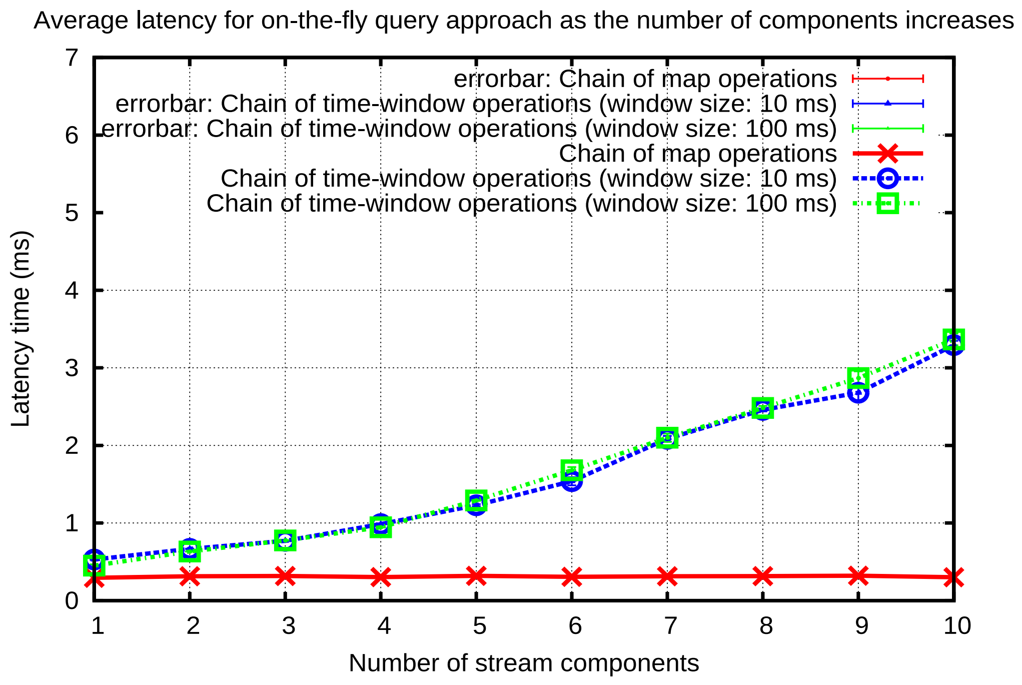


FIGURE 6.17: Average time latency for on-the-fly provenance query approach

window operations increases significantly. This can be explained by the characteristic of on-the-fly provenance query processing that for windowed operations such as time-window and length-window operations, internal data buffers of a stream engine (Esper) are used to store intermediate provenance assertions during query execution. This results in an increase of the average delay time in query processing. In addition, as shown in the latency graph, there are no significant differences between both chains of time-window operations in time latency for on-the-fly query processing. These results indicate that the increase of the size of time windows or more particularly the amount of provenance assertions stored in data buffers does not have a significant impact on time latency. From the latency graph, the average time latency increased per additional stream component is about 0.3 ms. Therefore, with a relatively low latency per additional component, we can guarantee that our on-the-fly query solution offers low-latency processing, which is critical for stream systems, and we can also establish that our solution can provide provenance query results in real-time or near real-time.

6.7 Analysis

Based on our experimental evaluation, we offer a set of recommendations for application developers to use when integrating our provenance solution with their applications. These recommendations are presented in terms of trade-offs by which developers can decide what costs are worth suffering for what benefits and which mechanism is the most suitable for their applications.

6.7.1 Replay execution and multiple queries vs. storage space

The ability to reproduce stream events is very important and useful for stream processing systems. It allows users to replay stream execution with new input in order to obtain precise and up-to-date stream processing results. It also allows the original results produced by stream processing systems to be validated, so that users can have confidence and trust in the results. The ability to perform provenance queries multiple times over persistent provenance information is another significant function for stream provenance systems. This function allows users to perform a variety of provenance queries to answer various types of provenance questions and also to track the provenance of stream elements at whatever time users need. However, to support these two important functions, provenance information needs to be recorded in a long-term persistent storage (provenance store). As presented in several studies on provenance tracking in stream processing systems [134, 96, 72], the persistence of the provenance information of high volume stream events potentially results in a storage burden problem. Therefore, application developers face a trade-off between the capability to replay stream execution and perform provenance queries multiple times, and the amount of storage space consumed for recording provenance information.

From our experimental evaluation, as shown in Figures 6.3 and 6.4, the storage space consumed by the implementation of our provenance solution for provenance recording can be significantly reduced by applying the storage reduction mechanism (optimized stream ancestor function). Our storage reduction mechanism offers not only the reduction of storage overheads but also a constant storage cost when a stream processing system is scaled up. Furthermore, application developers can estimate the physical storage space used for their stream provenance systems during the system design process by using our storage prediction formulas. Therefore, for the stream-based applications that require the replay execution and multiple provenance query functionalities, they should exploit the store provenance assertions approach with the optimized stream ancestor function mechanism (the storage reduction technique).

6.7.2 Storage space and real-time response vs. processing overheads and memory space

As discussed by many studies [120, 116, 70, 96], the practical challenge pertaining to the persistence of stream elements is one of the most important issues which needs to be carefully considered in the design of stream processing systems. In some cases, a high volume of streaming events with a large size of content data need to be processed in real-time and continuously by stream processing applications. This makes the persistence of provenance information of stream elements, which generally requires an *immutable* characteristic (provenance information that has been previously recorded will not be overwritten, deleted or modified), unfeasible. Another significant challenge for stream processing applications is to process high-volume streaming data in real-time and deliver instantaneous responses. This challenge is applied to provenance systems for streams as well, due to the fact that in many cases, such as decision support systems, traceability of streaming data in real-time and instantaneous provenance query results are necessary in order to support mission critical operations such as disaster recovery.

To address the challenges mentioned previously, provenance assertions are required to be processed as they fly by (on-the-fly query processing) without any requirement to store them. After that, provenance query results are generated instantaneously and sent to client applications that register to receive the results as a stream. This concept we describe is the key idea that underpins our on-the-fly provenance query approach. However, the more intensive and complex the on-the-fly query processing, the greater the impact on performance. In this context, application developers face a trade-off between the capability to perform on-the-fly processing and provide instantaneous query results without the requirement to store provenance information persistently, and the processing overheads and the amount of memory space consumed resulting from the use of this capability.

From our experimental evaluation, as shown in Figure 6.13 (throughput experiments), the average processing overheads for the on-the-fly provenance query approach is slightly higher than that for the other approach - the store provenance assertions approach. Likewise in the memory usage experiments, as presented in Figure 6.14 and 6.15, the amount of memory consumed for the on-the-fly provenance query approach increases continuously when a stream processing system is scaled up. This is the real evidence that indicates how much the processing overheads resulting from the on-the-fly provenance query approach are compared to the other our approach. Our recommendation on this issue is that, for stream processing applications where storage space is very limited or where the applications require real-time and instantaneous provenance query responses, they should exploit the on-the-fly provenance query approach. Furthermore, in the design phase of application development, the types of stream operations used and the size of data windows utilized in a stream processing system need to be carefully designed in order that the amount of memory consumed can be strictly controlled.

6.7.3 Throughput vs. processing overheads

As shown in Figure 6.9, in the system throughput experiments with no time delays for processing, a significant drop-off in system throughput is presented when provenance information is recorded for both the store provenance assertions and the on-the-fly provenance query approaches. The trend of system throughput also continues to decrease as the number of stream components (stream operations) increases. This experimental result indicates the significant impact of provenance recording on the performance of stream processing systems under normal processing. However, from Figures 6.10, 6.11 and 6.12, as time delay for processing increases, the impact of provenance recording (processing overheads) drops significantly. The average processing overheads are considerably reduced to be less than 10 percent for all our provenance processing approaches compared to the case of "no provenance" implementation (7 percent for the on-the-fly query approach and 4 percent for the store provenance assertions approach). Therefore, there exists a trade-off between maximizing system throughput and minimizing processing overheads resulting from recording provenance information.

Based on our experimental evaluation, our provenance solution (both the store provenance assertions and the on-the-fly provenance query approaches) is more suitable for stream processing applications that process slightly low-rate data streams (e.g. greater than 1ms per event used in our experiments) due to the fact that the impact of provenance recording is minimal. However, in the case that it is required to apply our provenance solution to stream applications that process extremely high-rate data streams, the number of stream operations used in a stream processing system has to be carefully designed by application developers in order that the processing overheads resulting from recording provenance information can be controlled.

6.8 Summary

We started this chapter by presenting the implementation design of our stream provenance system which was used as a test system for our experimental evaluation. The design of the provenance service was detailed in order to give a clear understanding of how the implementation of our stream provenance system works, what the responsibilities of each internal component are and how the provenance service operates for each different provenance processing mode. We also discussed the technologies used in our implementation and the reasons behind their selection.

After describing the design and the experimental set-up in detail, four different sets of performance evaluation experiments, including the storage overheads for provenance collection, the provenance recording impact (system throughput), the memory consumption for a provenance service and the time latency for on-the-fly query processing, were presented. The storage overheads experiments showed that the storage overheads resulting from recording provenance information are reduced significantly when applying our storage reduction technique (optimized SAF). The results of the storage overheads experiments also indicate that our provenance solution offers considerably small and fixed marginal costs based on the number of input stream events when a stream system is scaled up.

In the provenance recording impact experiments, we have demonstrated that recording provenance information for individual stream elements, which we refer to as fine-grained provenance recording, can be done with reasonable processing overheads. The processing overheads caused by our provenance solution (both the store provenance assertions and the on-the-fly provenance query approaches) can be greatly reduced when time delay for stream processing increases, and the average overheads are significantly reduced to be less than 10 percent for all our provenance processing approaches (7 percent for the on-the-fly query approach and 4 percent for the store provenance assertions approach in"1ms processing delay" experiment). Furthermore, according to the effect of time delay for processing on system throughput, we can conclude that our provenance solution is more suitable for stream processing applications that process on slightly low-rate data streams.

