An Inference-based Framework for Managing Data Provenance

MOHAMMAD REZWANUL HUQ

An Inference-based Framework for Managing Data Provenance

MOHAMMAD REZWANUL HUQ

Graduation Committee

Chairman and Secretary

Prof. dr. ir. A. J. Mouthaan	University of Twente, NL

Supervisor

Prof. dr. P. M. G. Apers	University of Twente, NL

Assistant Supervisor

Dr. A. Wombacher	University of Twente, NL

Members

Prof. dr. R. J. Wieringa	University of Twente, NL
Prof. dr. ir. M. Aksit	University of Twente, NL
Prof. dr. J. de Vlieg	Radboud University of Nijmegen, NL
Dr. P. Groth	VU University of Amsterdam, NL
Dr. L. P. H. van Beek	Utrecht University, NL

CTIT Ph.D. Thesis Series No. 13-258
Center for Telematics and Information Technology (CTIT)
P. O. Box 217, 7500 AE Enschede, The Netherlands

SIKS Dissertation Series No. 2013-27
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems

ISBN	978-90-365-0178-1
ISSN	1381-3617 (CTIT Ph.D. Thesis series No. 13-258)
DOI	10.3990/1.9789036501781
	http://dx.doi.org/10.3990/1.9789036501781

Cover Design	A. K. M. Shahidur Rahman
Printed by	Wöhrmann Print Service

AN INFERENCE-BASED FRAMEWORK FOR MANAGING DATA PROVENANCE

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof. dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Friday, November 01, 2013 at 16:45

by

Mohammad Rezwanul Huq
born on October 03, 1982
in Barisal, Bangladesh

This dissertation is approved by:

Prof. dr. P. M. G. Apers (supervisor)

Dr. A. Wombacher (assistant supervisor)

Dedicated to my Parents

for their unconditional love and endless support to me

ABSTRACT

Scientists can facilitate data intensive applications to study and understand the behavior of a complex system. In a data intensive application, a scientific model facilitates raw data products, collected from various sources, to produce new data products. Based on the generated output, scientists used to make decisions that could potentially affect the system which is being studied. Therefore, it is important to have the ability of tracing an output data product back to its source values if that particular output seems to have an unexpected value.

Data provenance helps scientists to investigate the origin of an unexpected value. Provenance could be also used to validate a scientific model. Existing provenance-aware systems have their own set of constructs to design the workflow of a scientific model for extracting workflow provenance. Using these systems requires extensive training for scientists. Preparing workflow provenance manually is also not a feasible option since it is a time consuming task. Moreover, the existing systems document provenance records explicitly to build a fine-grained provenance trace which is used for tracing back to source data. Since most of the scientific computations handle massive amounts of data, the storage overhead to maintain provenance data becomes a major concern.

We address the aforesaid challenges by introducing a framework managing both workflow and fine-grained data provenance in a generic and cost-efficient way. The framework is capable of extracting workflow provenance of a scientific model automatically at reduced effort and time. It also infers fine-grained data provenance without explicit documentation of provenance records. Therefore, the framework reduces the storage consumption to maintain provenance data. We introduce a suite of inference-based methods addressing different execution environments to make the framework more generic in nature. Moreover, the framework has the self-adaptability feature so that it can provide optimally accurate provenance at minimal storage costs. Our evaluation based on two use cases shows that the framework provides a generic, cost-efficient solution to scientists who want to manage data provenance for their data intensive applications.

SAMENVATTING

Wetenschappers gebruiken data intensieve toepassingen om het gedrag van complexe systemen te modelleren zodat ze deze systemen kunnen bestuderen en begrijpen. In een data intensieve toepassing wordt ruwe data, verzameld uit verscheidene bronnen, omgezet naar afgeleide data. Op basis van deze afgeleide data worden beslissingen genomen die het gemodelleerde systeem beïnvloeden. Het is hiervoor belangrijk dat het mogelijk is om de afgeleide data te herleiden naar zijn oorsprong, zeker als er een onverwacht resultaat bij zit.

Data provenance helpt wetenschappers om de oorsprong van een dergelijk onverwacht resultaat te vinden. Data provenance kan ook gebruikt worden om een wetenschappelijk model te valideren. Bestaande provenance-aware systemen hebben een eigen verzameling methoden om een workflow te ontwerpen waarin de data provenance van een wetenschappelijk model bijgehouden wordt. Het handmatig opzetten van een data provenance workflow is geen reële optie, omdat dit erg tijdrovend is. Daarnaast houden de bestaande provenance-aware systemen een expliciete, gedetailleerde provenance trace bij, welke gebruikt wordt voor het herleiden van de data. Omdat er in wetenschappelijke berekeningen grote hoeveelheden data omgaan wordt de overhead van opgeslagen provenance data een belangrijke kwestie.

We gaan in op de voorgaande kwesties, en introduceren een framework voor het omgaan met zowel workflow en gedetaileerde data provenance in een generieke en kosten-efficiënte wijze. Dit framework kan gebruikt worden om de workflow provenance van een wetenschappelijk model automatisch af te leiden, wat zowel tijd als moeite bespaart. Het framework is ook in staat om gedetailleerde provenance data af te leiden zonder daarvoor expliciete opslag nodig te hebben. Hierdoor verlaagt het framework de benodigde opslagruimte. We introduceren een geheel van inferentiegebaseerde methoden gericht op verschillende omgevingen om de generieke aard van het framework te versterken. De evaluatie is gebaseerd op twee use cases, welke tonen dat het framework een generieke, kosten-

efficiënte oplossing is voor wetenschappers die provenance data willen bijhouden in data intensieve toepassingen.

ACKNOWLEDGMENTS

Time flies!

On a bright, sunny afternoon on Summer 2009, I came to UT with a colorful dream - that someday I would complete my PhD. After more than four years, now I can feel that my dream will turn to reality very soon. And, this could never happen without the guidance and support I have got from a few faces.

How could I thank them? I am afraid that I do not have enough words in my dictionary to express my gratitude to them. They are my supervisors. Peter, thank you very much to believe in my capabilities. Without your ever encouraging comments especially in my early days here, I could not finish this job. Andreas, I learned how to do an independent research from you. Your care and guidance always kept me in the right direction. This thesis could remain incomplete without the brainstorming sessions we had together. Thank you so much.

I would also like to thank members of my graduation committee for accepting to be part of the committee and for taking their time and effort to read my thesis. Especially, I want to thank Paul for his detailed comments and suggestions that surely helped me to give the final touch to the thesis.

I had an opportunity to work with Rens and Yoshi from Utrecht University during a case study. They supported me in every possible way to conduct that case study. My warmest thanks to them. I am also honored to have Rens in my graduation committee. In this connection, I would also like to thank Bram from TNO Groningen since he introduced us to Rens and Yoshi. Recently, I have worked with Alessandra and Sean from DERI on another use case. Throughout the case study, they were very helpful explaining details on Answer Set Programming (ASP). Thanks to both of you. I would like to extend my thanks to Philipp from University of Zurich, who invited us for a short visit there to talk to scientists from different domains using Python programs. In this regard, I also want to thank Paul again to allow me to access Python scripts from 'Data2Semantics' project to verify my work.

My days in UT could not be more comfortable without my colleagues in the database group. I want to express my deepest gratitude to them. Maurice was always supportive and also helped me during my demonstration in EDBT'13. Djoerd and Maarten gave occasional tips and constructive criticisms to shape my work. Jan always assisted with technical issues. Ida and Suse were always at the office for helping out with so many things. Specially, Ida - you are the heartbeat of the DB group that holds us together. I will miss you. I would also like to thank Juan for being so supportive and helpful to me in those days with highs and lows. My sincere gratitude to Brend for translating the abstract of my thesis into Dutch. I also want to thank Eleftheria, my colleague from DIES group, with whom I shared my thoughts during those short breaks at the office. I also want to thank my direct colleagues over the years, particularly: Robin and Mena, and also: Kien, Dolf, Sergio, Lei, Victor, Mohammad, Zhemin, Iwe, Riham, Almer, Harold and Sander.

I would like to extend my thanks to my friends who made my stay in Enschede an enjoyable and a memorable one. Kallol Bhai, Mahua Bhabi, Antora, Tumpa, Reza Bhai, Dhrubo Bhai, Shawrav Bhai, Anupoma Apu, Zubair Bhai, Atik Bhai, Morshed, Rubaiya, Ashif and Hasib - many many thanks, my dear.

Getting myself into the highest level of education had never been possible without the sacrifice and support from my family. My father - a teacher, a freedom fighter and the most sincere person I have ever seen in my life, has always encouraged me to explore my potential and excel myself. My mother has sacrificed everything in her entire life to brought me up. My younger sister had taken care of my parents while I was in Enschede and this helped me to focus into my study. I am grateful to them for their unconditional love and support to me.

Last and foremost I want to thank my sweet wife Nitu. Nitu, I met you on Winter 2009 and you light up my life like the ever shining summer days. You are very understanding and caring. I am forever grateful for your support and patience especially during the final stage of my study. I am blessed to have you next to me.

Rezwan
October 6, 2013
Dhaka, Bangladesh.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

Scientists from many domains such as physical, geological, environmental, biological etc. facilitate data intensive applications to study and better understand these complex systems [100]. Most of these applications facilitate data fusion [78] which combines several sources of raw data to produce new data products. The data collection might contain both in-situ data collected from the field and data streams sent by sensors. Scientists might also facilitate geospatial data, i.e., measurements or sensor readings with time and space, from various sources. Scientists use this data, fitting into their model that describes processes in the physical world and as a consequence, scientists get the output, i.e., a data product, which is used to facilitate either a process control application or a decision support system. A new generation of information infrastructure, known as cyberinfrastructure, is being developed to support these data intensive applications [135].

The Swiss Experiment[1] is an example of such type of cyberinfrastructures, providing a platform to enable real-time environmental experiments. One of the experiments in the context of this platform is to study how river restoration affects water quality. To perform this experiment, scientists, first, design their scientific model which facilitates sensor readings of electrical conductivity (input data products) in a known region of the river to produce interpolated values of electrical conductivity (output data products) over the same region. Afterward, scientists execute the model to generate the result set. Based on this generated result, they could make a decision to control a nearby drinking water well to prevent the drinking water quality being compromised by a flood.

1 Available at `http://www.swiss-experiment.ch/`

One of the requirements of this cyberinfrastructure is the ability to trace the origin of an output data product. This could be useful in cases of the generation of any imprecise or unexpected data product during the execution of a scientific data processing model. To investigate the origin of the unexpected data, scientists need to debug their models used for actual processing as well as to trace back values of the input data sources.

Furthermore, reproducibility of data products is another major requirement in the scientific domain. Reproducibility of data products refers to the ability to produce the same data product using the same set of input data and model parameters irrespective of the model execution time. It allows scientists to validate their own model and to justify the decision made based on the data products. Maintaining data provenance [20, 114], also known as lineage [80], allows scientists to achieve these requirements and thus, leading towards the development of the provenance-aware cyberinfrastructure.

1.1 DATA PROVENANCE

Provenance is defined in many different contexts. One of the earlier definitions was given in the context of geographic information system (GIS). In GIS, data provenance is known as lineage which explicates the relationship among events and source data in generating the data product [80]. In the context of database systems, data provenance provides the description of how a data product is achieved through the transformation activities from its input data [20]. In a scientific workflow, data provenance refers to the derivation history of a data product starting from its origin [114]. In the context of the geoscientific domain, geospatial data provenance is defined as the processing history of a geospatial data product [135].

In all contexts, provenance can be defined at different levels of granularity [19]. Fine-grained data provenance is defined at the value-level of a data product which refers to the determination of how that data product has been created and processed starting from its input values. It helps scientists to trace the value of an output data product. Fine-grained data provenance could be facilitated to have reproducible results as well. Coarse-grained or workflow provenance is defined at the more higher level of granularity. It captures association among different activities within the model at design time. Workflow provenance can achieve reproducibility in

a few cases where data is collected beforehand, i.e., offline data or data streams arriving at a fixed rate without any late arrivals. In other cases of data streams which have more time-related assumptions like variable data arrival rate, workflow provenance itself cannot achieve reproducibility due to the creation of new data products and update of existing data products during the model execution. However, based on the workflow provenance of a model, we can infer fine-grained data provenance which can significantly reduce storage overhead for provenance data. Therefore, a framework integrating both workflow and fine-grained data provenance will be proven beneficial to scientists using provenance data.

1.2 GOAL OF THIS THESIS

In this thesis, we aim to develop a framework managing both workflow and fine-grained data provenance for data intensive scientific applications. To accomplish such a framework, we identify three key design factors. Firstly, the framework should be *generic*, i.e., applicable for any given model. The biggest challenge to make the framework generic in nature is to address different types of developing approach as, i.e., with or without facilitating any specific tools, as well as to address different types of coordination scheme within a model (e.g. data-flow or control-flow) [111]. Secondly, the framework should be *cost-efficient*, i.e., manage data provenance with minimal user effort in terms of time and training as well as at reduced storage costs. Maintaining provenance information by facilitating a particular platform might be time consuming because of training sessions arranged for users to make them understand the basic constructs of the platform used. Moreover, the explicit documentation of data provenance incurs storage overhead because of storing the relationship between input and output data products at each execution of the model for all the associated processing steps including the intermediate ones. The storage overhead might be further increased if a particular input data product contributes to produce several output data products. Therefore, the framework should manage data provenance at reduced cost in terms of time, training and storage consumption. Finally, it is important for the framework to address not only the characteristics of a given model like model developing platform, model coordination scheme etc. but also the characteristics of the associated data products and the execution such as the arrival pat-

tern of data products, the time required to process data etc. Because, each model could have variations in the aforesaid characteristics, also referred to as system dynamics. Considering the system dynamics of a given model, the framework should be capable of managing both workflow and fine-grained data provenance, which is referred to as the *self-adaptable* nature of the framework. The self-adaptability of the framework should analyze characteristics of the given model and underlying system dynamics and based on this analysis, the framework would choose the most suitable approach to manage data provenance. Accomplishing a framework with these key properties requires us to closely examine the complete problem domain, i.e., entities involved in a data intensive scientific application.

1.3 COMPLETE PROBLEM SPACE

At the beginning of this chapter, we described an example of a scientific model that was facilitated to control a drinking water well. First, scientists designed the model and then executed the model to produce the result set. Based on this example, we can characterize the problem space into two phases: design phase and execution phase. Figure 1.1 depicts different entities pertinent to a scientific model both at design and execution phase, represented by rectangles. Figure 1.1 also shows examples of different scientific models based on their characteristics, represented by round-shaped boxes. The entities defined during the design phase of a scientific model are: i) the scientific model itself and ii) different activities within the model. These two entities are represented by the top two rectangles in Figure 1.1. The entities involved during the execution phase of a scientific model are represented by the bottom two rectangles shown in Figure 1.1. Each activity defined at the design phase instantiates a corresponding processing element during the execution of a model and these processing elements process input data products and produce output data products. Data products have different characteristics with regard to their access and availability. We discuss different characteristics of these entities below.