In addition, in the memory consumption experiments, we demonstrated that the amount of memory space consumed by the on-the-fly provenance query approach depends on the types of stream operations used and the size of data windows specified for each stream operation. This is due to the fact that these implementation choices indicate the number of provenance assertions required to be buffered in memory. We also demonstrated that the marginal cost of memory consumption for our provenance solution is relatively small compared to the total memory consumed. Finally, from the results of the time latency experiment, we can guarantee that our on-the-fly provenance query solution offers low-latency processing and support real-time or near real-time responses (the average time latency per additional component is about 0.3 ms).

Overall, the experimental evaluation demonstrated that our conceptual design, including our stream provenance model, provenance architecture, fine-grained provenance query and on-the-fly provenance query mechanism, enables the fine-grained provenance problem in stream processing systems to be addressed with acceptable performance and reasonable overheads.

Chapter 7

Conclusion

The main objective of this dissertation is to address provenance tracking in stream processing systems. Three requirement use-cases have been identified in order to present provenance problems which are commonly found in this kind of context. The first use-case is pertaining to the need for stream systems to precisely trace individual stream events in order to validate stream processing results. The second use-case concerns the need for stream systems to reproduce stream events in order to handle stream imperfections and validate the original results produced in this kind of system. The third use-case is the need for stream processing systems to perform provenance tracking on-the-fly in order to provide real-time or near real-time responses.

In our literature review we have shown that there are no existing provenance collection techniques and data models for streams which can support our requirements. Several solutions to provenance tracking in the context of stream provenance systems focus on coarse-grained stream provenance which identifies streams or sets of stream elements and stream processing units as the smallest unit for which provenance information is collected. However, the level of granularity for capturing provenance information in these solutions is coarse and it is not detailed enough to satisfy our requirements. Although some recent stream provenance solutions provide the ability to express data dependencies for individual stream events, they still have some limitations, particularly concerning a storage burden problem resulting from the persistence of high volume stream events. In addition, another important limitation is that these solutions also fail to accurately identify all individual stream elements used in the production of a stream processing result. To deal with our use-case requirements, a fine-grained provenance solution which precisely captures provenance of every individual stream element is required.

In this dissertation, we have shown that through the use of our stream provenance model with a reverse mapping method (stream ancestor function) which precisely captures data dependencies for every individual stream elements, the problem of fine-grained provenance tracking can be solved. We have also illustrated that, by applying our provenance query mechanism for streams and our replay execution method, stream reproduction functionality for stream processing systems can be facilitated. In addition, our stream provenance solution is extended to support a stream-specific query mechanism (on-the-fly provenance queries). The use of this query mechanism enables provenance queries to be created automatically and computed continuously over streams of provenance assertions, without requiring the assertions to be stored persistently. Thus, the significant problem pertaining to the persistence of high volume stream events which potentially results in a storage burden problem can be eliminated.

We now revisit the key contributions of this dissertation.

7.1 Contributions

7.1.1 A stream provenance model and stream ancestor functions

This first contribution is *a stream provenance model* that defines the structure and the key elements of provenance representation in stream processing systems and *a set of primitive stream ancestor functions* that is utilized as a crucial mechanism to explicitly express dependency relationships among individual stream elements.

A provenance data model for streams defines the structure and the key elements of provenance representation for streams - provenance-related information to be stored in a provenance store in order that the provenance of stream processing results can be retrieved. The key elements of stream provenance representation include provenance assertions (dynamic information) and auxiliary information (static information e.g. stream topology information, configuration parameters and metadata). Because of the generic nature of the provenance data model, this offers the vital advantage of allowing our stream provenance model to be applied in a wide range of stream processing applications for a variety of domains. To support the stream provenance model, another important mechanism - a reverse mapping method namely stream ancestor function - has been proposed. The stream ancestor function is defined for each stream operation in a stream processing system to express dependency relationships between input and output events of stream operations. To deal with the storage problem potentially resulting from the persistence of all intermediate stream elements, the original stream ancestor function is extended by introducing the optimized version of stream ancestor functions that can be used to reduce storage consumption by recording only necessary information. With the composition of all stream ancestor functions in a stream processing system, the complete provenance of a stream processing result can be captured.

The novelty of this contribution is in its ability to precisely identify all stream elements involved in the production of a stream processing result - fine-grained stream provenance tracking - and the combination of a compact provenance representation and a storage reduction mechanism (optimized stream ancestor functions) that can significantly reduce the amount of storage consumed for provenance collection and eliminate the requirement to store every intermediate stream element. We have concretely demonstrated how dependency relationships between input and output stream elements for each type of stream operations can be determined and how our reverse mapping mechanism can accurately identify input elements used in the production of a particular output element by presenting the programmatic specifications of stream ancestor functions as defined in Section 3.4. In addition, as presented in the experimental evaluation (Chapter 6), we have shown that the stream ancestor functions and the provenance model can be concretely implemented and the performance of the implementation is acceptable. The results of the storage consumption experiments (presented in Section 6.3) demonstrate the evidence that substantial storage savings can be achieved, and considerably small and constant marginal cost of storage consumption based on the number of input stream events can be offered when a stream system is scaled up by utilizing our storage reduction mechanism (about 5 MB per additional stream component for 100,000 input stream events). The results of the provenance recording impact experiments (presented in Section 6.4) indicate a 4% overhead for our store provenance assertions approach. Therefore, from these experimental results, we have established that our stream provenance solution can effectively address the storage problem caused by the persistence of all intermediate stream elements and our provenance solution does not have a significant effect on the normal processing of stream systems.

7.1.2 Provenance query and replay execution methods

The second contribution is a provenance query method for streams that underpins our first contribution - fine-grained stream provenance model. The provenance query method exploits stream ancestor functions in order to perform fine-grained provenance tracking in stream processing systems. We have developed a novel provenance query algorithm based on the idea of function composition that combines simple functions to build more complicated functions. As described in Section 4.1, the key concept of our provenance query algorithm is that for a given stream processing result, the query algorithm dynamically composes stream ancestor functions for all stream operations in the processing flow of a stream processing system together (like traversing a graph in reverse order on a node by node basis) in order to resolve data dependencies among intermediate stream elements and finally provide the complete provenance of that stream processing result.

To establish that the results produced by the provenance query method are perfectly accurate and to provide a mechanism for validating stream processing results, we have developed a replay execution method for streams that defines a mechanism by which to perform stream reproduction by using provenance query results and provenance-related information stored in a provenance store. We have shown the generic algorithm for stream replay execution as described in Section 4.2. The replay execution algorithm dynamically composes the stream operations involved in the production of a particular output by utilizing provenance-related information recorded (e.g. provenance assertions and stream topology information) and finally produces replay execution outputs which are processing results originally produced by a stream processing system. To demonstrate how the provenance query method works in practice, an example case study for provenance queries is presented, as shown in Section 4.3. With the presentation of this straightforward case study, we have shown that the provenance query method can be concretely implemented, the provenance of each individual stream element can be traced by using our provenance query mechanism and also the actual query results can be validated by using our replay execution method.

7.1.3 Stream-specific provenance query

The third contribution is a stream-specific provenance query mechanism that enables provenance queries to be computed on-the-fly without requiring provenance assertions to be stored persistently. As presented in Section 5.1, the key concept of the stream-specific provenance query is inspired by the idea of property propagation where the provenance of each individual stream element can be obtained by propagating provenance-related properties from the source (the first input stream) to the destination (the final output stream) inside the provenance service. We have discussed the important characteristics of the stream-specific provenance query and also defined a specification of a new version of the stream ancestor functions that are designed specifically to work with the stream-specific query mechanism. To ascertain that the design of our stream-specific provenance query can be applied in practical stream-based applications, we have presented an example case study for on-the-fly provenance queries, as shown in Section 5.4. In this case study, the actual processing of data accumulation used for properties propagation is demonstrated in detail which allows us to assure that the results of on-the-fly provenance queries conducted over streams of provenance assertions are accurate.

The use of a stream-specific query mechanism offers two major advantages. Firstly, because provenance assertions for individual stream elements are not required to be stored persistently, the storage problem caused by storing provenance information for high-volume stream events can be solved. Secondly, as the query results are generated as streams, this allows for provenance systems to offer real-time or instantaneous provenance query results to their users. Our experimental results derived from the provenance recording impact study (Section 6.4) indicate that the stream-specific query approach offers relatively low and reasonable processing overheads (there is a 7% overhead for stream-specific query approach compared to the overhead of "no-provenance" implementation). This guarantees that the stream-specific query mechanism does not have a significant effect on the performance of stream processing systems under normal processing. The results of the latency experiment (Section 6.6) indicate that our stream-specific query solution offers low-latency processing (the average time latency per additional component is about 0.3 ms). As a result, we can establish that our stream-specific query can provide provenance query results in real-time or near real-time. In addition, as shown in the memory consumption experiments (Section 6.5), the experimental results show that the amount of memory consumed for the stream-specific query approach is based on the types of stream operations used and the size of data windows utilized in a stream processing system. Therefore, we can assure that the memory consumption for our stream-specific query approach can be practically controlled by developers during application design phase.

7.2 Future work

The fine-grained stream provenance tracking approach we have outlined in this dissertation is centered around a generic fine-grained provenance model and a reverse mapping method (stream ancestor functions) which provide fundamental concepts of how provenance of each individual stream processing result can be captured in stream processing environments and the structure of information required to be stored in order to support the concept. Based on the provenance model, enhanced mechanisms, including a storage reduction technique (optimized stream ancestor functions), provenance query and replay execution methods and a stream-specific query mechanism, are developed which extend the fundamental concept to deal with practical requirements related to the unique characteristics of data streams. Our experimental evaluation indicates that the novel contributions of this dissertation we have outlined are vitally important and they enable the problem of provenance tracking in stream processing systems to be addressed effectively. However, we also believe that there is a significant amount of research that remains to be done in the area of provenance tracking in stream processing systems. Therefore, in future, this dissertation work can be extended in the following directions.

7.2.1 Interoperability with stream optimization techniques

In some specific applications (e.g. battlefield monitoring systems [28]), a high volume of data streams with rapid arrival rates are required to be processed in real-time and continuously by stream processing systems, and outputs of stream processing are produced based on Quality-of-service (QOS) specifications. During execution time, the data rates can significantly increase and potentially exceed system capacity. This situation results in stream processing systems becoming overloaded, the system latency increasing rapidly, and thus stream systems probably fail to satisfy QOS requirements. Several stream optimization techniques are generally provided by stream processing systems to deal with such overload situations. One of the most common techniques is "load shedding" [126, 11] - the process of discarding some fraction of input stream elements in order to enable stream processing systems to continue to provide up-to-date stream processing results and satisfy given real-time constraints. Another example of common optimization techniques is a shared query execution strategy [68, 91] where a number of continuous stream queries shares execution of data windows together in order to avoid system degradation when a system is overloaded.

Our work presented in this dissertation has primarily focused on the ability of stream processing systems to capture the information necessary in order to precisely trace the provenance of individual stream elements. Particularly, our proposed technique requires that the provenance assertions of full input streams have to be recorded (either recording as stream or recording in a persistent storage) in order that the provenance of stream processing results can be determined. However, it would be a difficult task for our stream provenance solution to work with the stream optimization techniques mentioned above because, as described for the load shedding technique, some input stream elements for some stream operations are probably discarded. Therefore, it is still an open question as to the appropriate or extended mechanism that enables our fine-grained stream provenance solution to inter-operate with such stream optimization techniques for time-critical stream applications. The extended mechanism should enable our stream provenance solution to function alongside such stream optimization techniques with a well-understood effect on the accuracy of provenance query answers.

7.2.2 Integration with stream processing engine architectures

Extensive research in stream processing engines has been done in the area of data stream management in order to support the emergence of sensor technologies and real-time monitoring applications. Several stream processing engines, including Aurora [28], TelegraphCQ [29], STREAM [8] and Borealis [1], have been developed for research purposes. These stream processing engines, particularly their system architectures, usually consist of different types of processing components or sub-systems depending on the model of data streams and query languages used. Besides stream processing engines developed for research purposes, there is considerable interest in the development of stream processing engines for industrial purposes as well. Examples of industrial stream processing engines include Esper [46] and StreamBase [121]. The industrial stream processing engines provide not only fundamental facilities and architectural components required to process data streams in timely and continuous fashion, but also enhanced architectural components (e.g. high availability modules and security and privacy support components) that can efficiently deal with large-scale and distributed requirements of enterprise systems.

In this dissertation, our work has mainly focused on investigating a novel fine-grained provenance tracking method for streams, and a provenance model that allow the provenance of individual stream elements to be obtained. We have also specified a provenance architecture for stream systems to describe the structure of the stream provenance system, system components and interactions between each component. However, our provenance architecture for stream systems exists as a set of separate components from stream processing engine architecture. Therefore, another open question still remains as to how our stream provenance solution can be integrated into the holistic view of stream processing engine architecture considering the various architectures of stream processing engines designed for both research and industrial purposes. From our point of view, this integration of our provenance solution into stream engine architecture is an important step forward for practical and industrial use of stream provenance systems. It is very important that the provenance-specific components for our stream provenance solution need to be seamlessly integrated, co-exist and interchanged with stream processing engines' architectural components (e.g. Query processor and High availability modules). In addition, the question pertaining to where the provenance-specific components should be fitted into the stream processing engine architectures is another important issue that needs to be carefully considered as this will probably affect the overall performance of stream processing engines.

7.2.3 Integration with the W3C provenance standard

As the need to understand and manage provenance in computer systems is growing rapidly, the ability to share and exchange provenance information among disparate systems will need to be provided. This raises an interesting challenge in terms of the interoperability of provenance systems. To address the challenge, the World Wide Web Consortium (W3C) provenance working group has developed the provenance data model (PROV) [129] as a standard mechanism for exchanging provenance information among systems. The introduction of PROV allows existing provenance systems to export their provenance information into such a standard provenance model and then other systems that need to utilize the provenance information can import and query over it.

In this dissertation, work has focused primarily on the design and the implementation of a provenance model that can support fine-grained provenance tracking in stream processing systems. Therefore, another future research issue is how to integrate or more particularly translate our fine-grained provenance model for streams into the W3C standard provenance model - PROV. The integration with this exchange provenance model will allow our stream provenance system to seamlessly share and exchange provenance information with other systems and thus ensure interoperability of our fine-grained stream provenance solution.

Appendix A

Utility functions for the replay execution algorithm

```
(* fn : ''a-> ''a list -> bool *)
fun member e [ ] = false
| member e (e1 :: l) =
            (e = e1) orelse member e l;


(* fn : ''a -> (''a * 'b) list -> 'b *)
fun assoc sid ((id, func) :: opList) =
        if (sid = id) then func
        else assoc sid opList;


(* fn : ''a list -> (''a * 'b) list -> bool *)
fun containsKeys [ ] HM = true
| containsKeys (k :: KL) HM =
   if containsKeys_1 k HM then
      containsKeysKLHM
   else false

(*fn :'' a-> (''a *' b)list-> bool*)
fun containsKeys_1 k [ ] = false
| containsKeys_1 k ((k1, _) :: HM) =
      (k = k1) orelse containsKeys_1 k HM;
```

```
$(* fn : ''a\ list -> (''a * 'b)\ list -> (''a * 'b)\ list *)$
$fun\ getElements\ [\,]\ HM = [\,]$
$|\ getElements\ (k :: KL)\ HM = (getElements_1\ k\ HM)\ @\ (getElements\ KL\ HM);$

$(* fn : ''a -> (''a * 'b)\ list -> (''a * 'b)\ list *)$
$fun\ getElements_1\ key\ [\,] = [\,]$
$|\ getElements_1\ key\ ((K,V) :: HM) =$
        $if\ key = K\ then\ [(K,V)]$
        $else\ getElements_1\ key\ HM$

$(* fn : ''a\ list -> (''a * 'b)\ list -> (''a * 'b)\ list *)$
$fun\ removeElm\ kList\ [\,] = [\,]$
$|\ removeElm\ kList\ ((K,V) :: HM) =$
    $if\ member\ K\ kList\ then\ removeElm\ kList\ HM$
    $else\ (K,V) :: removeElm\ kList\ HM$

$(* fn : ''a -> (''a * int * ''b)\ list -> ''b -> int *)$
$fun\ getOpID\ sid_{in}\ ((sid, opid, io) :: streamLut)\ io_{in} =$
    $if\ sid_{in} = sid\ andalso\ io_{in} = io\ then\ opid$
    $else\ getOpID\ sid_{in}\ streamLut\ io_{in};$

$(* fn : ''a -> ('b * ''a * ''c)\ list -> ''c -> 'b\ list *)$
$fun\ getSID\ opID_{in}\ [\,]\ io_{in} = [\,]$
$|\ getSID\ opID_{in}\ ((sid, opid, io) :: streamLut)\ io_{in} =$
    $if\ opID_{in} = opid\ andalso\ io_{in} = io\ then$
        $sid :: (getSID\ opID_{in}\ streamLut\ io_{in})$
    $else$
        $getSID\ opID_{in}\ streamLut\ io_{in};$

$(* fn : TIME -> STREAMID -> int -> KEY\ list -> (int * TIME)\ list *)$
$fun\ getDelay\ ts\ sid_n\ cnt\ [\,] = [\,]$
$|\ getDelay\ ts\ sid_n\ cnt\ (Key(t, n, sid, d) :: KL) =$
    $if\ sid_n = sid\ andalso\ t\ GTE\ ts\ andalso\ cnt > 0\ then$
        $(n, d) :: getDelay\ ts\ sid_n\ (cnt - 1)\ KL$
    $else\ getDelay\ ts\ sid_n\ cnt\ KL$

$(* fn : ''a\ list -> ''a\ list *)$
$fun\ remdupl\ [\,] = [\,]$
$|\ remdupl\ [x] = [x]$
$|\ remdupl\ (x :: xs) = if\ mem\ xs\ x\ then\ remdupl\ xs$
                      $else\ x :: remdupl\ xs;$
```

```
$(* fn : ''a\ list -> ''a -> bool *)$
$fun\ mem\ [\,]\ a = false$
$|\ mem\ (x :: xs)\ a = a = x\ orelse\ mem\ xs\ a;$

$(* fn : (int * ('a\ EVENT\ list -> 'a\ list\ EVENT\ list))\ list$
$ -> (int * ('a\ EVENT\ list -> 'a\ EVENT\ list -> ''c\ list\ EVENT\ list))\ list$
$ -> int -> STREAMID\ list -> STREAMID\ list -> KEY\ list$
$ -> (STREAMID * 'a\ EVENT\ list)\ list$
$ -> (STREAMID * 'a\ EVENT\ list)\ list *)$
$fun\ executeOp\ opList1\ opList2\ opID\ sid_{in}\ sid_{out}\ paList\ EM =$
$let$
    $val\ ((\_, eList) :: IM) = getElements\ sid_{in}\ EM;$
    $val\ output = (executeOp_1\ opList1\ opList2\ opID\ sid_{out}\ paList\ eList\ IM)$
    $val\ EM' = removeElm\ sid_{in}\ EM$
$in$
    $EM'\ @\ remdupl\ output$
$end$

$(* fn : (int * ('a\ EVENT\ list -> 'a\ list\ EVENT\ list))\ list$
$ -> (int * ('a\ EVENT\ list -> 'a\ EVENT\ list -> ''c\ list\ EVENT\ list))\ list$
$ -> int -> STREAMID\ list -> KEY\ list$
$ -> 'a\ EVENT\ list -> (STREAMID * 'a\ EVENT\ list)\ list$
$ -> (STREAMID * 'a\ EVENT\ list)\ list *)$
$fun\ executeOp_1\ opList1\ opList2\ opID\ (id :: sid_{out})\ paList\ eList1\ IM =$
$let$
  $val\ output = if\ IM = [\,]\ then\ (* unary\ operation *)$
                    $((assoc\ opID\ opList1)\ eList1)$
                  $else$
                    $let\ (* binary\ operation *)$
                      $val\ (\_, eList2) = hd\ IM;$
                    $in$
                      $((assoc\ opID\ opList2)\ eList1\ eList2)$
                    $end$
$in$
  $if\ sid_{out} = [\,]\ then$
     $let$
        $val\ output' = updateKeys\ id\ paList\ (length\ output)\ output$
     $in$
        $[(id, output')]$
     $end$
  $else\ multicast\ paList\ output\ (id :: sid_{out})$
$end$
```

```
$(* fn : STREAMID -> KEY list -> int -> 'a EVENT list$
$ -> 'a EVENT list *)$
$fun\ updateKeys\ sid_n\ paList\ cnt\ ((evt\ as\ Event(Key(t, n, sid, d), \_)) :: output) =$
$let$
    $val\ dList = getDelay\ t\ sid_n\ cnt\ paList$
$in$
    $if\ dList <> [\,]\ then$
        $updateKeys_1\ dList\ (evt :: output)$
    $else\ (evt :: output)$
$end$

$(* fn : (int * TIME)\ list -> 'a EVENT list -> 'a EVENT list *)$
$fun\ updateKeys_1\ \_\ [\,] = [\,]$
$|\ updateKeys_1\ ((sn, d_n) :: dList)\ (Event(Key(t, n, sid, d), e) :: output) =$
    $(Event(Key(t ++ d_n, sn, sid, d_n), e)) :: (updateKeys_1\ dList\ output)$


$(* fn : KEY list -> 'a EVENT list -> STREAMID list ->$
$ (STREAMID * 'a EVENT list)\ list *)$
$fun\ multicast\ paList\ output\ [\,] = [\,]$
$|\ multicast\ paList\ output\ (id :: sidList) =$
    $let$
        $val\ output' = updateKeys\ id\ paList\ (length\ output)\ output$
    $in$
        $(id, output') :: (multicast\ paList\ output\ sidList)$
    $end$
```