1.3.1 Design Phase Characteristics

During the design phase, scientists define the model which is based on different activities, i.e., atomic units of work performed as a whole [1]. In

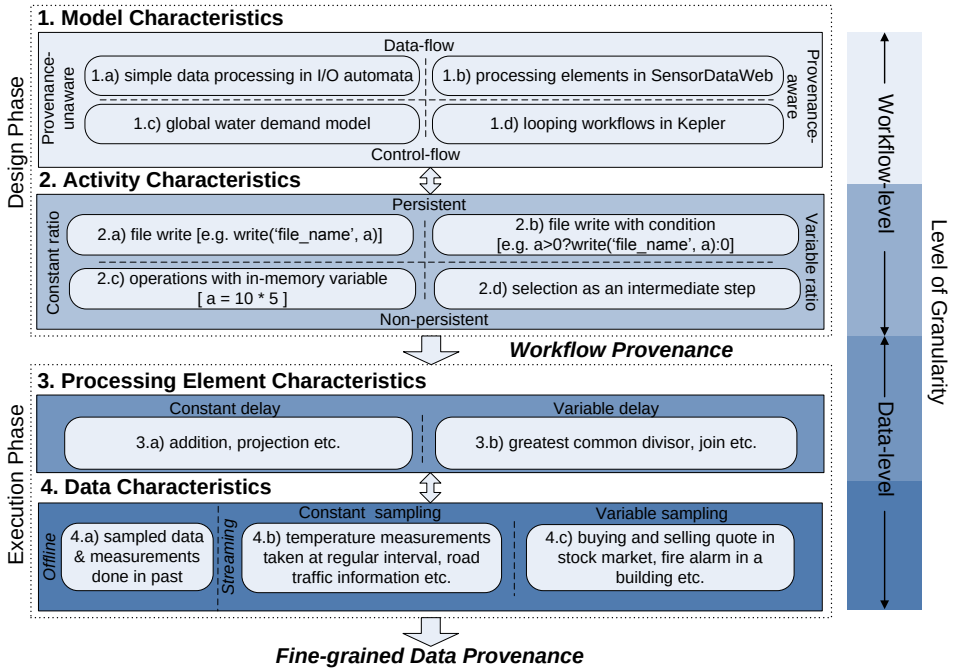


Figure 1.1: The problem space showing different characteristics of a scientific model

case the scientific model is specified in a *provenance-aware* platform [92], the provenance information is automatically acquired. Examples of the platforms where provenance awareness has been considered are scientific workflow engines such as Taverna [102], VisTrails [24], Kepler [84], Karma2 [116], Wings/Pegasus [77], stream data processing or complex event processing engines like SensorDataWeb[2], STREAM [8], Aurora [2], Borealis [3], or Esper[3]. Provenance has been considered in these platforms because they are targeted towards particular applications where provenance plays an important role.

Model characteristics

In case the scientific model is specified in a *provenance-unaware* platform, the provenance information must be maintained manually by the user. This requires training of the user and a significant effort in manually documenting provenance information. Examples of provenance-unaware platforms are general-purpose programming languages such as Python, generic data processing tools such as Microsoft Excel, R, MATLAB etc.

2 Available at `https://sourceforge.net/projects/sensordataweb/`

3 Available at `http://esper.codehaus.org/`

The second dimension of classifying scientific models is based on the underlying coordination approach of the model (e.g. data-flow or control-flow) [111]. In control-flow coordination the execution of an activity depends on the successful completion of the preceding activity. This paradigm is used in many programming languages that a statement/activity can only be executed after the previous statement has been completed. It also applies to many workflow models and logical formulations. As a contrast, in data-flow coordination the execution of an activity depends on the availability of data. The execution of an activity produces again data, which may trigger the execution of other activities. This paradigm is used in stream data processing and complex event processing engines as well as in models used in distributed systems research such as I/O automata [86].

These different dimensions of categorizing scientific models are represented by the first rectangle from the top in Figure 1.1. The rectangle is divided into 4 quadrants where each quadrant has specific characteristics. The round-shaped boxes inside the rectangle contains an example of a scientific model having characteristics of that corresponding quadrant.

Activity characteristics

The distinctions between model's developing platform like provenance-aware vs. provenance-unaware as well as model's underlying coordination approach like control-flow vs. data-flow describe the characteristics of a scientific model and classify them accordingly. In addition to this, further classifications are required on the activity-level, which is represented by the second rectangle from the top in Figure 1.1. Several activities comprise a scientific model. There are two important characteristics of an activity that need to be documented to help scientists finding and understanding the origin of a data product during execution phase. One of them is *input-output ratio*. The *input-output ratio* [70] refers to the ratio between the number of contributing input data products producing output to the number of produced output data products. Depending on activities, it can be either variable (e.g. *select* in a database) or constant (e.g. *project* in a database). The *input-output ratio* is required to establish data dependencies between contributing input and output data products. The other important characteristic of an activity indicating the availability of produced data product by that particular activity, is referred to as *IsPersistent*. The *IsPersistent* property describes whether the data product produced by an activity is stored persistently into a file/database or not.

Documenting the *input-output ratio* and the *IsPersistent* characteristics and potentially the other characteristics of the activities during the de-

sign phase explicated in the workflow provenance helps to understand the working mechanism of the activities which in turn is required to infer fine-grained data provenance. These different characteristics of classifying different activities are represented by the second rectangle from the top in Figure 1.1. The round-shaped boxes inside the rectangle contains an example of an activity having characteristics of that particular quadrant.

The documented characteristics and the relationship between activities during the design phase results into the workflow provenance of the scientific model. While the workflow provenance is acquired automatically in a provenance-aware platform, this must be done manually in a provenance-unaware platform. However, there is a demand in the scientific community to capture workflow provenance automatically in a provenance-unaware platform such as a programming/scripting language [6]. To accomplish this, the challenge is to transform control dependencies between activities into data dependencies by interpreting and analyzing the code. That is to transform a control-flow statement (e.g. function call) into an activity or a group of activities which only exhibits data dependencies.

Workflow provenance

Please note that in different scientific models, activities have different granularities ranging from complex operations to a single arithmetic operation. While the granularity of the activities is not influencing the provenance acquisition, it is influencing the complexity of the acquired provenance information and the interpretation by the user.

1.3.2 Execution Phase Characteristics

The entities involved during the execution phase are: i) processing elements and ii) data. These entities and their characteristics are shown using the bottom two rectangles in Figure 1.1.

An activity defined in the design phase instantiates a corresponding processing element during the execution phase. Processing elements have variations in their processing delay, i.e. amount of time required to process input data products. As an example, processes performing addition or projections have constant processing delays, referred to as *constant delay* processing elements. Alternatively, executing some processing elements such as performing a join in a database or calculating the greatest common divisor, require different amount of time at each execution. Because, the execution of these processing elements depends on the number of input data products considered by the processing elements or the number of iterations

Processing element characteristics

needed to perform the operation successfully. These are referred to as *variable delay* processing elements. The third rectangle from the top in Figure 1.1 shows the different dimensions of classifying processing elements and the corresponding examples in round-shaped boxes within the rectangle.

Data characteristics

Independent of processing elements characteristics, the contributing data also exhibits its own characteristics. Data might arrive continuously (e.g. data streams) or can be collected before the execution begins (e.g. offline data). Data streams might have different data arrival patterns. Data tuples arriving at regular intervals are referred to as *constant sampling* data (e.g., temperature measurements sent at regular intervals). On the other hand, data might also arrive at an irregular interval such as buying and selling quotes on an instrument in a stock market. These are referred to as *variable sampling* data. The different characteristics of data products along with the examples are depicted by the bottom most rectangle in Figure 1.1.

Fine-grained data provenance

The relationships between data and processing elements during the execution phase are essentials to derive the fine-grained data provenance of a scientific model [19]. Existing work documents fine-grained data provenance explicitly in a database, also known as the annotation-based approach [21, 131, 109, 58, 108]. These approaches require a considerable amount of storage to maintain fine-grained data provenance especially if a single input data product contributes several times, producing multiple output data products. Sometimes, the size of provenance data becomes a multiple of the actual data. Since provenance data is 'just' metadata and less often used by end users, explicit documentation of fine-grained provenance seems to be infeasible and too expensive [64, 69]. One of the potential solutions to overcome this problem is to infer fine-grained data provenance based on the given workflow provenance and timestamps of data products. Therefore, inferring the fine-grained data provenance can make the complete framework *cost-efficient* in terms of storage consumption.

Like the representation of the workflow provenance based on the complexity of the associated activities, fine-grained data provenance might also be represented based on the different levels of granularity of the associated data products. In a particular scientific model, a data product might represent a data tuple in a relational database while in an other scientific model, a data product can represent a file in the physical memory. Though the inference of fine-grained data provenance is not influenced by the granularity of the data, the semantics of the fine-grained data provenance must be interpreted by the user.

Developing an inference-based framework to manage both workflow and fine-grained data provenance requires attention to the underlying environment along with the system dynamics including processing element and data characteristics. The inference mechanisms should take variation in the used platform, processing delay and data arrival pattern into consideration to infer highly accurate provenance information. To accomplish that, *self-adaptability* of the framework is required which can decide when and how to execute the most appropriate inference-based methods based on a given scientific model and its associated data products.

1.4 RESEARCH QUESTIONS

Based on the problem space described in Section 1.3, we need to answer the following primary research question which is in fact the center of investigation in this thesis.

> *Primary Research Question (RQ): How to manage data provenance with minimal user effort in terms of time and training and at reduced storage consumption for different kinds of scientific models?*

Our goal is to develop a framework for managing data provenance to satisfy the primary research question. To accomplish such a framework, first, we need to ensure that the workflow provenance of a scientific model which has been designed and developed in a provenance-unaware platform reported in Section 1.3.1, is captured automatically. This automatic capturing of workflow provenance ensures that the envisioned framework is applicable to any scientific model and thus the framework would be *generic*. Furthermore, acquiring workflow provenance automatically reduces user effort in terms of time and training. Therefore, one of the research questions to be satisfied to achieve such a framework is:

> *RQ 1: How to capture automatically the workflow provenance of a scientific model developed in a provenance-unaware platform at reduced cost in terms of time and training?*

The automatically captured workflow provenance of a scientific model provides an overview on the relationship between different activities within the model. However, it does not represent the provenance information produced during the execution of that scientific model. Therefore, we need

a mechanism incorporated into the framework that can manage provenance information produced at the tuple-level, i.e., the relationship between data products, also referred to as fine-grained data provenance. Fine-grained provenance information could become a multiple of actual data products due to the multiple processing of the same input data product producing multiple output data products. The envisioned framework should be able to reconstruct the fine-grained data provenance at reduced storage consumption. Therefore, the next research question to accomplish a provenance-aware framework is:

> *RQ 2: How to infer fine-grained data provenance under different system dynamics at reduced cost in terms of storage consumption?*

The envisioned provenance-aware framework should be a *self-adaptable* system as described in Section 1.3.2. The self-adaptability allows the complete framework applicable to any given model with variant system dynamics such as processing delays, data arrival pattern etc. It ensures that the provenance-aware framework always provides the optimal provenance information. Based on this requirement, we formulate the last research question which is given below.

> *RQ 3: How to incorporate the self-adaptability into the framework managing data provenance at reduced cost?*

Satisfying these three research questions leads us to develop a *generic*, *cost-efficient* and *self-adaptable* inference-based framework which in turn can satisfy the primary research question of this thesis.

1.5 RESEARCH DESIGN

The research in this thesis has three phases: i) problem investigation, ii) solution design and iii) solution validation. The research design is depicted in Figure 1.2. The research phases are represented by the rectangles where as the actions taken in that phase are shown by the round-shaped boxes within the particular rectangle.

Firstly, we start by sketching the complete problem space discussed in Section 1.3 followed by extensive literature study to facilitate a thorough problem investigation phase. Based on the problem space and existing literature, we formulate the key design factors the envisioned framework should comply with.

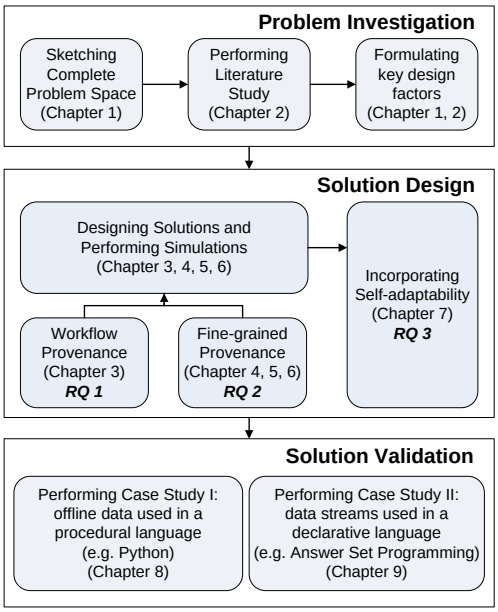


Figure 1.2: Research phases and corresponding actions in the context of this thesis

Secondly, we design the methods based on the research questions and solution criteria identified in the problem investigation phase. Our proposed methods address all aforesaid research questions. We develop a technique to capture workflow provenance automatically in a provenance-unaware platform. Furthermore, we propose several algorithms to infer fine-grained data provenance under variable system dynamics. All of these inference-based methods are capable of managing provenance in diverse situations and at reduced costs in terms of time, training and storage consumption. Finally, we introduce self-adaptability into the framework so that the framework itself can decide autonomously which method to apply under a given environment. In the solution design phase, we also simulate the proposed methods to evaluate their performance in general.

Finally, we validate the proposed inference-based framework by conducting two case studies with different characteristics. One of them is a scientific model written in Python, handling offline data while the other is a model written in Answer Set Programming (ASP), dealing with data streams. To demonstrate the case studies, we implement the methods and techniques designed during the solution design phase and develop the framework as a stand-alone tool in Java. The applicability of the proposed framework to these scientific models supports the claim that our frame-

work is generic. Furthermore, it can capture and infer provenance information at reduced time, training and storage consumption.

1.6 THESIS CONTRIBUTIONS

The primary contribution of this thesis is to develop a framework that manages both workflow and fine-grained data provenance for data intensive scientific models at reduced costs in terms of time, training and storage consumption. The primary contribution is realized by achieving the following contributions satisfying the research questions mentioned in Section 1.4.