Appendix B

SML code for the provenance query case study

B.1 An example provenance query

```
(* provenance assertions − event keys − used in this case study *)

− val key1 = Key(Time 1279398105675, 1, StreamID(1), Time 0);
− val key2 = Key(Time 1279398105684, 1, StreamID(2), Time 9);
− val key3 = Key(Time 1279398105684, 1, StreamID(4), Time 9);
− val key4 = Key(Time 1279398105693, 1, StreamID(3), Time 9);
− val key5 = Key(Time 1279398105701, 1, StreamID(5), Time 17);
− val key6 = Key(Time 1279398105702, 1, StreamID(6), Time 1);
− val key7 = Key(Time 1279398107678, 2, StreamID(1), Time 0);
− val key8 = Key(Time 1279398107678, 2, StreamID(2), Time 0);
− val key9 = Key(Time 1279398107678, 2, StreamID(4), Time 0);
− val key10 = Key(Time 1279398107680, 2, StreamID(5), Time 2);
− val key11 = Key(Time 1279398107681, 2, StreamID(3), Time 3);
− val key12 = Key(Time 1279398107682, 2, StreamID(6), Time 2);
− val key13 = Key(Time 1279398109678, 3, StreamID(1), Time 0);
− val key14 = Key(Time 1279398109678, 3, StreamID(2), Time 0);
− val key15 = Key(Time 1279398109678, 3, StreamID(4), Time 0);
− val key16 = Key(Time 1279398109680, 3, StreamID(5), Time 2);
− val key17 = Key(Time 1279398109681, 3, StreamID(3), Time 3);
− val key18 = Key(Time 1279398109682, 3, StreamID(6), Time 2);
```

$(*\ provenance\ assertions\ for\ each\ stream\ *)$

$-\ val\ S1_m\ =\ [key1, key7, key13];$
$-\ val\ S2_m\ =\ [key2, key8, key14];$
$-\ val\ S3_m\ =\ [key4, key11, key17];$
$-\ val\ S4_m\ =\ [key3, key9, key15];$
$-\ val\ S5_m\ =\ [key5, key10, key16];$
$-\ val\ S6_m\ =\ [key6, key12, key18];$

$(*\ prepare\ a\ look\ up\ table\ (list)\ for\ stream\ ancestor\ functions\ *)$

$(*\ fn\ :\ 'a -> 'a\ list\ *)$
$fun\ toList\ \ key\ =\ [key]\ @\ [\,];$

$-\ val\ saf1\ =\ toList\ o\ Map_{OA}(S1_m);$
$-\ val\ saf2\ =\ toList\ o\ Filter_{OA}(S2_m);$
$-\ val\ saf3\ =\ TW_{OA}(Time(5000), S4_m);$
$-\ val\ saf4\ =\ toList\ o\ Map_{OA}(S5_m);$
$-\ val\ saf5\ =\ JoinTW_{OA}(Time(1000), Time(1000), S3_m, S6_m);$

$-\ val\ safList\ =\ [(StreamID(2), saf1), (StreamID(4), saf1), (StreamID(3), saf2)$
$\quad , (StreamID(5), saf3), (StreamID(6), saf4), (StreamID(7), saf5)];$
$\quad :\ (STREAMID\ *\ (KEY -> KEY\ list))\ list$

$(*\ execute\ a\ provenance\ query\ by\ using\ the\ retrieveAncestors\ function\ *)$

$-\ val\ getAnc\ =\ retrieveAncestors(safList, [StreamID(1)]);$
$-\ getAnc\ \ [(Key(Time\ 1279398109685, 3,\ StreamID(7), Time\ 3))];$

$(*\ provenance\ query\ results\ *)$

$>\ [Key\ (Time\ 1279398105675, 1, StreamID\ 1, Time\ 0)$
$\quad , Key\ (Time\ 1279398107678, 2, StreamID\ 1, Time\ 0),$
$\quad Key\ (Time\ 1279398109678, 3, StreamID\ 1, Time\ 0)]\ :\ KEY\ list$

B.2 Example stream replay execution

```
(* prepare look up tables (lists) for stream operations *)

(* fn : 'a EVENT list -> 'a list EVENT list *)
fun toEventList [ ] = [ ]
| toEventList ((Event(key, value)) :: EL) =
          (Event(key, [value])) :: toEventList EL;

- val op1 = toEventList o Map(Ceil, StreamID(2));
- val op2 = toEventList o Filter(filterCond, 1, StreamID(3));
- val op3 = TW(Time(5000), [ ], StreamID(5));
- val op4 = toEventList o Map(avg(0, 0), StreamID(6));
- val op5 = JoinTW(Time(1000), Time(1000), cartesianJoin, 1, 1, StreamID(7));

- val opList1 = [(1, op1), (2, op2), (3, op3), (4, op4)];
- val opList2 = [(5, op5)];


(* prepare a look up table (list) for input and output streams *)

- val streamLut = [(StreamID(1), 1, "I"), (StreamID(2), 1, "O"), (StreamID(2), 2, "I")
     , (StreamID(4), 1, "O"), (StreamID(4), 3, "I"), (StreamID(3), 2, "O")
     , (StreamID(3), 5, "I"), (StreamID(5), 3, "O"), (StreamID(5), 4, "I")
     , (StreamID(6), 4, "O"), (StreamID(6), 5, "I"), (StreamID(7), 5, "O")];
     : (STREAMID * int * string) list


(* prepare a list of targeted stream IDs *)

- val sidList = [StreamID(7)];


(* prepare a list of provenance assertions *)

- val paList = [key1, key2, key3, key4, key5, key6, key7, key8, key9, key10,
                key11, key12, key13, key14, key15, key16, key17, key18];


(* execute a stream replay by using the replayExec function *)

- val replayResult = replayExec(opList1, opList2, streamLut, sidList, paList)
- replayResult [(StreamID(1), [Event(key13, 93.28)])]
```

```
(* replay execution results *)

> [(Stream7, [Event((Key (Time 1279398109685, 3, StreamID 7, Time 3))
    , (90.33, 94.0))])] : (STREAMID * real EVENT list) list
```