- *Capturing workflow provenance*: In this thesis, we propose a novel technique to capture workflow provenance automatically based on a given program which is used for actual processing. This overcomes the difficulties with collecting workflow provenance automatically for a model developed using a provenance-unaware platform such as any procedural or declarative language. The proposed technique also captures workflow provenance with reduced effort in time and training compared to the manual documentation. This technique of automatic capturing of workflow provenance satisfies *RQ 1*. Since there are many programming and scripting languages and each has its own set of programming constructs and syntax, we showcase our approach using Python programs. Python is widely-used to handle spatial and temporal data in the scientific community as well as in commercial products such as ArcGIS[4].

- *Inferring fine-grained data provenance*: We also propose several fine-grained provenance inference methods to infer fine-grained data provenance in a *cost-efficient* way in terms of storage consumption compared to the explicit fine-grained provenance documentation technique. The proposed inference-based methods are applicable to a variety of scientific models under different system dynamics discussed in Section 1.3.2. As an example, one of our proposed fine-grained provenance inference method is better suited to the systems handling offline data and having constant processing delay while another method is most appropriate to the systems processing data

4 Available at `http://www.esri.com/software/arcgis`

streams and having variable processing delays. We discuss the basic principle and the applicability of each of these methods which infers fine-grained data provenance based on the given workflow provenance of the model and the timestamps associated with data products. These inference-based methods satisfy *RQ 2*.

- *Introducing a self-adaptable framework*: Furthermore, to accomplish a *self-adaptable* framework, we introduce a decision tree which is used during the execution of a scientific model facilitating the proposed framework to decide the most appropriate fine-grained provenance inference method based on the underlying system dynamics. The self-adaptability feature dynamically decides per activity within the model on how to record and infer fine-grained provenance information based on the observed system dynamics, i.e., data products arrival pattern, processing delay etc. The outcome of the decision tree allows the framework to be self-adaptable and thus satisfies *RQ 3*.

In sum, we propose an inference-based framework to manage both workflow and fine-grained data provenance for a variety of data intensive scientific applications. Our proposed framework is applicable to any type of model, confirming its *generic* nature. Moreover, the framework is *cost-efficient* in terms of time, training and storage. It is also *self-adaptable* which copes with variant system dynamics such as input data products arrival pattern, processing delay etc. Therefore, the proposed framework addresses all three key design factors mentioned in Section 1.2.

We evaluate the proposed framework based on two use cases. One of them involves a scientific model for estimating the global water demand [127]. This model includes offline geospatial data and is developed using Python. The other case study is about estimating the degree of accessibility of a particular road segment. The scientific model is developed using a declarative language - Answer Set Programming (ASP). This model processes data streams collected from various sources like twitter, rss feeds etc. One of the key differences between these two models is the former provides deterministic results while the later generates non-deterministic result sets. In both cases, the framework can capture workflow provenance and infer fine-grained data provenance. Therefore, the evaluation demonstrates the applicability and suitability of the proposed framework in a scientific data processing model which leads to the conclusion that the framework satisfies *Primary Research Question* of this thesis.

1.7 THESIS STRUCTURE

The remainder of this thesis is structured as follows. In Chapter 2, we discuss the existing systems and research along with their pros and cons. Based on this discussion, we conclude that the key design factors reported in Section 1.2 should be addressed by the envisioned framework.

Chapter 3 presents the mechanism of capturing workflow provenance automatically from a given Python program. This chapter addresses *RQ 1* as shown by the Figure 1.3. In Chapter 4, 5 and 6, we explain the inference-based methods to infer fine-grained data provenance under variant system dynamics. Chapter 4 presents the method to infer fine-grained data provenance which is suitable for an environment where activities have constant processing delays and data products arrive at a regular interval. In Chapter 5, the proposed method is better suited to the systems processing data streams with variant system dynamics. This method is extended in Chapter 6, where we explain an inference-based mechanism that infers fine-grained data provenance for the complete workflow, i.e., multiple processing steps, with variant system dynamics. All these chapters address *RQ 2* as depicted in Figure 1.3. Chapter 7 explains the mechanism of incorporating self-adaptability into the framework. This chapter addresses *RQ 3* as shown in Figure 1.3.

To validate the framework, we perform two case studies discussed in Chapter 8 and 9. The characteristics of these case studies based on Section 1.3 are quite different from each other which helps us to demonstrate the wide applicability of the framework. At last, in Chapter 10, we summarize the contributions of this thesis based on the research questions, posed in Section 1.4, followed by a discussion on future research directions in the context of this thesis.

Research Questions \ Chapters	Chapter 1	Chapter 2	Chapter 3	Chapter 4	Chapter 5	Chapter 6	Chapter 7	Chapter 8	Chapter 9	Chapter 10
RQ 1			√					√	√	
RQ 2				√	√	√		√	√	
RQ 3							√	√	√	

Figure 1.3: Research questions related to chapters

2

RELATED WORK

The goal of this thesis is to develop a provenance-aware framework that can infer both workflow provenance and fine-grained data provenance in a cost-efficient manner. Therefore, data provenance is the core concept of this thesis. As a consequence, it is required to study existing research and systems in different dimensions of provenance to point out and emphasize the key criteria of the envisioned framework managing data provenance.

Figure 2.1 depicts the way we structure the existing research in the field of provenance. We categorize the existing research and systems at different dimensions of provenance such as provenance collection methods, provenance representation and sharing techniques as well as provenance applications, shown in Figure 2.1. A lot of attention has been paid to design and develop provenance-aware platforms in scientific workflow systems as discussed in surveys conducted by Simmhan et al. [114] and Davidson et al. [32]. Provenance-aware platforms have been also built in the context of database systems as discussed in [119, 27]. Furthermore, several studies on provenance for stream data processing have been undertaken. Recently, Moreau has investigated provenance in the context of Web [94]. There also exists provenance-aware platforms, developed specifically for a particular application domain or language. A comprehensive overview of provenance systems primarily focusing on the e-science domain is presented in [17]. Provenance systems targeting a specific programming language or data processing tools has also been built. In these aforesaid platforms, provenance has been collected at different levels of granularity depending on its target application and user [19].

After collecting provenance information, a provenance-aware system has to represent this information. Moreover, interoperability of provenance information has to be ensured to allow seamless sharing of provenance in-

formation between different systems. In the context of geographic information systems (GIS), one of the earliest application domain of provenance, there are a few existing work to represent provenance data for geographic information and services. Recently, a World Wide Web Consortium (W3C) family of specifications has been proposed for provenance representation and sharing.

Provenance information has been facilitated for a number of reasons [114]. Provenance can be used for auditing purposes such as monitoring resource consumption, error tracing etc. It could be also facilitated to validate a scientific model. Moreover, provenance information can be used as a replication recipe of output data products, produced by a scientific experiment. Very recently, Netherlands eScience center published a white paper on 'data stewardship' [33], referring to the practice of preserving data to ensure reproducibility and to also stimulate more data-driven research where provenance can play an important role. Another recent study [74] has shown that provenance information can be also used for debugging a scientific data processing model.

Chapter structure

In this chapter, we present a review of existing research and systems that capture provenance in different domains. Furthermore, we provide a brief discussion on techniques used for provenance representation and interoperability. We also highlight applications of provenance specially for debugging purpose by describing existing work in this direction. Based on this literature review, we emphasize a few points that should be considered to develop the envisioned framework inferring data provenance, which is in the center of investigation of this thesis.

2.1 PROVENANCE COLLECTION

The bottom part in Figure 2.1 shows existing provenance systems in different domains. In this section, we review the existing work in these domains that apply different methods and techniques to collect provenance, defined at different levels of granularity.

2.1.1 Provenance in Scientific Workflow Engines

Much of the research in provenance has come into the light from scientific workflow communities. Provenance has been studied from a wide an-

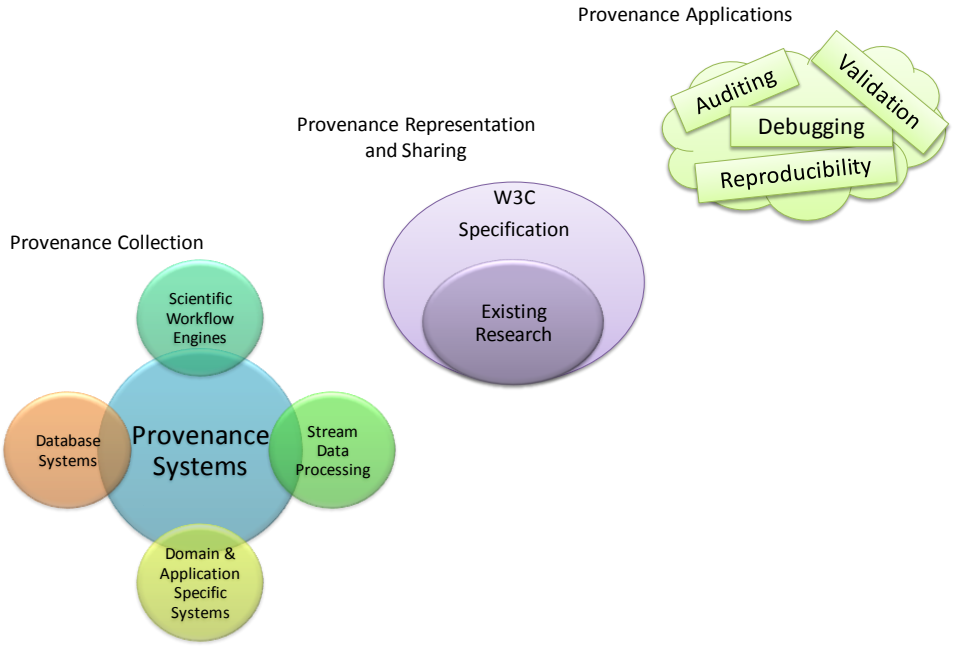


Figure 2.1: Existing research and systems in different dimensions of provenance

gle of perspectives including collection, representation, application-specific methods in the context of a scientific workflow engine. In this section, we discuss existing research and systems in this domain.

A workflow management system (e.g. Kepler [84], Taverna [102], VisTrails [24]) defines and manages a series of activities within a scientific data intensive experiment to produce an output data product. In such systems, activities create a processing chain and each activity takes input data products from a previous activity, i.e., data-driven workflows. Business workflows are different from scientific workflows [85]. Business workflows provide a common understanding of business processes that involve different persons and various information systems. It can serve as a blueprint for implementing the process. While scientific workflows mainly focus on derivation of data and in these kind of systems data processing activities are treated as *black boxes*, hiding details of data transformations [32].

Kepler is a scientific workflow management system for designing, executing, reusing, evolving, archiving, and sharing scientific workflows [84]. *Kepler*
Kepler provides process and data monitoring, provenance information and high speed data movement solutions. The Kepler system principally targets the use of a workflow metaphor for organizing computational tasks

that are directed towards particular scientific analysis and modeling goals. Thus, Kepler scientific workflows generally model the flow of data from one step to another in a series of computations that achieve some scientific goal. Several extensions of Kepler have been proposed and implemented to support provenance in different domains [75, 13]. Jararweh et al. have exploited the open-source features of Kepler system and have created customized processing models in order to accelerate and automate the experiments in ecosystems research [75]. In [13], authors have presented an extension to Kepler system to support streaming data, originating from environmental sensors. They have analyzed and archived data from observatory networks using distinct use cases in terrestrial ecology and oceanography.

Karma2

Karma2 provenance framework was developed to document provenance of data products produced by scientific workflows in a service-oriented architecture [115, 116]. Two forms of provenance are collected in Karma2 - workflow provenance and data provenance (fine-grained). Workflow provenance describes execution of workflows and invocations of associated services while data provenance explains the derivation of a data product, including input data products and associated activities/data transformations.

Taverna

In the life science domain, the Taverna project has developed a powerful, scalable tool for designing and executing bio-informatics workflows [102, 68]. The Taverna workbench includes the ability to monitor the running of a workflow and to examine the provenance of the data produced. In Taverna, recorded provenance information includes technical metadata explaining how each activity has been performed. In addition, start and end time of an activity as well as a description of the service operation used, are also recorded.

Barga et al.

Barga et al. have proposed a mechanism for capturing provenance information in scientific workflows [11]. In this study, authors have argued that a single representation of provenance cannot satisfy all existing provenance queries used in these kind of systems. Therefore, authors introduced a provenance model supporting multiple levels of provenance representation [12]. The different layers represent provenance information collected during both design and execution phase of a scientific workflow, i.e., provenance at different granularity levels. Therefore, scientists can comfortably deal with complexity and size of provenance information by facilitating this multi-layered provenance model.

VisTrails [24, 42] builds on a similar idea of multi-layered provenance representation presented in [11, 12]. In VisTrails, provenance information is captured for various stages of evolving workflows and their data products. VisTrails not only records intermediate results produced during workflow execution, but also records the operations/activities that are applied to the workflow. It documents the modification of workflows, as for instance adding or replacing activities/modules, deleting activities and setting parameters to an activity, by tracking the steps followed by users. Therefore, VisTrails can ensure reproducibility of scientific computations and can provide support for the layered-based tracking of workflow evolution. *VisTrails*

Kim et al. proposed another multi-layered provenance capturing mechanism in large-scale scientific workflow systems [77]. This approach is implemented in the Wings/Pegasus framework [34, 55]. It documents provenance information at different levels of granularity. "Application-level provenance" describes data-driven relationship among activities while "execution provenance" represents provenance information gathered during the execution of workflow which includes intermediate data, details on data transformations etc. *Wings/Pegasus*

Recently, Buneman et al. have proposed a hierarchical model on provenance information and have also demonstrated how this hierarchical structure can be derived from the execution of programs in *ProvL* programming language that describes the workflows [22]. *ProvL* is a functional language which can be used to express simple workflows. However, *ProvL* cannot handle the concept of streaming and concurrency in workflows. *ProvL*

Another study in the area of provenance-aware scientific workflow systems is Provenance Aware Service Oriented Architecture (PASOA) [62, 63]. PASOA builds an infrastructure for recording and reasoning over provenance in the context of e-Science. PASOA is designed to support interactions between loosely-coupled services. In this study, the idea of decomposing process documentation, i.e., what actually happened at execution time, has been proposed to record provenance information efficiently. Each part of the process from the whole process documentation is defined as a p-assertion. By capturing different types of p-assertions such as content of messages (interaction p-assertions), causal relationships between messages (relationship p-assertion) and the internal states of services (service state p-assertions), scientists can analyze an execution, validate it or compare it with other executions. Based on this idea, the Provenance Recording for Services (PReServ) [65] software package has been developed. This imple- *PASOA*

mentation allows developers to integrate recorded process documentation into their applications.

Annotation-based framework

In the area of scientific data management, authors proposed annotation-based provenance framework [87, 5]. This framework is implemented on the top of Kepler workflow management systems [84], representing provenance of scientific workflows. In this framework, each activity/module takes collections of data as an input and produces output collections by adding new computed data to the data structure it received. Output collections are annotated with explicit data dependency information to allow the framework to trace provenance of scientific data products. An extension of this framework described in [18] has introduced a solution to minimize size of documented provenance information by allowing annotations on collections to cascade to all descendant elements.