Appendix C

Utility functions for the on-the-fly provenance query algorithm

```
$(* fn : ''a-> ''a\ list -> bool *)$
$fun\ member\ e\ [\,] = false$
$|\ member\ e\ (e1 :: l) =$
            $(e = e1)\ orelse\ member\ e\ l;$
```

```
$(* fn : ''a -> (''a * 'b)\ list -> 'b *)$
$fun\ assoc\ sid\ ((id, func) :: opList) =$
        $if\ (sid = id)\ then\ func$
        $else\ assoc\ sid\ opList;$
```

```
$(* fn : ''a -> ''b -> (''a * 'c * ''b)\ list -> bool *)$
$fun\ containSID\ sid_{in}\ io_{in}\ [\,] = false$
$|\ containSID\ sid_{in}\ io_{in}\ ((sid, \_, io) :: streamLut) =$
    $if(sid_{in} = sid\ andalso\ io_{in} = io)\ then\ true$
    $else\ containSID\ sid_{in}\ io_{in}\ streamLut$
```

```
$(* fn : ''a -> (''a * int * ''b)\ list -> ''b -> int *)$
$fun\ getOpID\ sid_{in}\ [\,]\ io_{in} = 0$
$|\ getOpIDsid_in((sid, opid, io) :: streamLut)\ io_{in} =$
    $if\ sid_{in} = sid\ andalso\ io_{in} = io\ then\ opid$
    $else\ getOpID\ sid_{in}\ streamLut\ io_{in};$
```

```
$(* fn : ''a -> ('b * ''a * ''c) list -> ''c -> 'b list *)$
$fun getSID opID_{in} [ ] io_{in} = [ ]$
$| getSID opID_{in} ((sid, opid, io) :: streamLut) io_{in} =$
    $if opID_{in} = opid andalso io_{in} = io then$
        $sid :: (getSID opID_{in} streamLut io_{in})$
    $else$
        $getSID opID_{in} streamLut io_{in};$


$(* fn : ''a -> (''a * int * InOut) list -> ''a list *)$
$fun getRsID sid streamLut = getSID(getOpID sid streamLut O) streamLut I;$


$(* fn : int * string * (int * string * int * string) list -> int * string *)$
$fun getParam opID_{in} pName_{in} ((opid, pName, param, pType) :: paramLut) =$
   $if(opID_{in} = opid andalso pName_{in} = pName) then (param, pType)$
    $else getParam opID_{in} pName_{in} paramLut$


$(* fn : ''a list -> (''a * 'b) list -> bool *)$
$fun containsKeys [ ] HM = true$
$| containsKeys (k :: KL) HM =$
    $if containsKeys_1 k HM then$
        $containsKeysKLHM$
    $else false$

$(*fn :'' a-> (''a *' b)list-> bool*)$
$fun containsKeys_1 k [ ] = false$
$| containsKeys_1 k ((k1, \_) :: HM) =$
        $(k = k1) orelse containsKeys_1 k HM;$


$(* fn : ''a list -> (''a * 'b) list -> (''a * 'b) list *)$
$fun getElements [ ] HM = [ ]$
$| getElements (k :: KL) HM = (getElements_1 k HM) @ (getElements KL HM);$

$(* fn : ''a -> (''a * 'b) list -> (''a * 'b) list *)$
$fun getElements_1 key [ ] = [ ]$
$| getElements_1 key ((K, V) :: HM) =$
        $if key = K then [(K, V)]$
        $else getElements_1 key HM$
```

```
$(* fn : (STREAMID * 'a)\ list -> (STREAMID * int * InOut)\ list$
$-> (int * string * int * string)\ list -> (STREAMID * 'b\ ASSERTION\ list)\ list$
$-> 'c\ ASSERTION -> (STREAMID * 'b\ ASSERTION\ list)\ list *)$
$fun\ Dequeue\ psafLut1\ streamLut\ paramLut\ B\ (pa\ as\ Assertion(Key(t, n, sid, d), \_)) =$
  $if\ containsKeys\ [sid]\ psafLut1\ then$
    $let\ (*\ unary\ operation\ *)$
      $val\ rsID = hd\ (getRsID\ sid\ streamLut);$
      $val\ ((\_, rs) :: RS) = getElements\ [rsID]\ B;$
      $val\ opID = getOpID\ sid\ streamLut\ O;$
      $val\ (w, wType) = getParam\ opID\ "w"\ paramLut;$
      $val\ rs' = if\ (wType = "T")\ then$
                      $DequeueT(t - -d - -Time(w), rs)$
                $else$
                      $DequeueN(n - w + 1, rs)$
    $in$
      $(replaceElm\ [rsID]\ [(rsID, rs')]\ B)$
    $end$
  $else$
    $let\ (*\ binary\ operation\ *)$
      $val\ rsIDList = (getRsID\ sid\ streamLut);$
      $val\ ((rsID1, rs1) :: B1') = getElements\ [hd\ rsIDList]\ B;$
      $val\ ((rsID2, rs2) :: B2') = getElements\ (tl\ rsIDList)\ B;$
      $val\ opID = (getOpID\ sid\ streamLut\ O);$
      $val\ (w1, wType1) = getParam\ opID\ "w1"\ paramLut;$
      $val\ (w2, wType2) = getParam\ opID\ "w2"\ paramLut;$
      $val\ [rs1', rs2'] = if\ (wType1 = "T")\ then$
                          $[DequeueT(t - -d - -Time(w1), rs1)]$
                          $@\ [DequeueT(t - -d - -Time(w2), rs2)]$
                    $else$
                          $[DequeueN(n - w1 + 1, rs1)]$
                          $@\ [DequeueN(n - w2 + 1, rs2)]$
    $in$
      $(replaceElm\ rsIDList\ [(rsID1, rs1'), (rsID2, rs2')]\ B)$
    $end$
```

```
$(* fn : TIME * 'a\ ASSERTION\ list -> 'a\ ASSERTION\ list *)$
$funDequeueT(lb, [\,]) = [\,]$
$|\ DequeueT(lb, ((paasAssertion(Key(t, \_, \_, \_), \_)) :: Q)) =$
  $if\ (t\ GT\ lb)\ then$
      $pa :: DequeueT(lb, Q)$
  $else\ [\,];$
```

```
(* fn : int * 'a ASSERTION list -> 'a ASSERTION list *)
fun DequeueN(lb, [ ]) = [ ]
| DequeueN(lb, ((pa as Assertion(Key(_, n, _, _), _)) :: Q)) =
    if (n > lb) then
        pa :: DequeueN(lb, Q)
    else [ ];
```

```
(* fn : ''a list -> (''a * 'b) list -> (''a * 'b) list *)
fun removeElm kList [ ] = [ ]
| removeElm kList ((K, V) :: HM) =
    if member K kList then removeElm kList HM
    else (K, V) :: removeElm kList HM
```

```
(* fn : ''a -> (''a * 'b list) list -> 'b -> (''a * 'b list) list *)
fun updateElm sid B pa =
let
    val ((_, rs) :: RS) = getElements [sid] B;
    val B' = removeElm [sid] B;
in
    (sid, (pa :: rs)) :: B'
end
```

```
(* fn : (STREAMID * 'a ASSERTION list) list
 -> 'a ASSERTION -> (STREAMID * 'a ASSERTION list) list *)
fun addElm B (pa as Assertion(Key(_, _, sid, _), _)) =
    if (containsKeys [sid] B) then (updateElm sid B pa)
    else (sid, [pa]) :: B
```

```
(* fn : ''a list -> (''a * 'b) list
-> (''a * 'b) list -> (''a * 'b) list *)
fun replaceElm kList elmList B =
let
    val B' = removeElm kList B;
in
    elmList @ B'
end
```

```
$(* \, fn \, : \, STREAMID \, * \, (STREAMID \, * \, 'a \, * \, InOut) \, list -> \, bool \, *)$
$fun \; isFirstIStream \;\; sid \;\; streamLut \; =$
    $(containSID \;\; sid \;\; I \;\; streamLut) \;\; andalso \;\; not \, (containSID \;\; sid \;\; O \;\; streamLut);$


$(* \, fn \, : \, KEY -> \, 'a \, ASSERTION \, *)$
$fun \; ASU \;\; (key : KEY) \; = \; (Assertion(key, [\,]))$


$(* \, fn \, : \, (KEY -> \, 'a \, list -> \, 'b) -> \, 'a \, ASSERTION -> \, 'b \, ASSERTION \, *)$
$fun \; PCU \;\; func \;\; (Assertion(key, pl)) \; = \; (Assertion(key, [func \;\; key \;\; pl]));$


$(* \, fn \, : \, (STREAMID \, * \, ('a -> \, 'b \, ASSERTION -> \, 'a \, ASSERTION)) \, list$
$-> \, (STREAMID \, * \, ('a -> \, 'a -> \, 'b \, ASSERTION -> \, 'a \, ASSERTION)) \, list$
$-> \, (STREAMID \, * \, int \, * \, InOut) \, list -> \, (KEY -> \, 'c \, list -> \, 'd)$
$-> \, (STREAMID \, * \, 'a) \, list -> \, 'b \, ASSERTION -> \, 'a \, ASSERTION \, *)$
$fun \; executePSAF \;\; psaf1 \;\; psaf2 \;\; streamLut \;\; pFun \;\; B \;\; (pa \; as \; Assertion(Key(\_, \_, sid, \_), \_)) \; =$
    $if \; (containsKeys \;\; [sid] \;\; psaf1) \; then$
        $let \; (* \, unary \; operation \, *)$
            $val \; rsID \; = \; hd \;\; (getRsID \;\; sid \;\; streamLut);$
            $val \; ((\_, rs) :: RS) \; = \; getElements \;\; [rsID] \;\; B;$
        $in$
            $PCU \;\; pFun \;\; ((assoc \;\; sid \;\; psaf1) \;\; rs \;\; pa)$
        $end$
    $else$
        $let \; (* \, binary \; operation \, *)$
            $val \; rsIDList \; = \; (getRsID \;\; sid \;\; streamLut);$
            $val \; ((\_, rs1) :: B1') \; = \; getElements \;\; [hd \;\; rsIDList] \;\; B;$
            $val \; ((\_, rs2) :: B2') \; = \; getElements \;\; (tl \;\; rsIDList) \;\; B;$
        $in$
            $PCU \;\; pFun \;\; ((assoc \;\; sid \;\; psaf2) \;\; rs1 \;\; rs2 \;\; pa)$
        $end$
```

Appendix D

SML code for the on-the-fly provenance query case study

A list of input provenance assertions (event keys) used in the case study:

$$\begin{aligned}
-\ val\ aStream\ =\ & [Key(Time1279398105675, 1, StreamID(1), Time0),\\
& Key(Time1279398105684, 1, StreamID(2), Time9),\\
& Key(Time1279398105684, 1, StreamID(4), Time9),\\
& Key(Time1279398105693, 1, StreamID(3), Time9),\\
& Key(Time1279398105701, 1, StreamID(5), Time17),\\
& Key(Time1279398105702, 1, StreamID(6), Time1),\\
& Key(Time1279398105703, 1, StreamID(7), Time1),\\
& Key(Time1279398107678, 2, StreamID(1), Time0),\\
& Key(Time1279398107678, 2, StreamID(2), Time0),\\
& Key(Time1279398107678, 2, StreamID(4), Time0),\\
& Key(Time1279398107680, 2, StreamID(5), Time2),\\
& Key(Time1279398107681, 2, StreamID(3), Time3),\\
& Key(Time1279398107682, 2, StreamID(6), Time2),\\
& Key(Time1279398107683, 2, StreamID(7), Time1),\\
& Key(Time1279398109678, 3, StreamID(1), Time0),\\
& Key(Time1279398109678, 3, StreamID(2), Time0),\\
& Key(Time1279398109678, 3, StreamID(4), Time0),\\
& Key(Time1279398109680, 3, StreamID(5), Time2),\\
& Key(Time1279398109681, 3, StreamID(3), Time3),\\
& Key(Time1279398109682, 3, StreamID(6), Time2),\\
& Key(Time1279398109683, 3, StreamID(7), Time1)];
\end{aligned}$$

Required parameters for the on-the-fly query algorithm:

```
(* prepare look up tables for property stream ancestor functions *)

- val psaf1 = $Map_{psaf}$
- val psaf2 = $Filter_{psaf}$
- val psaf3 = $TW_{psaf}(Time(5000))$
- val psaf4 = $Map_{psaf}$
- val psaf5 = $JoinTW_{psaf}(Time(1000), Time(1000))$

- val psafLut1 = [(StreamID(2), psaf1), (StreamID(4), psaf1), (StreamID(3), psaf2),
                  (StreamID(5), psaf3), (StreamID(6), psaf4)];
- val psafLut2 = [(StreamID(7), psaf5)];


(* prepare a look up table for input and output streams *)

- val streamLUT = [(StreamID(1), 1, I), (StreamID(2), 1, O), (StreamID(2), 2, I)
                 , (StreamID(4), 1, O), (StreamID(4), 3, I), (StreamID(3), 2, O)
                 , (StreamID(3), 5, I), (StreamID(5), 3, O), (StreamID(5), 4, I)
                 , (StreamID(6), 4, O), (StreamID(6), 5, I), (StreamID(7), 5, O)];
                 : (STREAMID * int * InOut) list


(* prepare a list of targeted stream IDs *)

- val sidList = [StreamID(7)];


(* property computing function *)

(* fn : KEY -> TIME list -> TIME *)
fun totalDelay (key as Key(_, _, sid, d)) pList =
      if (isJoinOp sid streamLut) then
         (maxDelay (Time(0)) pList) ++ d
      else
         $totalDelay_1$ key pList

fun $totalDelay_1$ (Key(t, n, sid, d)) [ ] = d
| $totalDelay_1$ (Key(t, n, sid, d)) [p] = p ++d
| $totalDelay_1$ (Key(t, n, sid, d)) (p :: pList) = p ++ $totalDelay_1$ (Key(t, n, sid, d)) pList
```

$(*\, fn\ :\ ''a -> (''a\ *\ int\ *\ InOut)\, list -> bool\, *)$
$fun\, isJoinOp\ (sid)\ streamLut\ =\ ((getOpID\ sid\ streamLut\ O)\ =\ 5);$

$(*\, fn\ :\ TIME -> TIME\, list -> TIME\, *)$
$fun\, maxDelay\ maxD\ [\,]\ =\ maxD$
$|\, maxDelay\ maxD\ (d :: dList)\ =\ maxDelay\ (MAX_{TIME}(maxD, d))\ dList;$

$(*\, fn\ :\ TIME\ *\ TIME -> TIME\, *)$
$fun\, MAX_{TIME}(Time(x), Time(y))\ =\ if\ x\ >\ y\ then\ Time(x)\ else\ Time(y);$

The on-the-fly provenance query for the case study can be performed as follows:

$(*\, execute\ provenance\ queries\ by\ using\ the\ OTFquery\ function\, *)$

$-\ val\ query\ =\ OTFpquery(psafLut1, psafLut2, sidList, streamLut, paramLut, totalDelay);$
$-\ query\ \ aStream;$

$(*\, query\ results\, *)$

$>\ [Assertion(Key(Time\ 1279398109683, 3, StreamID\ 7, Time\ 1), [Time 14]),$
$\quad Assertion(Key(Time\ 1279398107683, 2, StreamID\ 7, Time\ 1), [Time\ 14])$
$\quad Assertion(Key(Time\ 1279398105703, 1, StreamID\ 7, Time\ 1), [Time\ 28])]$
$\quad :\ TIME\ ASSERTION\ list$

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the borealis stream processing engine," in *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR'05)*, Asilomar, California, USA, 2005, pp. 277–289.

[2] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom, "Trio: a system for data, uncertainty, and lineage," in *Proceedings of the 32nd international conference on Very large data bases (VLDB '06)*, Seoul, Korea, 2006, pp. 1151–1154.

[3] G. Alonso and A. El Abbadi, "Cooperative modeling in applied geographic research," University of California at Santa Barbara, Technical Report, 1994.

[4] G. Alonso and C. Hagen, "Geo-opera: Workflow concepts for spatial processes," in *Proceedings of the 5th International Symposium on Advances in Spatial Databases (SSD'97)*, Berlin, Germany, 1997, pp. 238–258.

[5] G. Alonso, C. Hagen, H.-J. Schek, and M. Tresch, "Distributed processing over stand-alone systems and applications," in *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, Athens, Greece, 1997, pp. 575–579.

[6] G. Amato, S. Chessa, and C. Vairo, "Mad-wise: a distributed stream management system for wireless sensor networks," *Software Practice and Experience*, vol. 40, no. 5, pp. 431–451, April 2010.

[7] M. K. Anand, S. Bowers, T. McPhillips, and B. Ludäscher, "Efficient provenance storage over nested data collections," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT'09)*, Saint Petersburg, Russia, 2009, pp. 958–969.

[8] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," Stanford InfoLab, Stanford University, Technical Report, 2004.

[9] A. Arasu, S. Babu, and J. Widom, “The cql continuous query language: semantic foundations and query execution,” *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, June 2006.

[10] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, “Models and issues in data stream systems,” in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS'02)*, Madison, Wisconsin, USA, 2002, pp. 1–16.

[11] B. Babcock, M. Datar, and R. Motwani, “Load shedding for aggregation queries over data streams,” in *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*, Boston, Massachusetts, USA, 2004, pp. 350–361.

[12] S. Babu and J. Widom, “Continuous queries over data streams,” *SIGMOD Record*, vol. 30, no. 3, pp. 109–120, 2001.

[13] M. Balazinska, A. Deshpande, M. J. Franklin, P. B. Gibbons, J. Gray, M. Hansen, M. Liebhold, S. Nath, A. Szalay, and V. Tao, “Data management in the worldwide sensor web,” *IEEE Pervasive Computing*, vol. 6, no. 2, pp. 30–40, 2007.

[14] R. Barga and L. Digiampietri, “Automatic generation of workflow provenance,” in *Provenance and Annotation of Data*, ser. Lecture Notes in Computer Science, L. Moreau and I. Foster, Eds. Springer Berlin / Heidelberg, 2006, vol. 4145, pp. 1–9.

[15] R. S. Barga and L. A. Digiampietri, “Automatic capture and efficient storage of e-science experiment provenance,” *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 419–429, April 2008.

[16] O. Benjelloun, A. D. Sarma, C. Hayworth, and J. Widom, “An introduction to uldbs and the trio system,” *IEEE Data Engineering Bulletin, Special Issue on Probabilistic Databases*, vol. 29, no. 1, pp. 5–16, 2006.

[17] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, “An annotation management system for relational databases,” *The VLDB Journal*, vol. 14, no. 4, pp. 373–396, 2005.

[18] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, “IBM infosphere streams for scalable, real-ti intelligent transportation services,” in *Proceedings of the 2010 international conference on Management of data (SIGMOD'10)*, Indianapolis, Indiana, USA, 2010, pp. 1093–1104.

[19] M. Blount, J. Davis, M. Ebling, J. H. Kim, K. H. Kim, K. Lee, A. Misra, S. Park, D. Sow, Y. J. Tak, M. Wang, and K. Witting, “Century: Automated aspects of

patient care," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, Daegu, Korea, 2007, pp. 504–509.

[20] R. Bose and J. Frew, "Composing lineage metadata with xml for custom satellite-derived data products," in *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM'04)*, Santorini Island, Greece, 2004, pp. 275–284.

[21] ——, "Lineage retrieval for scientific data processing: a survey," *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 1 – 28, 2005.

[22] S. Bowers and B. Ludscher, "Actor-oriented design of scientific workflows," in *Conceptual Modeling ER 2005*, ser. Lecture Notes in Computer Science, L. Delcambre, C. Kop, H. Mayr, J. Mylopoulos, and O. Pastor, Eds. Springer Berlin / Heidelberg, 2005, vol. 3716, pp. 369–384.

[23] S. Bowers, T. McPhillips, M. Wu, and B. Ludäscher, "Project histories: managing data provenance across collection-oriented scientific workflow runs," in *Proceedings of the 4th international conference on Data integration in the life sciences (DILS'07)*, Philadelphia, PA, USA, 2007, pp. 122–138.

[24] S. Bowers, T. M. McPhillips, and B. Ludascher, "Provenance in collection-oriented scientific workflows," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 519–529, 2008.

[25] P. Buneman, S. Khanna, and W. C. Tan, "Why and where: A characterization of data provenance," in *Proceedings of the 8th International Conference on Database Theory (ICDT'01)*, London, UK, 2001, pp. 316–330.

[26] P. Buneman, S. Khanna, and W.-C. Tan, "On propagation of deletions and annotations through views," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS'02)*, Madison, Wisconsin, USA, 2002, pp. 150–158.

[27] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Managing the evolution of dataflows with vistrails," in *Proceedings of the 22nd International Conference on Data Engineering Workshops (ICDEW'06)*, Atlanta, Georgia, USA, 2006, pp. 71–75.

[28] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams: a new class of data management applications," in *Proceedings of the 28th international conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, 2002, pp. 215–226.

[29] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "Telegraphcq: continuous dataflow processing," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD'03)*, San Diego, California, USA, 2003, pp. 668–668.

[30] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "Niagaracq: a scalable continuous query system for internet databases," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data (SIGMOD'00)*, Dallas, Texas, USA, 2000, pp. 379–390.

[31] J. Cheney, L. Chiticariu, and W.-C. Tan, "Provenance in databases: Why, how, and where," *Found. Trends databases*, vol. 1, no. 4, pp. 379–474, April 2009.

[32] M. Cherniack, "Squal: The aurora [s]tream [qu]ery [al]gebra," Brandeis University, Technical Report, 2003.

[33] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, "DBNotes: a post-it system for relational databases based on provenance," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD'05)*, Baltimore, Maryland, USA, 2005, pp. 942–944.

[34] A. Chowdhury, B. Falchuk, and A. Misra, "Medially: A provenance-aware remote health monitoring middleware," in *Proceedings of the Eighth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2010)*, Mannheim, Germany, 2010, pp. 125–134.

[35] G. Cormode and S. Muthukrishnan, "What's new: finding significant differences in network data streams," *IEEE/ACM Transactions on Networking*, vol. 13, no. 6, pp. 1219–1232, 2005.

[36] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck, "Gigascope: high performance network monitoring with an sql interface," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data (SIGMOD'02)*, Madison, Wisconsin, USA, 2002, pp. 623–623.

[37] Y. Cui and J. Widom, "Tracing the lineage of view data in a warehousing environment," *ACM Transactions on Database Systems*, vol. 25, no. 2, pp. 179–227, 2000.

[38] ——, "Lineage tracing for general data warehouse transformations," *The VLDB Journal*, vol. 12, no. 1, pp. 41–58, May 2003.

[39] B. Cutt and R. Lawrence, "Managing data quality in a terabyte-scale sensor archive," in *Proceedings of the 2008 ACM symposium on Applied computing (SAC'08)*, Fortaleza, Ceara, Brazil, 2008, pp. 982–986.

[40] C. Dai, H.-S. Lim, E. Bertino, and Y.-S. Moon, "Assessing the trustworthiness of location data based on provenance," in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'09)*, Seattle, Washington, USA, 2009, pp. 276–285.