Summary

Based on the aforesaid discussion, we can conclude that scientific workflow engines capture provenance at different levels of granularity. Provenance information can be used not only for explaining the origin of output data products but also for debugging and troubleshooting the workflow and its execution. The existing solutions capture data-driven relationship among activities. Some of them proposed multi-layered provenance representation to allow users to deal with the complexity and large size of provenance information [115, 11, 24, 77]. Existing provenance-aware scientific workflow systems require scientists to learn basic constructs of a particular workflow management system and design the scientific experiment accordingly which is time consuming and also need substantial training for scientists. Furthermore, some of these scientific workflow engines are developed for a particular domain such as Taverna [102, 68] addresses bioinformatics workflows only. Developing a *generic* provenance management framework, i.e., applicable to any given scientific model/experiment, and a *cost-efficient* one also in terms of time and required training would be beneficial to the scientific community.

2.1.2 Provenance in Database Systems

A considerable research effort has been made by the database community to manage data provenance. Data provenance can be defined at different granularity levels (e.g. relation or tuple). Furthermore, data provenance has been categorized based on the type of queries (e.g. why, where, how) it can satisfy. Different techniques have been proposed to generate data

provenance in the context of a database system. Collected provenance information has been also represented using various techniques. In this section, we provide a review on existing research focusing provenance in database systems.

Types of data provenance

In the context of database systems, data provenance provides the description of how an output data product is achieved through the transformation activities from its input data [20]. In [20], Buneman et al. have also described a data model to compute provenance on both relations (coarse-grained) and tuples (fine-grained) level. In this data model, the location of any piece of data can be uniquely described by a path. Furthermore, in this study, authors have also drawn a distinction between *why-provenance* and *where-provenance*. *Why-provenance* determines the input data tuples which contributed to produce an output data tuple. This type of provenance is studied in [133, 29]. On the other hand, *where-provenance* identifies locations in the source database from where data products were extracted. An annotation-based approach has been proposed to address where-provenance [21]. In this approach, annotations associated with tuples in the source database can be propagated to the output database based on where data products are copied from.

Generation techniques

Provenance of a particular data product could be generated following two approaches: *lazy* approach and *eager* approach. In the *lazy* approach, provenance information is generated on demand based on a user request [133, 29]. On the other hand, in the *eager* approach, provenance information is propagated from one activities to another during execution of a scientific experiment [21, 15, 28, 60, 61, 56]. Next, we discuss these existing work.

Weak inversion

Woodruff et al. [133] have proposed the notion of *weak inversion* and *verification* to generate data lineage (provenance). Based on a given output data product, *weak inversion* functions regenerate input data products that contributed to produce that given output. However, the set of contributing input data products returned by the inverse function is not guaranteed to be perfectly accurate. Therefore, a separate *verification* function is required to examine the answer produced by weak inversion function. One pitfall of this technique is that not all functions are invertible. Therefore, this technique has limited applicability.

Structural analysis

Another work falling in the class of *lazy* approach of provenance generation is the study reported in [30, 29]. In this study, Cui et al. have presented an algorithm for lineage tracing in a data warehouse environment. This generic algorithm automatically generates lineage (provenance) data

through analyzing view definitions and algebraic structures of queries. The algorithm generates data provenance on tuple level and allows users to trace the lineage of a data product to its contributing input data products.

Based on the work mentioned in [30, 29], Trio, a database system managing not only data but also the accuracy and the lineage of data is proposed in [131, 4]. Trio introduces an integrated technique, combining both lazy and eager approaches, to document data provenance into a database. Trio introduces a new query language TriQL [14], an extension of SQL, to deal with uncertainty and lineage information. Trio explicitly documents lineage information about direct ancestors only for each output data tuple. Therefore, users have to facilitate Trio's recursive traversing lineage algorithm to achieve complete provenance of a particular output tuple. However, Trio does not address temporal data rather it focuses on adding accuracy and lineage for conventional, non-temporal data.

Trio

LIVE

To overcome the limitations of Trio project [131] addressing temporal data, Sarma et al. introduced LIVE [109], a complete DBMS, which can store relations with simple versioning capabilities. Versioning in LIVE is realized by adding start and end logical version number to each data tuple. LIVE is an offshoot of Trio which can also manage data with its uncertainty and lineage. LIVE adopts Trio's lineage functionality to propagate annotations containing lineage and uncertainty information of data tuples. Like Trio, LIVE also provides fine-grained (tuple level) data provenance.

Propagation rules

The rest of the work generating provenance in the context of database systems fall in the category of eager approach or annotation-based approach. Buneman et al. have proposed *propagation rules* [21] which are defined for each relational operator to determine how annotations are carried from source to output database. This technique can express location-based dependency between source and output database.

DBNotes

DBNotes [28, 15] is an annotation management system for relational database systems. It also adopts the idea of annotation propagation. It allows users to specify how annotations should propagate from source to output database by facilitating pSQL, an extension of SQL. It provides three different types of annotation propagation scheme that are supported by pSQL. In the *default* scheme, annotations are propagated based on where data is copied from (where-provenance) whereas in *default-all* scheme, annotations are propagated according to where data is copied from in all equivalent queries. DBNotes also provides *custom* scheme based on user specification. DBNotes has the ability to provide a detailed explanation on

provenance of a data tuple by analyzing automatically propagated annotations through SQL queries.

Green et al. have proposed another technique to capture provenance in database systems called *provenance semirings* [61]. This study not only focuses on *why-provenance*, but also identifies the need to understand *how-provenance* which describes how the input data is transformed to produce the output data. *Provenance semirings* facilitates relational algebra calculations to represent provenance information. In this approach, annotations in the form of variables are attached to each data tuple 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. Authors described an application that applied the technique of provenance semirings in the context of collaborative data sharing in [60].

Provenance semirings

In another work, Glavic et al. have proposed the Perm system (Provenance Extension of the Relational Model) which represents provenance as a relation containing both output tuples and contributing input tuples [56]. Perm propagates provenance related annotation along with the actual results by rewriting relational operators within a query. The Perm system focuses on *why-provenance*.

Perm

Another project has been initiated for recording and querying provenance data, called Tupelo2 project[1]. This project is aimed at creating a metadata management system based on semantic web technologies [46]. It stores annotation triples (subject-predicate-object) in several kinds of databases, including relational databases.

Tupelo2

Studies reported in [21, 28, 15, 61, 56] are annotation-based provenance capturing techniques. A recent study conducted by Buneman et al. [23] has shown that annotations may itself be annotated. In this study, authors have described a hierarchical model of annotations where annotations are treated as first class data. A particular query together with the data, describes what is to be treated as data and what as annotation. Propagating annotations through the query are based on lineage and boolean semantics. Authors have validated their idea by annotating datalog programs.

Hierarchical annotations

Several provenance solutions in database systems address *why-provenance* [133, 30, 29, 56, 131, 109]. Recent studies in this area provide extended solutions that can address *where-provenance* [21, 28, 15]. Another study conducted by Green et al. focuses on the need to understand *how-provenance* [61]. Usually, scientists from other domains would like to investigate the

Summary

1 Available at `http://tupeloproject.ncsa.uiuc.edu/`

derivation history of an unexpected result, i.e., why is it in the output, referring back to the *why-provenance*. Therefore, there is a scope of further research in this direction.

Furthermore, existing works supporting provenance in the context of database systems can be also classified based on the provenance capturing approach - *eager* and *lazy* approach. Eager or annotation-based approaches [21, 131, 109, 28, 15, 60, 56, 46] documents provenance information explicitly which incurs a considerable amount of storage overhead for maintaining provenance data. From this discussion, we can sense the need of a framework managing data provenance at reduced storage costs.

2.1.3 Provenance in Stream Data Processing

In general, stream data processing engines handle massive amount of sensor data which are transmitted in form of a data stream - a real-time, continuous, ordered sequence of data products [59]. A data stream should be processed immediately unlike conventional data. Therefore, traditional data processing techniques/queries are not sufficient because of their inability to handle characteristics of streaming data. Instead, a new class of queries, known as continuous queries, are defined to process streaming data. Continuous queries facilitate push-based query semantics, i.e., once new data tuples arrive, queries are executed automatically and the result is provided to users or to the next level of processing.

Dataflow process networks

Lee et al. proposed a computational model, known as *dataflow process networks*, that can model stream based data processing approaches [82]. In dataflow process networks, each process consists of repeated *firings* of a dataflow *actor*. When the actor fires, it maps input tokens into output tokens. Firing consumes input tokens and produces output tokens. A sequence of such firings is a particular type of a Kahn process network [76]. By dividing processes into actor firings, the considerable overhead of context switching incurred in most implementations of Kahn process networks is avoided.

TelegraphCQ

Chandrasekaran et al. adopted the concept of *dataflow process networks* for stream data processing engine [25]. They proposed a continuous query processing engine, called TelegraphCQ, which mainly focuses on shared query evaluation and adaptive query processing. Since streaming data is an infinite sequence of data products, a new mechanism was required to bound this infinite data stream so that continuous queries could be exe-

cuted over a finite set of data products which is a subset of the entire data stream. In [25], authors defined different windowing schemes to bound this infinite data stream. However, provenance tracing has not been supported in TelegraphCQ.

The interest of the research community in stream data processing initiates several academic and commercial research projects which result into a number of stream data processing prototypes. The Stanford InfoLab has undertaken a data stream processing project called STREAM (STanford stREam datA Manager) [8, 7]. STREAM focused on computing approximate results and also tried to understand the memory requirements of posed queries. It proposed *synopsis data structure* which is an approximate data structure rather than an exact representation and was used to optimize storage requirement for stream data. In 2006, this project has officially wound down. It did not provide any support for data provenance. *STREAM*

There were several other projects in this field. The Aurora [2] system manages data streams for monitoring applications. It has a *resample box* which is used to coordinate between different data streams. In Aurora, one tuple must be delayed for the arrival of the second one for processing if there is any *resample box* attached to the system. Aurora minimizes storage need for streaming data tuples by using *load shedding*. This mechanism drops tuples randomly or filter out tuples based on a given condition. The Borealis [3] is an extended version of Aurora and also includes distributed functionality. Since it inherits core stream processing functionality from Aurora, it coordinates various streams and also optimizes the storage requirement. However, Borealis can dynamically revise the query results if the previously generated result is imperfect or partial. Moreover, it optimizes the distributed processing of sensor data in terms of processing speed and memory consumption. Gedik et al. proposed a large-scale, distributed data stream processing middleware, called System S [53]. It supports structured as well as unstructured data stream processing and can be scaled from one to thousands of compute nodes. It introduces a *barrier* operation which is used as a synchronization point. This point consumes multiple data streams and provides output only when a tuple from each of those arrived. Unlike Aurora and Borealis, System S does not optimize storage requirement of data products. These aforesaid prototypes do not possess any support to maintain provenance. *Aurora* *Borealis* *System S*

There are a few existing work addressing data provenance in stream data processing environment. Vijayakumar has recorded provenance of

Change in stream events

data streams by documenting the change in terms of rate and accuracy of the input streams [123]. In this connection, a data model and architecture for capturing and collecting provenance by facilitating timestamps of change events has been proposed in [124, 125]. This model captures provenance by accommodating two types of information: initial information about input streams and a change log. Since the change events have attached timestamps, the model can associate these change events with the set of events in the output stream which have been affected by a particular change. It can save a lot of disk space by storing the information only if a change event occurs during execution. However, it does not provide provenance at a fine level of granularity (tuple level).

Provenance-aware sensor data

Another study conducted by Ledlie et al. have outlined the structure of a provenance-aware storage for sensor data [81]. They have identified two levels of data indexing mechanism in a streaming context: for each tuple or for a set of tuples. This is similar concept like fine- and coarse-grained data provenance, respectively. This work also discussed trade-offs between a centralized data model and a decentralized one to manage provenance information.

Fine-grained provenance in data streams

The work reported in [124, 125, 81] does not provide details about a provenance capturing framework that can maintain provenance at tuple level, i.e., fine-grained data provenance, in a stream data processing environment. Glavic et al. presented several use cases that motivates the necessity of maintaining fine-grained data provenance for streams as well as the study highlighted a few challenges to achieve such a framework [57]. Studies reported in [103, 129, 93, 40, 58, 108] proposed different mechanisms to manage fine-grained data provenance for data streams. Next, we discuss these work in turn.

Tuple-level link

Park et al. proposed an annotation-based approach, called *tuple-level link*, to document provenance of sensor data explicitly [103]. In this system, the transformation of online sensor data is encoded into a URI compatible link, known as a *predecessor link*, which allows users to understand how processed results are derived from original source data, supporting detection and correction of anomalies. A predecessor link can describe the location of source database and table alongside the search used to retrieve data from that table and a timestamp. Moreover, this study also presents a compression technique to reduce the size of links, called *incremental compression*. This work is designed especially to provide provenance of online

sensor data in sensornet systems [106] and therefore, it is difficult to apply this technique for data streams in other domains.

Another study by Wang et al. argued that annotation-based approaches are not suitable for recording provenance in case of data streams because of the high storage overhead [129]. To overcome this limitation, the TVC (Time-Value-Centric) model was introduced to provide a model-based provenance solution supporting stream processing in medical information systems. The approach taken in TVC model focuses on the relationship between tuples of input and output streams of a processing element/activity. This relation can be explained in terms of some primitive invariants: time-invariant, value-invariant and sequence-invariant. *Time* is a primitive that captures dependencies between an output data product and a past time window, which is comprised of input data products, generating that particular output data product. The *value* primitive defines a dependency in terms of predicates over the attributes of the input data products. The *sequence* primitive expresses dependencies in terms of the sequence number of arriving elements, i.e., explaining dependency on a window based on number of tuples. By combining these primitive invariants provenance of any data product in output streams can be explained.

TVC model

Later, Misra et al. have proposed an storage optimization technique [93] over the work described in [129]. Since data products in input and output streams and their intermediate result data has to be stored to achieve fine-grained data provenance, this persistence of high volume streaming data incurs too much storage overhead. Misra et al. have proposed a technique called *Composite Modeling with Intermediate Replay* (CMIR) to reduce the storage requirement. According to this technique, a group of stream processing elements are aggregated into a virtual group which is referred to as a virtual PE (Processing Element). Only input and output streams of the virtual PE are made persistent and thereafter, define dependencies based on TVC model. This technique can reduce storage costs consumed by provenance recording by grouping intermediate processing elements together.

CMIR

Another work reported in [40, 58] also addresses fine-grained data provenance management for data streams. In this study, authors have proposed a provenance-aware data stream management system, called Ariadne. Ariadne has been implemented on the top of Borealis stream processing engine [3]. It is an annotation-based approach that propagates provenance information alongside output tuples through operator instrumentation. Ari-

Ariadne

adne also applies different techniques to optimize storage space required by provenance data.