[41] S. B. Davidson, S. C. Boulakia, A. Eyal, B. Ludscher, T. M. McPhillips, S. Bowers, M. K. Anand, and J. Freire, "Provenance in scientific workflow systems," *IEEE Data Engineering Bulletin*, vol. 30, no. 4, pp. 44–50, 2007.

[42] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, July 2005.

[43] J. Dunkel, A. Fernndez, R. Ortiz, and S. Ossowski, "Event-driven architecture for decision support in traffic management systems," *Expert Systems with Applications: An International Journal*, vol. 38, no. 6, pp. 6530–6539, 2011.

[44] F. Dvorák, D. Kouril, A. Krenek, L. Matyska, M. Mulac, J. Pospísil, M. Ruda, Z. Salvet, J. Sitera, and M. Vocu, "glite job provenance," in *Provenance and Annotation of Data*, ser. Lecture Notes in Computer Science, L. Moreau and I. Foster, Eds. Springer Berlin / Heidelberg, 2006, vol. 4145, pp. 246–253.

[45] EGEE Project. (2011, july) glite – ligthweight middleware for grid computing. [Online]. Available: http://glite.cern.ch/

[46] EsperTech Inc. (2011, August) Event stream intelligence: Esper & nesper. [Online]. Available: http://esper.codehaus.org/

[47] J. Freire, C. Silva, S. Callahan, E. Santos, C. Scheidegger, and H. Vo, "Managing rapidly-evolving scientific workflows," in *Provenance and Annotation of Data*, ser. Lecture Notes in Computer Science, L. Moreau and I. Foster, Eds. Springer Berlin / Heidelberg, 2006, vol. 4145, pp. 10–18.

[48] J. Frew and R. Bose, "Earth system science workbench: A data management infrastructure for earth science products," in *Proceedings of the 13th International Conference on Scientific and Statistical Database Management (SSDBM'01)*, Fairfax, Virginia, USA, 2001, pp. 180–189.

[49] J. Frew, D. Metzger, and P. Slaughter, "Automatic capture and reconstruction of computational provenance," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 485–496, April 2008.

[50] F. Geerts, A. Kementsietsidis, and D. Milano, "MONDRIAN: Annotating and querying databases through colors and blocks," in *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, Atlanta, Georgia, USA, 2006, pp. 82–92.

[51] F. Geerts and J. Van Den Bussche, “Relational completeness of query languages for annotated databases,” in *Proceedings of the 11th international conference on Database programming languages (DBPL’07)*, Vienna, Austria, 2007, pp. 127–137.

[52] S. Geisler, C. Quix, and S. Schiffer, “A data stream-based evaluation framework for traffic information systems,” in *Proceedings of the ACM SIGSPATIAL International Workshop on GeoStreaming (IWGS’10)*, San Jose, California, USA, 2010, pp. 11–18.

[53] Y. Gil, J. Kim, V. Ratnakar, and E. Deelman, “Wings for pegasus: A semantic approach to creating very large scientific workflows.” in *Proceedings of the 2nd international workshop on ”OWL: Experiences and Directions (OWLED’06)*, Athens, Georgia, USA, 2006.

[54] Y. Gil, V. Ratnakar, E. Deelman, G. Mehta, and J. Kim, “Wings for pegasus: creating large-scale scientific applications using semantic representations of computational workflows,” in *Proceedings of the 19th national conference on Innovative applications of artificial intelligence - Volume 2 (IAAI’07)*, Vancouver, British Columbia, Canada, 2007, pp. 1767–1774.

[55] A. Gilani, S. Sonune, B. Kendai, and S. Chakravarthy, “The anatomy of a stream processing system,” in *Flexible and Efficient Information Handling*, ser. Lecture Notes in Computer Science, D. Bell and J. Hong, Eds. Springer Berlin / Heidelberg, 2006, vol. 4042, pp. 232–239.

[56] S. Gilmore. (2009, March) Programming in Standard ML ’97: An on-line tutorial. [Online]. Available: http://homepages.inf.ed.ac.uk/stg/NOTES/

[57] B. Glavic and G. Alonso, “Perm: Processing provenance and data on the same data model through query rewriting,” in *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE’09)*, Washington, DC, USA, 2009, pp. 174–185.

[58] ——, “Provenance for nested subqueries,” in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT ’09)*, Saint Petersburg, Russia, 2009, pp. 982–993.

[59] L. Golab and M. T. Ozsu, “Data stream management issues - a survey,” School of Computer Science, University of Waterloo, Technical Report, 2003.

[60] ——, “Issues in data stream management,” *ACM SIGMOD Record*, vol. 32, no. 2, pp. 5–14, 2003.

[61] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen, “Update exchange with mappings and provenance,” in *Proceedings of the 33rd international conference on Very large data bases (VLDB ’07)*, Vienna, Austria, 2007, pp. 675–686.

[62] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS'07)*, Beijing, China, 2007, pp. 31–40.

[63] S. Greenhill and S. Venkatesh, "Virtual observers in a mobile surveillance system," in *Proceedings of the 14th annual ACM international conference on Multimedia (MULTIMEDIA '06)*, Santa Barbara, California, USA, 2006, pp. 579–588.

[64] P. Groth, "The origin of data," PhD. Thesis, University of Southampton, 2007.

[65] P. Groth, S. Jiang, S. Miles, S. Munroe, V. Tan, S. Tsasakou, and L. Moreau, "An architecture for provenance systems," University of Southampton, Technical Report, 2006.

[66] P. Groth, S. Miles, and L. Moreau, "PReServ: Provenance recording for services," in *Proceedings of the UK e-Science All Hands Meeting 2005*, Nottingham, UK, 2005.

[67] P. Groth and L. Moreau, "Recording process documentation for provenance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 9, pp. 1246–1259, September 2009.

[68] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid, "Scheduling for shared window joins over data streams," in *Proceedings of the 29th international conference on Very large data bases (VLDB'03)*, Berlin, Germany, 2003, pp. 297–308.

[69] Q. Han, S. Mehrotra, and N. Venkatasubramanian, "Application-aware integration of data collection and power management in wireless sensor networks," *Journal of Parallel and Distributed Computing*, vol. 67, no. 9, pp. 992–1006, 2007.

[70] K. Hildrum, F. Douglis, J. L. Wolf, P. S. Yu, L. Fleischer, and A. Katta, "Storage optimization for large-scale distributed stream-processing systems," *ACM Transactions on Storage*, vol. 3, pp. 1–28, February 2008.

[71] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn, "Taverna: a tool for building and running workflows of services." *Nucleic Acids Research*, vol. 34, pp. 729–732, July 2006.

[72] M. R. Huq, A. Wombacher, and P. M. G. Apers, "Facilitating fine grained data provenance using temporal data model," in *Proceedings of the Seventh International Workshop on Data Management for Sensor Networks (DMSN'10)*, Singapore, 2010, pp. 8–13.

[73] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik, "Towards a streaming

sql standard," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1379–1390, August 2008.

[74] S. R. Jeffery, M. Garofalakis, and M. J. Franklin, "Adaptive cleaning for RFID data streams," in *Proceedings of the 32nd international conference on Very large data bases (VLDB'06)*, Seoul, Korea, 2006, pp. 163–174.

[75] A. Kementsietsidis and M. Wang, "On the efficiency of provenance queries," in *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE'09)*, Shanghai, China, 2009, pp. 1223–1226.

[76] ——, "Provenance query evaluation: what's so special about it?" in *Proceeding of the 18th ACM conference on Information and knowledge management (CIKM'09)*, Hong Kong, China, 2009, pp. 681–690.

[77] J. Kim, E. Deelman, Y. Gil, G. Mehta, and V. Ratnakar, "Provenance trails in the wings-pegasus system," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 587–597, April 2008.

[78] C. Koncilia, "A bi-temporal data warehouse model," in *Proceedings of the Fifteenth International Conference on Advanced Information Systems Engineering (CAiSE'03)*, Klagenfurt/Velden, Austria, 2003, pp. 77–80.

[79] A. Kruger, R. Lawrence, and E. C. Dragut, "Building a terabyte nexrad radar database for hydrometeorology research," *Computers and Geosciences*, vol. 32, no. 2, pp. 247–258, March 2006.

[80] J. Kurose, E. Lyons, D. McLaughlin, D. Pepyne, B. Philips, D. Westbrook, and M. Zink, "An end-user-responsive sensor network architecture for hazardous weather detection, prediction and response," in *Technologies for Advanced Heterogeneous Networks II*, ser. Lecture Notes in Computer Science, K. Cho and P. Jacquet, Eds. Springer Berlin / Heidelberg, 2006, vol. 4311, pp. 1–15.

[81] A. Křenek, J. Sitera, L. Matyska, F. Dvořák, M. Mulač, M. Ruda, and Z. Salvet, "glite job provenance – job-centric view," *Concurrency and Computation Practice and Experience*, vol. 20, no. 5, pp. 453–462, April 2008.

[82] D. Lanter, "Design of a lineage-based meta-data base for gis," *Cartography and Geographic Information Systems*, vol. 18, no. 4, pp. 255–261, 1991.

[83] ——, "Lineage in gis: The problem and a solution," National Center for Geographic Information and Analysis (NCGIA), University of California at Santa Barbara, Technical Report 90-6, 1991.

[84] J. Ledlie, C. Ng, and D. A. Holland, "Provenance-aware sensor data storage," in *Proceedings of the 21st International Conference on Data Engineering Workshops*, Tokyo, Japan, April 2005, pp. 1189–1193.

[85] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: a new architecture for high-performance stream systems," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 274–288, Aug. 2008.

[86] X. Li, B. Plale, N. Vijayakumar, R. Ramachandran, S. Graves, and H. Conover, "Real-time storm detection and weather forecast activation through data mining and events processing," *Earth Science Informatics*, vol. 1, no. 2, pp. 49–57, 2008.

[87] H.-S. Lim, Y.-S. Moon, and E. Bertino, "Research issues in data provenance for streaming environments," in *Proceedings of the 2nd SIGSPATIAL ACM GIS 2009 International Workshop on Security and Privacy in GIS and LBS (SPRINGL '09)*, Seattle, Washington, USA, 2009, pp. 58–62.