Stream ancestor functions

Recently, a study conducted by Sansrimahachai have proposed a solution to track fine-grained data provenance for data streams [108]. Author has proposed a provenance model for streams that allows to obtain provenance of individual stream elements/data products. In this study, a provenance query method has been introduced which utilizes *stream ancestor functions* to obtain the provenance of a particular data product. A stream ancestor function designed for a particular activity takes the recorded provenance assertions of input streams as one of its parameter. It may also take the size of the window and processing delay as additional parameters to establish dependency between the output stream element and a set of input stream elements based on the activity. This technique also supports on-demand provenance queries and optimizes the storage overhead for recording provenance assertions.

Summary

One of the major challenges to track fine-grained provenance for streams is the amount of provenance data. Most of the aforesaid existing work try to apply different techniques to optimize the storage requirement of provenance data. However, solutions reported in [103, 129, 58, 108] have to maintain persistent provenance information to some extent. Documenting provenance information explicitly at a lower level of granularity, i.e., in tuple level, often makes provenance information much larger than the size of actual data tuples. Especially in case of operations/activities with sliding windows, a particular data product in input stream elements may contribute to produce several data products in output stream element. The large window size and a significant overlaps between sliding windows can increase the storage cost to maintain fine-grained data provenance substantially. In this case, the storage overhead for provenance collection could be several orders of magnitude higher than the storage required by the collection of both input and output tuples. Therefore, investigating the challenge of managing fine-grained data provenance at reduced storage costs is an worthy research direction in the field of provenance.

Moreover, neither of these solutions [129, 58] consider the erratic nature of stream data processing. Depending on the nature of operations/activity and current workload of the system, it is possible that at each execution of a particular operation, it would take variable amount of time to process input data products, which we refer to as *processing delay*. Furthermore, arrival of data tuples in the input stream element could be also irregu-

lar, i.e., the time between two successive input data tuples arrival, also referred to as *sampling interval*, could vary. The reverse mapping technique using stream ancestor functions presented in [108] explicitly documents the parameters associated with the processing (e.g. processing delay) in provenance assertions. However, it might not be a feasible choice for data streams since documenting this information for every input stream element/data product would incur a significant amount of storage overhead. These aforesaid factors should be considered while designing new methods to achieve fine-grained provenance for data streams.

2.1.4 Provenance in Domain and Application Specific Systems

Provenance-aware systems have been developed in many domains, targeting towards specific applications. There exists a number of work in Geographic Information Systems (GIS) to capture, represent and share provenance data. Furthermore, systems collecting provenance in generic data processing tools have been also proposed. There are a few language specific tools that can capture functional level provenance information. In this section, we discuss these application specific provenance-aware systems.

One of the earliest definitions of provenance was given in the context of geographic information system (GIS). In GIS, data provenance is known as lineage, which explicates the relationship among events and source data in constructing the data product [80]. Provenance is seen as a type of data quality measure in geospatial domains. Yue et al. have proposed an approach to capture provenance data automatically using semantic web technologies [136]. The provenance data has been stored in a RDF triple store and has been queried using SPARQL based on a geospatial data provenance ontology. In another study, a provenance framework has been proposed for geoprocessing workflows [137]. This framework provides provenance at different levels of a given geoscientific model. In [138], authors have reported an approach which enables interoperability for the collected provenance information in a service-oriented GIS architecture.

Geospatial domain

Another study by Frew et al. have proposed a data management infrastructure to track processing of satellite images [43]. In this study, specialized tools are used to collect metadata about the processing steps applied over different data objects. This provenance information is recorded into the database in XML format. Afterward, this technique has been extended by the study reported in [45]. In this study, authors have introduced a new

ES3

approach to capture provenance by monitoring system-level calls from different running processes instead of explicitly specifying provenance about different processing steps. This provenance collection technique is similar to the techniques that capture provenance at operating system level [99]. This technique has been used to build a prototype, called Earth System Science Server (ES3) [44]. Furthermore, in another study, Dozier et al. facilitates ES3 system to build provenance traces of a scientific experiment computing snow-covered areas [38]. In large-scale computational systems like Grid and Web Services, Szomszor et al. have proposed a data model to provide infrastructure level support for a provenance recording capability [118]. They have facilitated relational database systems for creating a provenance repository.

Process mining

In the field of business process intelligence, event logs are used to extract knowledge to do process mining. These log files store detailed derivation history of processes, i.e., provenance of business processes. In this direction, van Dongen et al. proposed a meta model for event logs, represented as an XML format, called MXML [121]. Moreover, the same group of authors also developed the ProM framework [122], which is flexible with respect to the input and output format of log files, and is also open enough to allow for the easy reuse of code during the implementation of new process mining ideas.

Generic data processing

In another study conducted by Groth et al. [66], authors have proposed a technique that can reconstruct provenance of the manipulations done over the data in a provenance-unaware platform like excel sheet or a programming tool like R. This approach used a library of basic transformations to infer and reconstruct provenance for a particular value. Since it requires predefined possible transformations to reconstruct the data provenance, this approach is not easy to apply in other platforms. In [113], authors proposed a variant of R interpreter, CXXR, which can maintain and represent collected provenance information. Miles et al. [90] have proposed a provenance collection mechanism by modifying the source code of a program automatically. It provides fine-grained data provenance after executing the script. This mechanism has been demonstrated over Java programs. In studies reported in [61, 23], authors have demonstrated their proposed provenance collection mechanisms in context of Datalog programs.

PriMe

There exist a few research on collecting provenance from specific programming languages/tools which are typically do not provide any built-in support for provenance collection and query, i.e., *provenance-unaware plat-*

form. In this connection, Miles et al. have proposed a software engineering methodology, known as Provenance Incorporating Methodology (PrIMe) [92], that can adapt applications to make them provenance-aware by exposing application information documented through a series of steps and by modifying the application design.

Software engineering

Provenance can be defined at different levels of granularity. As an example, provenance documented at workflow level only explicates the relationship among different operations/activities. Similar work has also been done in software engineering domain by facilitating static program analysis. A program dependence graph (PDG) makes both the data and the control dependences explicit for each statement in a program [41]. A PDG can be used to describe an overview of a program. A system dependence graph (SDG) extends the definition of a program dependence graph and it is capable of providing data and control dependences for multi-procedure programs [67]. One of the conceptual difference between workflow provenance and SDG/PDG is that workflow provenance explains the relationships among activities in terms of data dependences only whereas a PDG/SDG may have control dependences among activities.

Language specific tools

Similar types of software engineering tools have been developed for general-purpose programming languages like C, Python etc. Frama C[2] is a code analyzer developed for C programming language. It also supports the concept of program slicing [130]. Angelino et al. have proposed StarFlow [6] which can build a provenance trace at functional level for a Python program. However, this tool can not explicate data dependencies within a function. There exists a tool called Sumatra[3], developed for Python. It is a tool for automated tracking of scientific experiments to achieve reproducible results. There exist some other tools for analyzing Python programs[4,5,6]. These tools can show call graphs based on a given Python program, i.e., dependency among different modules used in the script. However, neither of these tools can provide complete data-driven relationships among different operations in a given Python program.

There are several existing work that extract provenance information from a scientific model, developed in a *provenance-unaware* platform such as

2 Available at `http://frama-c.com/`
3 Available at `https://pypi.python.org/pypi/Sumatra`
4 Available at `http://furius.ca/snakefood/`
5 Available at `http://pycallgraph.slowchop.com/`
6 Available at `http://www.tarind.com/depgraph.html`

generic data processing tools, general-purpose programming languages etc. Studies reported in [90, 66, 113] capture provenance information from a program developed in a provenance-unaware platform. These methods do not employ static program analysis techniques, used in software engineering and hence, can capture provenance only at the execution time. There exist several other tools and packages, developed for Python, which can produce dependency graphs showing functional/modular level dependencies. It could be worth investigating to extract data dependencies between different operations/activities without executing a particular program.

Summary

2.2 PROVENANCE REPRESENTATION AND SHARING

Based on the discussion in the previous sections, we have seen that the provenance-aware systems have been addressed by different domains. As a consequence, the representation of provenance might vary from one system to another. To ensure seamless exchange of provenance information, standardization of provenance representation and sharing methods is required. In this section, we discuss a few provenance representation methods facilitated by different applications and then describe the state-of-the-art in connection with provenance representation and sharing.

Provenance has been represented in different ways. As an example, in geographic information systems (GIS), there is a standard known as ISO 19115:2003[7] which defines the schema describing metadata, for geographic information and services. This metadata schema can be translated into Geography Markup Language (GML)[8], which is the XML grammar defined by the Open Geospatial Consortium (OGC) to express geographical features. GML serves as a modeling language for geographic systems as well as it provides an open interchange format for geographic transactions on the Internet. In another study, Yue et al. have proposed to facilitate XML encoding to represent geospatial provenance data for better interoperability [138].

GML

Since provenance representation changes from one system to another, it makes the exchange of provenance information between systems extremely difficult. The Open Provenance Model (OPM) [96, 98] is designed for supporting interoperability between provenance systems. OPM allows

Open Provenance Model

7 Available at `http://www.iso.org/iso/catalogue_detail.htm?csnumber=26020`
8 Available at `http://www.opengeospatial.org/standards/gml`

provenance information to be exchanged between systems, by means of a compatibility layer based on a shared provenance model proposed in [97]. The model defines a causality graph that consists of *artifacts*, *processes*, *agents* and the causal relationships among these nodes. *Artifacts* denote data products whereas *processes* represent a series of operations/activities performed on artifacts and their execution produces new artifacts (data products). *Agents* point to users/scientists who control the execution of processes. To represent causal relationships between these nodes, OPM introduces five primitive types of edges. An edge represents a causal dependency, between its source, denoting the effect, and its destination, denoting the cause. OPM has been designed to represent any kind of provenance, even if it has not been produced by computer systems.

W3C specifications

The Open Provenance Model (OPM) is further extended and realized by a set of W3C provenance (PROV) specifications[9], defining various aspects of PROV. PROV-DM is the conceptual data model that forms a basis for the PROV family of specifications [95]. PROV-DM defines the core concepts of PROV which are categorized as *types* and *relations*. There are three PROV-DM types: *entities*, *activities* and *agents*. These PROV-DM types are analogous to the nodes defined in OPM [98]. An *entity* is a physical, digital, conceptual, or other kind of thing, either real or imaginary, with some fixed aspects. *Entities* in PROV-DM are similar to *artifacts* in OPM. An *activity* is something that occurs over a period of time and acts upon or with entities, i.e., activities in PROV-DM are similar to *processes* in OPM. An *agent* is something that bears some form of responsibility for an activity taking place, for the existence of an entity, or for another agent's activity. Therefore, *agents* defined in PROV-DM is a super set of *agents* defined in OPM. Furthermore, as core concepts of PROV, there are seven PROV-DM relations which represent casual dependencies between PROV-DM types. PROV-DM is a generic data model for provenance that allows domain and application specific representations of provenance to be translated into such a data model and then enabling interchange between systems. Therefore, PROV-DM is domain and application agnostic. Heterogeneous systems can export their native provenance into such a core data model. PROV includes other W3C specifications for different purposes. PROV-O allows mapping of the PROV-DM to RDF (Resource Description Framework) whereas PROV-CONSTRAINTS defines a set of constraints

PROV-DM

9 Available at `http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/`

applying to PROV-DM. PROV-N defines notations of types and relations in PROV-DM, represented in a human understandable form.

In this thesis, we mainly focus on the collection of provenance information based on a scientific model. Therefore, we do not address this particular dimension of research in the field of provenance in this thesis. Nevertheless, we expect that the provenance information provided by the envisioned framework could be exported to these standardized representation techniques.

2.3 PROVENANCE APPLICATIONS

A study conducted by Simmhan et al. [114] narrated different applications of provenance information. Provenance can be used to have reproducible results. It becomes also useful for validating scientific models. Furthermore, provenance information can be facilitated to trace where data come from and for other auditing purposes [91]. To debug the outcomes of a scientific model and the model itself, provenance information could also play a role. In this section, we primarily focus on the usage of provenance information for debugging purposes.

Debugging in software engineering

In the software engineering domain, there are two common methods for debugging: log-based debugging and breakpoint-based debugging. Log-based debugging inserts logging statements within the source code to produce an ad-hoc trace during program execution. Breakpoint-based debugging consists in running the program under a dedicated debugger which allows the programmer to pause the execution at determined points, inspect memory contents, and then continue execution step-by-step. Most of the current IDEs (Interactive Development Environment) have a breakpoint-based debugging feature. Both log-based and breakpoint-based debugging require manual analysis of massive execution traces which is difficult and not scalable. To overcome this drawback, Pothier et al. [105] have proposed a scalable omniscient debugger, called Trace-Oriented Debugger for Java (TOD), that makes it possible to navigate backwards in time within a program execution trace, drastically improving the task of debugging complex applications. in TOD, event traces (provenance) are explicitly stored in a distributed database and therefore, it is very storage-expensive. A recent study [134] on debugging aspect-oriented programs has argued that some bugs in this programming paradigm could be difficult to detect,

since aspect-oriented source code elements and their locations could be transformed or even lost after compilation. In this study, authors have proposed a debugger for aspect-oriented languages by facilitating an intermediate representation of the particular program to debug which preserves all source-level abstractions so that the programmer could inspect the execution of all program elements in case of any bug reported.

PANDA

Very recently, provenance information has been also used for debugging [74]. In this study, Ikeda et al. have demonstrated Panda (Provenance and Data), a system that supports debugging of data-oriented workflows and drill-down to source data for a given output data. Panda documents provenance information explicitly and it is not known that whether Panda can handle data streams or not.

In this thesis, we emphasizes the application of provenance as a debugging tool. Provenance defined at workflow level explains data dependencies between activities which can be facilitated to detect any error in the scientific model. Further, fine-grained data provenance could help scientists to trace an output data product back to its source values if the output seems to have an abnormal value. In this way, provenance could be used as a potential tool for debugging at different levels.

2.4 RELATION TO RESEARCH QUESTIONS

Several provenance-aware systems and models have been described in previous sections. Based on this discussion, we point out a few key observations in the context of the research questions (see Section 1.4) which are in the center of the investigation in this thesis. These observations and their consequence on our work are discussed below.

This thesis mainly focuses on three research questions. The first research question, *RQ 1*, raises the concern on capturing workflow provenance of a scientific model automatically.

> *RQ 1: How to capture automatically the workflow provenance of a scientific model developed in a provenance-unaware platform at reduced cost in terms of time and training?*

Existing research and systems collecting provenance in the context of a scientific data processing model are discussed in Section 2.1. From this discussion, we have two major observations.

> Observation 1.1: Each scientific workflow engine supporting provenance collection has its own set of constructs to design the workflow of a scientific model. A few of them are also domain-specific [102, 68].

Scientific workflow engines collect provenance at different levels of granularity - both workflow and fine-grained provenance [84, 115, 11, 24, 77, 63]. Scientists who would like to facilitate one of these systems, have to learn the design constructs of that particular system which is a time consuming task. Furthermore, these systems cannot extract workflow provenance from a typical Java or Python program carrying out scientific experiments. Therefore, a solution that can extract data-driven relationship between variables and operations within a program, i.e., workflow provenance, would help the scientific community to manage provenance at reduced effort in terms of time and training.

The second observation in connection with the *RQ 1* is given below:

> Observation 1.2: There exist a few systems that can extract workflow provenance, i.e., data dependencies between activities, in a typical provenance-unaware platform such as general-purpose programming languages, generic data processing tools etc. Some of these techniques can collect provenance during execution time only. The others can provide functional-level dependencies instead of data dependencies.

Studies reported in [90, 113, 66] can extract provenance information from programs/scripts developed in different provenance-unaware environment such as Java, R, Microsoft Excel, respectively. These systems can only extract provenance during the execution of the program. There are a few other tools and packages developed for the Python programming language which can extract functional-level dependencies by facilitating static program analysis, discussed in Section 2.1.4. Since workflow provenance explicates the data-driven dependencies between activities/operations in a scientific model or in a program, neither of these solutions can extract workflow provenance. Therefore, we would like to develop a method that can extract workflow provenance from a given scientific model, consisting of programs/scripts, automatically. In this thesis, we present workflow provenance inference method applicable for Python.

The *RQ 2* highlights the practical challenge of collecting fine-grained data provenance at reduced storage cost under different execution environments.

> *RQ 2: How to infer fine-grained data provenance under different system dynamics at reduced cost in terms of storage consumption?*

Existing work collecting fine-grained data provenance are discussed in previous sections. From this discussion, we point out the following observations.

> Observation 2.1: Most of the existing solutions for tracing fine-grained data provenance explicitly annotate provenance information. Some of them also propagate provenance from one activity to the next one [21, 131, 60, 56, 58].

Studies reported in [21, 131, 109, 28, 15, 60, 56, 46] are annotation-based approaches to collect provenance in the light of database systems. Moreover, there exist some other work in the context of data streams which also document provenance assertions/information explicitly [103, 129, 58, 108]. As we have already discussed in Section 2.1.2 and 2.1.3, provenance information defined at tuple level, i.e., fine-grained data provenance, consume a lot of storage space. Especially in cases of collecting provenance for data streams, the storage overhead due to the explicit provenance collection could be several magnitudes higher than the storage required by actual data tuples. We reported this storage problem of maintaining fine-grained data provenance in one of our earlier [69]. Therefore, in this thesis, we would like to develop techniques that can infer fine-grained data provenance to minimize storage cost.

The second observation pertaining to the *RQ 2* is the following.

> Observation 2.2: In stream data processing, several techniques facilitate timestamps attached to data tuples as the key element to derive dependencies between output and input data products.

As discussed in [8, 59], each data product in a stream is associated with a timestamp. Several existing research facilitate this timestamp to express data dependencies between input and output data products in streams

[129, 103, 109, 58, 108]. In one of our initial ideas [69], we have also reported the use of timestamps to establish dependencies between input and output data products, i.e., fine-grained data provenance. We extend this idea further and in this thesis, we present inference-based methods to extract fine-grained data provenance by facilitating timestamps of available input and output data products.

> Observation 2.3: A few systems address underlying system dynamics (e.g. processing delay) that refers to the erratic nature of stream data processing. In these systems, the related parameter such as processing delay is explicitly documented to achieve accurate fine-grained data provenance.

System dynamics is defined as a set of parameters that could affect the underlying data processing techniques. One of the parameters is *processing delay*. In stream data processing, operations could have variable processing delays that affects the generation of output data products. The other parameter defined as a part of system dynamics is *sampling interval*, which refers to the amount of time between arrivals of two successive data products in input streams. Variable *sampling interval* could affect the formation of windows, consisting of input data products to be processed.

In the study reported in [108], the author proposed to document processing delay explicitly as an additional attribute in output data products. In this case, given an output stream element, *stream ancestor functions* can map contributing input stream elements to that particular output stream element by using timestamps of data products and annotated processing delay. Documenting processing delay for each output data products incurs extra storage and processing overhead. To reduce the storage costs of managing fine-grained data provenance trace, we would like to infer fine-grained data provenance without explicit documentation of parameters (e.g. processing delay) influencing system dynamics per output data product. Furthermore, the envisioned inference-based methods do not store any provenance record (provenance assertion) explicitly, which ensures minimal storage overhead. It could be possible that in a few cases like extremely delayed processing or highly irregular sampling interval, the envisioned inference-based methods cannot infer accurate provenance information. In this thesis, we will elaborate this trade-off between storage overhead and accuracy of provenance information.

The last research question in this thesis, *RQ 3*, emphasizes the need of a self-adaptable system that can choose the appropriate inference-based method based on the given scientific model and current system dynamics to infer provenance information.

> *RQ 3: How to incorporate the self-adaptability into the framework managing data provenance at reduced cost?*

In the context of this research question, we have the following observation.

> Observation 3.1: A self-adaptable provenance management framework has yet to be addressed by the research community.

Existing provenance-aware systems are developed to address a particular application or a particular settings. One of the goals in this thesis is to achieve a *generic* framework to manage data provenance. To accomplish this, we would like to introduce a suite of inference-based methods to infer fine-grained data provenance. Each inference-based method should be suitable for a particular environment. To ensure optimal accuracy of inferred provenance information, it is required to dynamically select the best-suited inference-based method depending on the current system dynamics and the characteristics of the scientific model. Therefore, a self-adaptability feature would nicely fit into the envisioned framework, selecting an optimal solution.

In sum, we propose a framework inferring provenance information at different levels of granularity in this thesis. We present a novel workflow provenance inference method that can extract workflow provenance based on a given Python program, realizing a scientific computation. Then, we propose a suite of inference-based methods to infer fine-grained data provenance at different system settings. Our inference-based methods are cost-efficient in terms of storage, time and required training. Finally, we incorporate a self-adaptability feature into our framework so that the framework can select the most appropriate method autonomously based on current system dynamics.

2.5 SUMMARY

The ultimate goal of this thesis is to develop a self-adaptable, inference-based framework managing data provenance in a cost-efficient way. Since

data provenance is at center of the investigation in this thesis, we reviewed existing provenance literature.

Provenance is a widely studied topic. Provenance-aware systems and prototypes are available in the context of scientific workflow, database, stream data processing, e-Science applications etc. At the beginning of this chapter, we discussed these existing systems and research in brief. During the discussion, we also pointed out the limitations of existing approaches.

Representation and sharing of provenance is another important dimension of provenance research. We described state-of-art techniques in provenance representation and sharing. Since the major investigation area in the context of this thesis is the efficient extraction of provenance information, we will not discuss much about adoption of these standards in the rest of the thesis. We consider it as a potential future work. Moreover, we emphasized the use of provenance as a debugging tool. We cited examples from software engineering domain where debugging could be very expensive in terms of storage. Therefore, provenance information could be facilitated while debugging scientific models, if it can be achieved at reduced storage costs.

Next, we analyzed the existing work in the context of the research questions mentioned in this thesis. During the analysis, we made a few observations about the existing work. Based on these observations, we pointed out the key characteristics of the envisioned provenance management framework. We found that the automatic extraction of workflow provenance based on a scientific model could be worth investigating. Furthermore, most of the existing systems store explicit provenance records that incur a considerable amount of storage overhead. Therefore, the proposed inference-based methods, building fine-grained provenance traces without explicitly documenting provenance records, could minimize this storage overhead incurred by provenance data. We would like to also investigate the effects of employing these inference-based methods over the accuracy of inferred provenance information. Finally, we proposed to incorporate a self-adaptable mechanism into the envisioned framework to have an optimal result in terms of storage consumption and accuracy based on the given scientific model and system dynamics. The framework proposed in this thesis, can provide a solid platform to scientists who want to manage data provenance for their data intensive scientific computations.

3

WORKFLOW PROVENANCE INFERENCE

PROVENANCE information collected at different levels of granularity helps scientists to validate a scientific model as well as to understand the origin of a data product with unexpected value. Workflow provenance explicates the data-driven relationships between the activities within a scientific model. One of the emerging applications of workflow provenance information is that it can be used to debug a scientific model. Furthermore, workflow provenance can visualize the overall structure of a scientific model.

As discussed in Chapter 1, there exist several provenance-aware tools and platforms that can collect workflow provenance automatically while designing a scientific model. Scientists can facilitate these provenance-aware platforms to capture workflow provenance of their scientific models. Each of these tools has their own set of programming constructs and operators to define activities of a scientific model. Therefore, using a particular provenance-aware platform requires extensive training effort for scientists, which might also take a considerable amount of time.

Challenges

Moreover, these scientists are an important but limited user group. Angelino et al. [6] reported that these scientists used to write programs/scripts in a general purpose programming/scripting language. Later, they execute these programs to process the collected data and to generate output data products. These programming languages have no built-in support to collect provenance information and we refer to these model develop-

Part of this chapter is based on the following work: An Inference-based Framework to Manage Data Provenance in Geoscience Applications. Accepted in *IEEE Transactions on Geoscience and Remote Sensing*, IEEE Geoscience and Remote Sensing Society, 2013. (Impact Factor: 2.895) & From scripts towards provenance inference. In *Proceedings of the IEEE International Conference on E-Science (e-Science'12)*, pages 118–127, IEEE Computer Society, 2012.

ing platforms as provenance-unaware platforms. In a provenance-unaware platform, scientists have to manually annotate provenance information of their workflow which could become a tedious job if the scientific model is large and complex and could take a lot of time.

Solution

Challenges of extracting workflow provenance in a provenance-unaware platform are posed in the first research question (*RQ 1*) of this thesis, discussed in Section 1.4. To satisfy this research question, we propose to capture workflow provenance automatically based on a given program which is used for the actual processing. The proposed *workflow provenance inference* method infers the data dependencies between activities by interpreting the source code of the given program. To accomplish that, the method has to transform all control-flow based coordination between activities into data-flow based coordination. Later, this workflow provenance can be facilitated to infer more detailed fine-grained provenance information. The *workflow provenance inference* method puts minimal burden to the scientists to capture workflow provenance of their scientific models developed in a provenance-unaware platform automatically and therefore, this method can save a significant amount of time. Furthermore, scientists need less training effort to facilitate the proposed workflow provenance inference method compared to other provenance-aware platforms.

In this thesis, we consider Python[1] programs to showcase the proposed method. However, one can easily adapt the general principles presented in this chapter to develop such a tool for other provenance-unaware platforms. In Chapter 9, we describe an automatic workflow provenance capturing technique in the context of Answer Set Programming (ASP) based on the general principles discussed in this chapter.

Chapter structure

In this chapter, first, we describe a few core concepts of the workflow provenance inference method including workflow provenance model and its semantic alongside provenance representation scheme. Afterward, we present the overview of the workflow provenance inference method followed by the example of building an initial workflow provenance graph. Then, we introduce a set of re-write rules which are applied over the initial workflow provenance graph to transform its control dependencies into data dependencies, thus achieving the workflow provenance graph. We also evaluate the workflow provenance inference method on a collection of the Python programs used by the scientists in several real-life projects. Moreover, we briefly discuss the limitations of the method at its current

1 Available at `http://www.python.org/`

stage followed by the summary of the workflow provenance inference method.

3.1 WORKFLOW PROVENANCE MODEL

The core concept of the workflow provenance inference method is the workflow provenance model. The workflow provenance model defines different types of entities and their relationships with each other in a scientific model. It represents the captured workflow provenance information as a bipartite graph, which we refer to as a *workflow provenance graph*. A workflow provenance graph, G_w, is a set of (U, V, E) where U denotes the set of vertices/nodes representing data products participated in a scientific model, V denotes the set of nodes representing activities/processing elements in a scientific model and E denotes the set of directed edges between an element in U and an element in V (or vice versa), representing a data-driven relationship between these two nodes.

Bipartite graph

The set U is comprised of nodes, representing data products used in a scientific model. The nodes in U can be classified into two categories.

- *View*: represents either any variable defined in the scientific model or a result set/data products produced by a processing element.
- *Constant*: represents any constant value taking part in an activity/processing element.

The set V consists of nodes, representing activities or processing elements within a scientific model. These nodes can be classified into two types.

- *Source Processing Element*: represents an operation that either assigns a constant value into a variable or an operation that reads/acquires data from the disk or any other source.
- *Computing Processing Element*: represents an operation that either computes a value/data product based on its parameters or writes data products into a file, database etc.

The set E consists of directed edges connecting two nodes (one from U and the other from V, or vice versa), representing data dependency between these two nodes. When there is a directed edge from a vertex $u_i \epsilon U$

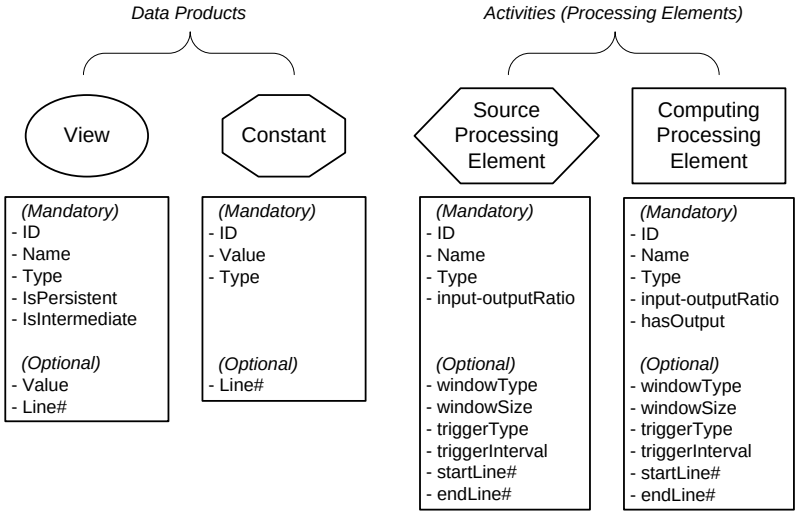


Figure 3.1: Properties of different types of nodes in a workflow provenance graph

to a vertex $v_i \epsilon V$, it means that the view/constant u_i contributes to the processing of the source/computing processing element v_i. On the other hand, if there is a directed edge from a vertex $v_j \epsilon V$ to a vertex $u_j \epsilon U$, it indicates that the processing element (either source or computing) v_j produces the view (data products) u_j.

Each type of aforesaid nodes has different properties. Figure 3.1 shows the graphical representation of these nodes and also lists their properties. In each node, there are several properties which are *mandatory* and the others are *optional*. The set of mandatory properties is used to uniquely identify a node as well as it describes key characteristics of a node based on the discussion in Section 1.3. The set of optional properties can be used to identify the exact data dependencies between these nodes, i.e., fine-grained data provenance. Next, we discuss these properties of each type of nodes.