[88] ——, "Provenance-based trustworthiness assessment in sensor networks," in *Proceedings of the Seventh International Workshop on Data Management for Sensor Networks (DMSN'10)*, Singapore, 2010, pp. 2–7.

[89] Lojack.com. (2011, August) Lojack – a vehicle theft recovery system. [Online]. Available: http://www.lojack.com/

[90] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system: Research articles," *Concurrency and Computation: Practice and Experience - Workflow in Grid Systems*, vol. 18, no. 10, pp. 1039–1065, August 2006.

[91] L. Ma, D. Liang, Q. Zhang, X. Li, and H. Wang, "Load shedding for shared window join over real-time data streams," in *Proceedings of the Joint International Conferences on Advances in Data and Web Management (APWeb/WAIM '09)*, Suzhou, China, 2009, pp. 590–596.

[92] T. McPhillips, S. Bowers, and B. Ludäscher, "Collection-oriented scientific workflows for integrating and analyzing biological data," in *Proceedings of the International Workshop on Data Integration in the Life Sciences (DILS'06)*, Hinxton, UK, 2006, pp. 248–263.

[93] A. Melski, L. Thoroe, and M. Schumann, "Managing RFID data in supply chains," *International Journal of Internet Protocol Technology*, vol. 2, no. 3/4, pp. 176–189, December 2007.

[94] S. Miles, "Electronically querying for the provenance of entities," in *Provenance and Annotation of Data*, ser. Lecture Notes in Computer Science, L. Moreau and I. Foster, Eds. Springer Berlin / Heidelberg, 2006, vol. 4145, pp. 184–192.

[95] A. Misra, B. Falchuk, and S. Loeb, "Server-assisted context-dependent pervasive wellness monitoring," in *Proceedings of the 3rd International Conference on Pervasive Computing Technologies for Healthcare*, London, UK, April 2009, pp. 1–4.

[96] A. Misra, M. Blount, A. Kementsietsidis, D. Sow, and M. Wang, "Advances and challenges for scalable provenance in stream processing systems," in *Provenance and Annotation of Data and Processes*, ser. Lecture Notes in Computer Science, J. Freire, D. Koop, and L. Moreau, Eds. Springer Berlin / Heidelberg, 2008, vol. 5272, pp. 253–265.

[97] L. Moreau, P. Groth, S. Miles, J. Vazquez, J. Ibbotson, S. Jiang, S. Munroe, O. Rana, A. Schreiber, V. Tan, and L. Varga, "The provenance of electronic data," *Communications of the ACM*, vol. 51, no. 4, pp. 52–58, 2008.

[98] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. den Bussche, "The open provenance model core specification (v1.1)," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 743–756, June 2011.

[99] L. Moreau, J. Freire, J. Futrelle, R. E. Mcgrath, J. Myers, and P. Paulson, "The open provenance model: An overview," in *Provenance and Annotation of Data and Processes*, J. Freire, D. Koop, and L. Moreau, Eds. Springer Berlin / Heidelberg, 2008, vol. 5272, pp. 323–326.

[100] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, Boston, Massachusetts, USA, 2006, pp. 43–56.

[101] S. Munroe and S. Miles, "PrIMe: A methodology for developing provenance-aware applications," University of Southampton, Technical Report, 2006.

[102] S. Munroe, S. Miles, L. Moreau, and J. Vázquez-Salceda, "PrIMe: a software engineering methodology for developing provenance-aware applications," in *Proceedings of the 6th international workshop on Software engineering and middleware (SEM '06)*, Portland, Oregon, USA, 2006, pp. 39–46.

[103] L. Nachman, R. Kling, R. Adler, J. Huang, and V. Hummel, "The intel mote platform: a bluetooth-based sensor network for industrial monitoring," in *Proceedings of the 4th international symposium on Information processing in sensor networks (IPSN '05)*, Los Angeles, California, USA, 2005, pp. 437–442.

[104] National Gallery of Art Website. (2011, August) Provenance of the painting - roses. [Online]. Available: http://www.nga.gov/collection/gallery/vangogh/vangogh-72328-prov.html

[105] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[106] Oracle Corporation. (2011, August) Mysql - the world's most popular open source database. [Online]. Available: http://www.mysql.com/

[107] U. Park and J. Heidemann, "Provenance in sensornet republishing," in *Provenance and Annotation of Data and Processes*, ser. Lecture Notes in Computer Science, J. Freire, D. Koop, and L. Moreau, Eds. Springer Berlin / Heidelberg, 2008, vol. 5272, pp. 280–292.

[108] S. Reddy, G. Chen, B. Fulkerson, S. J. Kim, U. Park, N. Yau, J. Cho, and J. H. M. Hansen, "Sensor-Internet Share and Search—Enabling Collaboration of Citizen Scientists," in *Proceedings of the ACM Workshop on Data Sharing and Interoperability on the World-wide Sensor Web*, Cambridge, Massachusetts, USA, April 2007, pp. 11–16.

[109] C. Scheidegger, D. Koop, E. Santos, H. Vo, S. Callahan, J. Freire, and C. Silva, "Tackling the provenance challenge one layer at a time," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 473–483, April 2008.

[110] Y. Simmhan, B. Plale, D. Gannon, and S. Marru, "Performance evaluation of the karma provenance framework for scientific workflows," in *Provenance and Annotation of Data*, ser. Lecture Notes in Computer Science, L. Moreau and I. Foster, Eds. Springer Berlin / Heidelberg, 2006, vol. 4145, pp. 222–236.

[111] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *ACM SIGMOD Record*, vol. 34, no. 3, pp. 31–36, 2005.

[112] ——, "A survey of data provenance techniques," Computer Science Department, Indiana University, Technical Report, 2005.

[113] ——, "A framework for collecting provenance in data-centric scientific workflows," in *Proceedings of the IEEE International Conference on Web Services*, Chicago, USA, 2006, pp. 427–436.

[114] ——, "Query capabilities of the karma provenance framework," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 441–451, April 2008.

[115] Southampton City Council, "Port of southampton off-site reactor emergency plan," SotonSafe report, Version 4, 2006.

[116] D. M. Sow, L. Lim, M. Wang, and K. H. Kim, "Persisting and querying biometric event streams with hybrid relational-xml dbms," in *Proceedings of the 2007 inaugural international conference on Distributed event-based systems (DEBS '07)*, Toronto, Ontario, Canada, 2007, pp. 189–197.

[117] L. Spery, C. Claramunt, and T. Libourel, "A lineage metadata model for the temporal management of a cadastre application," in *Proceedings of the 10th International Workshop on Database & Expert Systems Applications (DEXA '99)*, Florence, Italy, 1999, pp. 466–474.

[118] ——, "A spatio-temporal model for the manipulation of lineage metadata," *Geoinformatica*, vol. 5, no. 1, pp. 51 – 70, 2001.

[119] U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS '04)*, Paris, France, 2004, pp. 263–274.

[120] M. Stonebraker, U. etintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42 – 47, 2005.

[121] StreamBase Systems Inc. (2011, September) Complex event processing: Streambase. [Online]. Available: http://www.streambase.com/

[122] Sun Microsystems Inc. (2011, August) Messaging systems and the java message service (JMS). [Online]. Available: http://java.sun.com/developer/technicalArticles/Networking/messaging/

[123] SunSPOT Project. (2011, August) Sun spot world: Program the world! [Online]. Available: http://www.sunspotworld.com/

[124] W.-C. Tan, "Research problems in data provenance," *IEEE Data Engineering Bulletin*, vol. 27, no. 4, pp. 45–52, 2004.

[125] ——, "Provenance in databases: Past, current, and future," *IEEE Data Engineering Bulletin*, vol. 30, no. 4, pp. 3–12, 2007.

[126] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager," in *Proceedings of the 29th international conference on Very large data bases - Volume 29 (VLDB'03)*, Berlin, Germany, 2003, pp. 309–320.

[127] D. Terry, D. Goldberg, D. Nichols, and B. Oki, "Continuous queries over append-only databases," in *Proceedings of the 1992 ACM SIGMOD international conference on Management of data (SIGMOD '92)*, San Diego, California, USA, 1992, pp. 321–330.

[128] The Apache Software Foundation. (2011, August) Apache activemq - the most popular and powerful open source message broker. [Online]. Available: http://activemq.apache.org/

[129] The World Wide Web Consortium. (2012, January) The PROV data model and abstract syntax notation. [Online]. Available: http://www.w3.org/TR/prov-dm/

[130] A. Vahdat and T. Anderson, "Transparent result caching," in *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '98)*, New Orleans, Louisiana, USA, 1998, pp. 3–3.

[131] N. Vijayakumar and B. Plale, "Towards low overhead provenance tracking in near real-time stream filtering," in *Provenance and Annotation of Data*, ser. Lecture Notes in Computer Science, L. Moreau and I. Foster, Eds. Springer Berlin / Heidelberg, 2006, vol. 4145, pp. 46–54.

[132] ——, "Tracking stream provenance in complex event processing systems for workflow-driven computing," in *Proceedings of the 2nd International Workshop on Event-driven Architecture, Processing, and Systems, in conjunction with VLDB'07 (EDA-PS'07)*, Vienna, Austria, 2007.

[133] N. N. Vijayakumar, "Data management in distributed stream processing systems," PhD. thesis, Indiana University, 2007.

[134] M. Wang, M. Blount, J. Davis, A. Misra, and D. Sow, "A time-and-value centric provenance model and architecture for medical event streams," in *Proceedings of the 1st ACM SIGMOBILE international workshop on Systems and networking support for healthcare and assisted living environments (HealthNet'07)*, San Juan, Puerto Rico, 2007, pp. 95–100.

[135] J. Widom, "Trio: A system for integrated management of data, accuracy, and lineage," in *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR'05)*, Asilomar, California, USA, 2005, pp. 262–276.

[136] A. Woodruff and M. Stonebraker, "Supporting fine-grained data lineage in a database visualization environment," in *Proceedings of the Thirteenth International Conference on Data Engineering (ICDE'97)*, Birmingham, UK, 1997, pp. 91–102.

[137] Y. Zhu and D. Shasha, "Statstream: statistical monitoring of thousands of data streams in real time," in *Proceedings of the 28th international conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, 2002, pp. 358–369.