View

There are two types of nodes, representing data products (view/constant), in the proposed workflow provenance model. A node representing a *view* has a node identifier (ID) prefixed with 'V' based on current implementation, name and type. The type of a view node could be relational. Figure 3.1 also shows two important boolean properties of a view node. These are: i) *IsPersistent* and ii) *IsIntermediate*. The *IsPersistent* property describes whether the data products hold by a view is stored persistently or not. When *IsPersistent=true*, it means that data products hold by the view is read from the disk or is written into the disk depending on the preced-

ing processing element (either source or computing) and hence, is persistent. Otherwise, the view is not persistent and thus *IsPersistent* becomes *false*. The property *IsIntermediate* is *true* when the view is produced by a computing processing element and contains an intermediate result. Otherwise, *IsIntermediate* becomes *false* and it indicates that the view is created to hold the value of a corresponding variable, defined in the program. Furthermore, a view node has two optional properties: i) *Value* and ii) *Line#*. The property *Value* specifies the actual value of data products hold by the corresponding view. *Line#* refers to the line number in source code of the program where the corresponding variable referred by the view (if any), is defined.

Figure 3.1 also lists both mandatory and optional properties of a node, representing a *constant*. A constant node has a node identifier (ID), a value, the type of the value (e.g. integer, string etc.). Based on the current implementation, the *id* of a constant node always starts with 'C'. A constant node has optional *Line#* property which refers to the line number in source code of the program where that particular constant is assigned.

Constant

As already mentioned, the category defining activities/processing elements has two types of nodes: i) source processing element and ii) computing processing element. A node, representing a *source processing element*, has a node identifier (ID), name, type and input-output ratio as mandatory properties. In the current implementation, the *id* of a source processing element always starts with 'SP'. The type of a source processing element node could be assignment, file name, database location etc. Furthermore, a source processing element node has *input-output ratio* as another mandatory property which will be defined later.

Source processing element

Like a source processing element node, a node representing a *computing processing element* has a node identifier (ID), name, type and input-output ratio. The node *id* of a computing processing element is always prefixed with 'P' based on the current implementation. The type of a computing processing element node could be binary operation, function call, conditional branching etc. A computing processing element node has *input-output ratio* as another mandatory property. A source processing element node also has this property. As already defined in Section 1.3.1, *input-output ratio* refers to the ratio of the number of data products which contribute to produce output data products over the number of data products produced by executing a particular processing element. There are many operations where this ratio remains *constant* during execution of the corresponding process-

Computing processing element

ing elements such as projection, aggregate functions in a database etc. We refer to these activities as *constant ratio* activities/processing elements. As an example, an aggregate function (e.g. average, sum etc.) always considers all input data products to produce an output data product. Therefore, the *input-output ratio* for the processing element representing aggregate function is $n : 1$ ('many to one') where n is the number of input data products processed during execution. On the other hand, there are a few operations which do not keep the *input-output ratio* constant during execution of the corresponding processing element such as a typical selection operation in a database. These are referred to as *variable ratio* activities/processing elements.

There is one property, *hasOutput*, which is used to differentiate between a source and computing processing element. It is only applicable for a computing processing element. The property *hasOutput* indicates whether result/data products produced by a computing processing element is stored persistently, i.e., written into the disk, or not. If data products are stored persistently, the value of *hasOutput* is set to *true* for the computing processing element which produces that persistent data products. Otherwise, the value of *hasOutput* of that corresponding computing processing element is set to *false*.

Optional properties of Processing elements

Figure 3.1 also lists several optional properties of a processing element node (both source and computing). Please note that, the list of optional properties shown in Figure 3.1 is not a complete one. The listed properties are the most important ones for a processing element to describe the nature of the processing in terms of data dependencies. These properties are discussed in the following.

- *Window*: A window specifies a subset of data products processed by a processing element to produce an output data product. Therefore, a window with a predefined size is applied over the input data products, i.e., views, to limit the number of data products to be considered by the processing element. Formulating a window requires to define two properties: i) *windowType* and ii) *windowSize*. A window could be defined based on the number of tuples, i.e., tuple-based window (*windowType=Tuple*), or time units, i.e., time-based window (*windowType=Time*). A window is defined per input view and the default value is 1 tuple.

- *Trigger*: A trigger specifies when and how often a processing element will be executed. Two properties are required to define a trigger for a processing element. These are: i) *triggerType* and ii) *triggerInterval*. A *triggerType* specifies how a processing element will be triggered for execution. The value can be either *tuple* or *time*. A *triggerInterval* refers to the interval (number of tuples of time units) between successive executions of the same processing element. A trigger is defined per processing element and the default value is 1 tuple.

There exist two other optional properties of a processing element node. These are: *startLine#* and *endLine#*. Since the definition of a processing element could take multiple lines, *startLine#* and *endLine#* property refer to the start and end line number of the definition of a particular processing element in the source code of a given program, respectively.

3.2 WORKFLOW PROVENANCE MODEL SEMANTICS

In the previous section (see Section 3.1), we have described different types of nodes in a workflow provenance graph. A workflow provenance graph explicates data dependent relationships between processing elements/activities. Therefore, the coordination between different processing elements is data-flow based [111] specifying that once a processing element is triggered, the processing element processes data products hold by single/-multiple input views which are the output of the preceding processing elements. The size of input dataset per view is determined based on the corresponding window defined over an input view. The trigger of different processing elements are done in parallel, where each processing element is performed based on a time or tuple-based trigger. For a time-based trigger, at every point in time where the trigger predicate, consisting of trigger type and trigger interval, is interpreted as valid, the processing element is fired and output data products are produced. For a tuple-based trigger, it is more difficult to predict when the next trigger will be enabled. Therefore, the particular processing element continuously checks whether the set of new data products observed at a specific time is sufficient to validate the trigger predicate as true. In this case, the processing element is executed and output data products are generated. Execution of processing elements never stops as long as new data products arrive and the arrival of new data triggers a processing element which is receiving the data.

The aforesaid semantics of the proposed workflow provenance model is quite similar to the computational model, known as *dataflow process networks* [82]. In dataflow process networks, each process (processing element) consists of repeated firings (triggers) of a dataflow actor. When the actor fires, it maps input tokens (input data products) into output tokens (output data products). A sequence of such firings is a particular type of Kahn process network [76]. By dividing processes into actor firings, dataflow process networks overcomes the limitation of Kahn process network in which processes (processing elements) are not allowed to test an input channel (view) without consuming tokens (data products) resulting into a block over other processes (processing elements).

However, there exist a few differences between a dataflow process network and the semantics we use to define and execute a workflow. Based on the aforesaid semantics of the proposed workflow provenance model, a processing element can be executed based on the number of data products/tuples (tuple-based trigger) or a pre-defined time interval (time-based trigger). In a dataflow process network, a dataflow actor can be fired based on the number of input tokens only, i.e., no firing mechanism defined based on time domain.

3.3 WORKFLOW PROVENANCE REPRESENTATION

In the proposed workflow provenance model, discussed in Section 3.1, workflow provenance information is represented as a graph. Therefore, we facilitate GraphML[2], a XML-based file format for exchanging graph structured data, to represent provenance information. GraphML is supported by most of the graph editing tools such as yEd[3], Gephi[4] etc. Therefore, provenance graphs can be shared easily through out the scientific community. Currently, we do not facilitate any standardized methods, discussed in Section 2.2, to represent and share the workflow provenance. Since our main purpose is to visualize the workflow provenance graph, we stick to GraphML. In the future, we will extend the workflow provenance model accordingly to adopt the existing World Wide Web Consortium (W3C) provenance specifications[5].

2 Available at `http://graphml.graphdrawing.org/`
3 Available at `http://www.yworks.com/en/products_yed_about.html`
4 Available at `https://gephi.org/`
5 Available at `http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/`

3.4 OVERVIEW OF WORKFLOW PROVENANCE INFERENCE

The workflow provenance inference mechanism can infer workflow provenance by analyzing a Python program. A typical Python program is comprised of assignments, arithmetic operations, user-defined function calls, conditional branching, iterative operations etc. Some of these operations/activities (e.g. assignment, arithmetic operations) exhibit data-flow based coordination whereas the others exhibit control-flow based coordination [111]. As an example, an assignment operation assigns a value (represented by either a view or a constant node) into a variable (represented by a view). Therefore, the assignment activity shows data-flow based coordination where availability of data products trigger the activity and the workflow provenance graph can easily explicate the data dependency of such type of activities.

On the other hand, activities such as user-defined function calls, conditional branching, iterative operations etc. are implemented by using control-flow based coordination and result into control dependencies between activities. A control-flow based coordination maintains the dependence in such a way that an activity is started only after the preceding activity has been completed. As a consequence, variables defined or updated in an activity are accessible after the activity has been completed. Therefore, control dependencies can be represented as data dependencies by creating different versions of the same variables involved in the activity. In particular, a new version of a variable is created after its modification by an activity. The control-flow determines which version of a variable will be used by a read operation of an activity on that variable. Since workflow provenance describes the data dependencies between activities, control dependencies must be transformed into data dependencies to infer the workflow provenance.

Challenge

We start the inference mechanism by parsing a given Python program based on a combined grammar, containing parser and lexer rules. After parsing the source code of the program, it returns an abstract syntax tree (AST) of the given Python program. Then, we traverse through this AST based on a tree grammar and for each node in the AST, an object of the appropriate class based on the object model of the Python is created. Having all the objects, we build the *initial workflow provenance graph* maintaining the syntactic relationship between these objects including control-flow based coordination.

Solution

Transformation function

Since the initial workflow provenance graph preserves the control-flow based coordination and contains some extra nodes due to the syntactic sugar of the Python, it needs to be transformed in a form where the graph exhibits data dependencies only and becomes more compact. To accomplish that, we apply a transformation function, consisting of a set of graph re-write rules, called *flow transformation re-write rules*. Each re-write rule in the set handles a particular operation/activity and transforms control dependencies, found in the initial workflow provenance graph, into data dependencies. The function contains a set of re-write rules capable of handling conditional branching, iterative operations (looping), user-defined function call as well as some Python oriented control structures.The transformation function continues to apply re-write rules till no control dependency can be found in the initial workflow provenance graph.

Maintenance function

Later, we apply another function, called maintenance function, over the resulting graph after applying the transformation function. The maintenance function contains a set of re-write rules, called *graph maintenance re-write rules*. These re-write rules are not used to transform control dependencies into data dependencies rather they are used to propagate the persistent property of a view to the next view where applicable as well as to identify the computing processing element producing persistent data products.

Compression function

Finally, we apply a compression function which consists of a set of *graph compression re-write rules*, to reduce the number of nodes by combining nodes, representing *views* and *constants* with appropriate processing element (both source and computing) nodes. All re-write rules in aforesaid functions are executed one after another. After applying all these functions, we get the *workflow provenance graph*.

We have used an off-the-shelf grammar[6] as a starting point and extend it according to our requirements to obtain the abstract syntax tree. We also facilitate attributed graph grammar (AGG)[7] which is a graph writing engine to define re-write rules and transform the graph accordingly. We also encode these graph re-write rules in a rule notation scheme, RuleML[8], to make it easier to implement these re-write rules in a different settings. RuleML provides XML encoding of these re-write rules. An example is

6 Available at `http://www.antlr.org/grammar/1200715779785/python.g`

7 Available at `http://user.cs.tu-berlin.de/~gragra/agg/`

8 Available at `http://ruleml.org/`

given in Appendix A.1. In the next section, we discuss the mechanism of creating an initial workflow provenance graph with the aid of an example.

3.5 INITIAL WORKFLOW PROVENANCE GRAPH

Building an initial provenance graph depends on the created objects based on the object model of the Python and their syntactic relationship to each other. As already mentioned, we build the initial workflow provenance graph by facilitating AGG graph writing engine.

Program description

Figure 3.2 shows a sample program and the initial workflow provenance graph of the given program. In this program, the first two lines read two files, `'input1.txt'` and `'input2.txt'`, each containing a value and these values are transformed into `int` type which are then assigned into the variables `a` and `b` respectively. These variables are used in an addition operation and the resultant value is then assigned into variable `c` in line 3. In the next line, `a` is reassigned with the value 100. In line 5, the value hold by `c` is written into a file named as `'output.txt'`, stored persistently.

Example

In Figure 3.2, the source processing element nodes SP_1 and SP_2 represent the `read` method which reads the input files containing a value. For each method which is used for the first time, the user has to provide a few information beforehand such as: whether a method reads data from disk (true/false) and whether a method writes data into disk (true/false) to make a distinction between source and computing processing elements. For a `read` method, while the value of the former is *true*, the value of the later is *false*. SP_1 and SP_2 produce two intermediate views, V_1 and V_4, respectively. These views are used by computing processing elements P_1 and P_3 which transform the value hold by these views into *integer* type and then the values are assigned into variables `a` and `b` represented by the view nodes V_3 and V_6, respectively. These views participate in an addition operation denoted by the node P_5 which produces the result hold by an intermediate view V_7. The computing processing element P_6 assigns the view V_7 into the variable `c`, represented by the view node V_8. Afterward, the processing element node P_7 reassigns the variable `a`, represented by the node V_9 with the value 100, denoted by the constant node C_3. Eventually, the processing element node P_8 is created to represent the `write` method which writes the value hold by the view node V_8 into the output file, stored persistently into the disk. The nodes in Figure 3.2 also mention the value

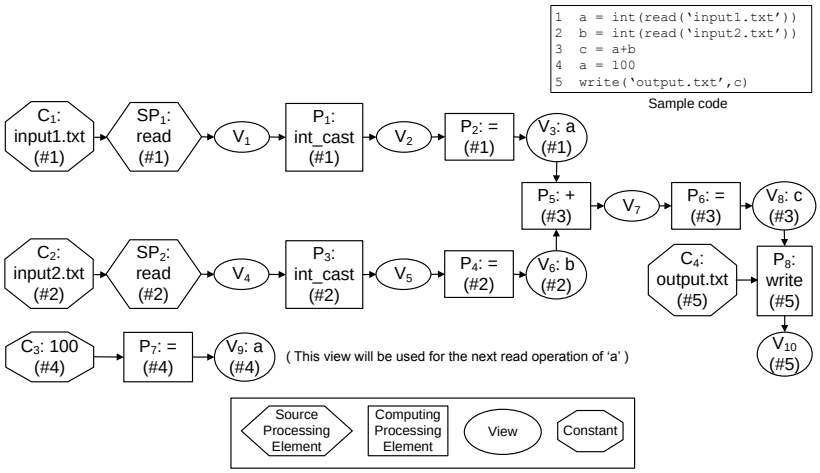


Figure 3.2: Example of the initial workflow provenance graph

of line number property(see Section 3.1), indicating the line in the program from where that particular node is created. Since in this example definition of both source and computing processing elements are limited to single line, the graph shows only one line number per processing element.

Handling ordering between operations

All operations/activities in the given program exhibit data dependencies. However, there is an implicit control dependency between the operations which determines the order of the execution of operations. Control dependencies occurred due to the ordering between operations, are translated into data dependencies by correlating a variable read of an operation to the latest version of that variable available along the control flow preceding the current operation. As an example, in line 3 when two variables `a` and `b` are summed up, the processing element P_5 reads the latest version of both `a` and `b` which are hold by the views V_3 and V_6, respectively. Once variable `a` is reassigned, the view V_9represents the current version of `a`. The view V_9 will be used for any later references of variable `a` until it is reassigned again. Therefore, the initial workflow provenance graph shown in Figure 3.2 transforms the implicit control dependencies between operations/activities into data dependencies by introducing different versions of the same variable based on their read and write sequences.

Other explicit control dependencies in an initial workflow provenance graph must be also transformed into data dependencies so that the graph could represent workflow provenance, i.e., relationship between activities based on the availability of data. In the next section, we describe trans-

formation function, consisting of a set of *flow transformation re-write rules* to infer a workflow provenance graph by transforming control-flow based coordination scheme into data-flow based coordination.

3.6 FLOW TRANSFORMATION RE-WRITE RULES

The transformation function includes a set of *flow transformation re-write rules*, transforming control dependencies into data dependencies. We define a re-write rule with two parts: left-hand side (LHS) and right-hand side (RHS). Once a rule is defined and is executed, it searches for the isomorphic sub-graph equivalent to the sub-graph pattern mentioned in the LHS of the rule. If the isomorphic sub-graph is found, it is replaced by the sub-graph pattern mentioned in the RHS of the rule.

In this chapter, we consider six types of operations/activities involving control dependencies: i) conditional branching (e.g. `if-elif-else`), ii) looping constructs (e.g. `for`) and iii) user-defined function/subroutine call (e.g. passing parameters to a defined function and assigned the returned value into a variable), iv) object instantiation of a user-defined class, v) exception handling by `try-except-finally` block and vi) `with` statement. Please note that, this is not a complete list of control-flow based operations available in the Python programming language. However, these are the most commonly used control-flow based operations in Python.

3.6.1 Conditional Branching

Conditional branching refers to the execution of a set of statements only if some condition is met. A conditional branching statement exhibits control-flow based coordination and is translated into data-flow based coordination by correlating a variable read of an activity in a conditional branch to the latest version of that variable available before the conditional branching language construct. All conditional branches are represented as parallel data dependencies where each branch contains an additional activity with variable ratio, i.e., selectively forwarding the data product based on the condition. Then, all parallel branches are condensed into a single data dependency again using a union activity.

LHS of Figure 3.3 shows the sub-graph pattern that could be found in the initial workflow provenance graph if a conditional branching is

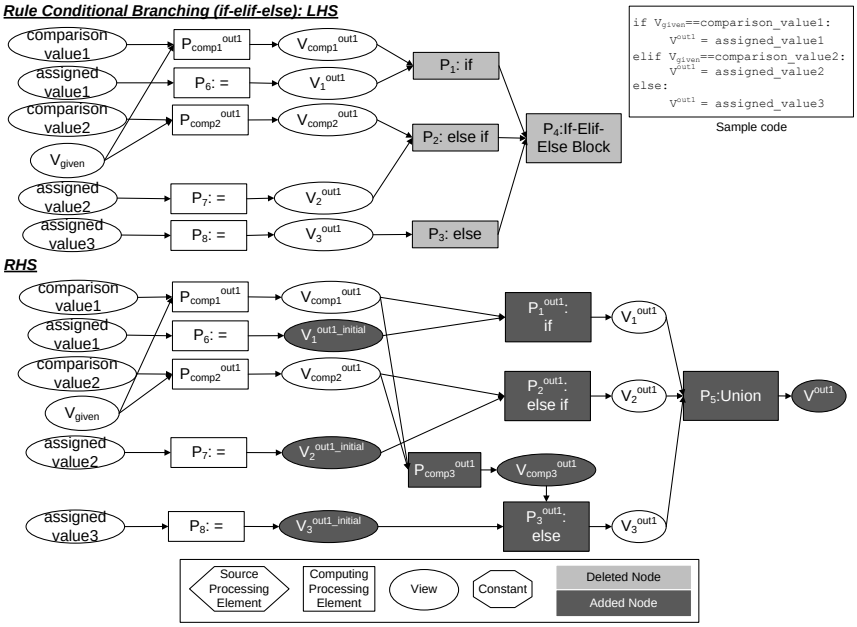


Figure 3.3: Re-write rule for conditional branching

Pattern

defined in a given program. In the initial workflow provenance graph, a computing processing element (e.g. P_1, P_2, P_3) is created for each of these conditional branches. Except the processing element (P_3) which is used to represent the `else` branch, other processing elements (P_1, P_2) have two parts: i) conditional part and ii) activity part if the condition is met, connecting towards itself. For P_1, the conditional part is originated from the node $comparison\_value1$ and the view V_{given}, i.e., `if` $(V_{given} == comparison\_value1)$. The node $assigned\_value1$ is the origin of the activity part which is then assigned into the variable V^{out1}, $(V^{out1} == assigned\_value1)$, if the condition is met. The same pattern could be found for `elif` branches. However, the `else` branch defines only the activity part if all other conditions are not met. Therefore, P_3, representing `else` branch, has the activity part only which assigns $assigned\_value3$ into V_{out1}. Due to these conditional branches, V^{out1} may hold different values depending on the condition that is satisfied. Therefore, we denote different versions of V^{out1} as the view nodes ${V_1}^{out1}$, ... , ${V_n}^{out1}$ where n is the total number of conditional branches in the current scope.

To transform the control dependencies into data dependencies in a conditional branching statement, we use the concept introduced in a program

Transformation

representation graph [67]. In a program representation graph, after every conditional branch one extra node is added to represent the output variable to follow static single assignment forms [31]. Therefore, in the RHS of Figure 3.3, we replace the nodes $V_1{}^{out1}$, $V_2{}^{out1}$, $V_3{}^{out1}$ with the nodes $V_1{}^{out1\_initial}$, $V_2{}^{out1\_initial}$, $V_3{}^{out1\_initial}$, representing the potential value to be assigned if that particular branch satisfies the condition. After checking the condition, the value could be assigned to any of these nodes $V_1{}^{out1}$, $V_2{}^{out1}$, $V_3{}^{out1}$. Therefore, they have been placed after P_1, P_2, P_3 respectively. Moreover, an explicit conditional part, connected towards P_3 (`else` branch), must be added to transform control dependencies into data dependencies successfully. Therefore, a processing element $P_{comp3}{}^{out1}$ which takes the output of the conditional part from other branches as input and produces output of the conditional part for the `else` branch, is added. $P_{comp3}{}^{out1}$ performs a *NOR* operation over its input data. Therefore, if all other conditions are *false*, it will produce *true* as output allowing the activity part of `else` branch to perform activities.

Several activities could be carried out if a particular condition is met. Therefore, each processing element is decomposed into multiple instances where each instance of the same processing element handles exactly one activity and produces the corresponding output. In the RHS of Figure 3.3, processing elements $P_1{}^{out1}$, $P_2{}^{out1}$ and $P_3{}^{out1}$ represent the instance of P_1, P_2 and P_3 respectively, created for handling the assignment activity of the variable V_{out1}. Depending on the condition, only one of these nodes: $V_1{}^{out1}$, $V_2{}^{out1}$ and $V_3{}^{out1}$, actually holds the value of V_{out1}. Therefore, we add an *union* processing element to capture the data available in one of these nodes and the *union* processing element produces the view V^{out1}. RHS of Figure 3.3 shows the pattern after the transformation. The light and dark shaded nodes in Figure 3.3 represent the deleted and added nodes, respectively.

Example

A sample program having a conditional branching block and the corresponding initial workflow provenance graph is shown in the top part of Figure 3.4. First, the given program asks the user to choose either '+' or '-'. Depending on the choice, the program then launches the conditional branching block which either adds or subtracts two variables `a` and `b` and the result is assigned into variable `c`.

The top part in Figure 3.4 shows the initial workflow provenance graph exhibiting explicit control dependencies due to the conditional branching. Executing the aforesaid re-write rule first identifies an isomorphic sub-

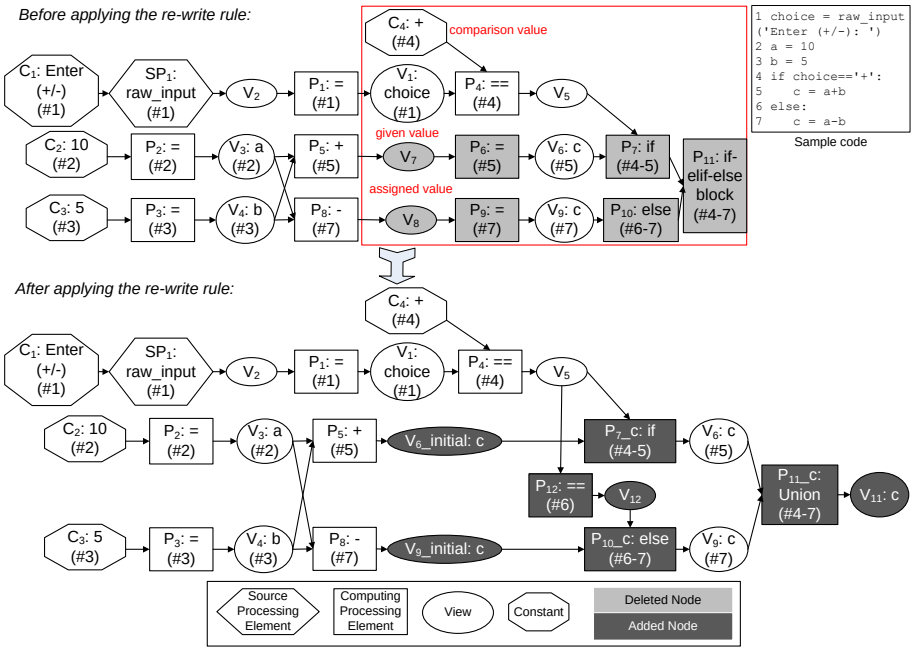


Figure 3.4: After applying the re-write rule for conditional branching on the given graph

graph equivalent to the LHS sub-graph pattern. This isomorphic sub-graph consists of nodes within the surrounding rectangle shown in the top part in Figure 3.4. The re-write rule replaces the found sub-graph with the one mentioned in the RHS of the re-write rule. The light and dark shaded nodes in Figure 3.4 represent the deleted and added nodes, respectively. The processing elements P_7_c and P_{10}_c have variable *input-output ratio* whereas the processing element P_{11}_c, representing *union* operation, has constant 'many to one' *input-output ratio*. All processing elements have default trigger interval of 1 tuple and all associated views have default window size of 1 tuple.

3.6.2 Looping Constructs

In any programming language, loops could be used for different purposes. We identify two major operations of looping constructs. First, loops could be used for iterating over input data products only. As an example, the usage of a loop to iterate over several input files falls into this category. The other usage of a loop is to manipulate input data products to produce

new output data products. As an example, the usage of a loop to produce running sum over a defined range of data tuples, executing at a fixed interval, falls into this class. In the former case, the trigger interval of the activity performing the looping operation is 1 time unit and the window size of views involved with that activity is also 1 time unit. The time unit depends on the period over which data are collected (e.g. hourly, daily, monthly, yearly basis). In the later case, a loop which manipulates input data products and produces new data products, can be implemented in several ways. The activity realizing a loop could either facilitate a special data structure like ring buffer to produce a result or define a conditional statement within its body based on which it performs the intended operation (e.g. conditional loop). The loop could be also used to produce a result from a given set of data products in an incremental manner (e.g. sum of elements in an array). In each aforesaid case of using loops, the activity, realizing a loop, shows control-flow based coordination which has to be transformed into data-flow based coordination by inferring the window size of associated views and trigger interval of the activity.

Please note that, we address only two cases of using a loop in this section. First, we describe the case where a looping construct is used to iterate over input files (data products). Second, we explain the mechanism of inferring data dependencies from a loop which produces a result by manipulating a set of input data products in a straightforward way, i.e., without involving any condition or a special data structure.

Loop iterating over input files (data products)

Pattern

LHS of Figure 3.5 shows the sub-graph pattern that could be found in the initial workflow provenance graph if the given program has a looping construct iterating over files (data products). Based on the sample code shown in Figure 3.5, each iteration of the loop reads an input file (e.g. `'sampleFile_1'`, `'sampleFile_2'` etc.) based on the value of the loop control variable, `i` (view V_2). The view V_2, referring to loop control control variable `i`, is produced by the processing element P_2, representing the defined `for` loop. P_2 controls the value of the loop control variable, `i`, based on the outputs of *range* function (P_1) which are hold by view V_1. V_1 contains the values ranging from `start` (inclusive) to `end` (exclusive) based on the given increment value (`incr`). In this case, the loop control variable `i`, represented by V_2, is used only by a source processing element (SP_1) that

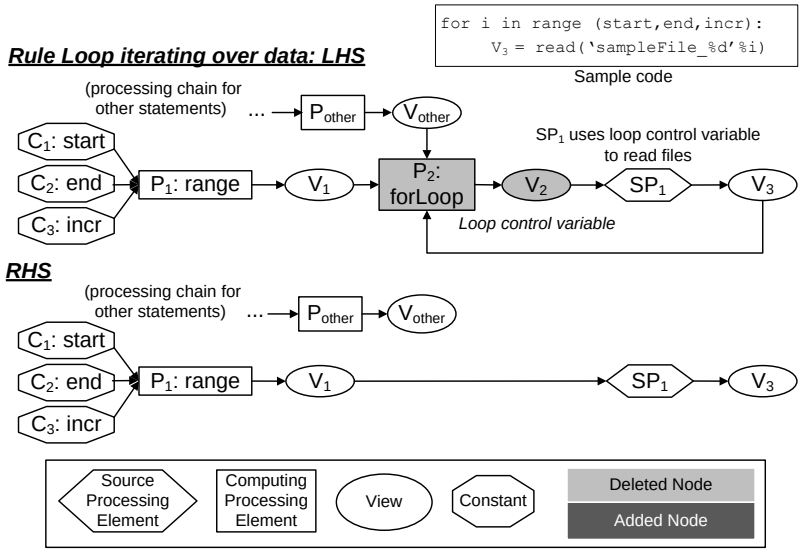


Figure 3.5: Re-write rule for a loop that iterates over files (data products)

reads input files, consisting of data products. Later, input data products are assigned into the view V_3 which has an outgoing edge to P_2, exhibiting control-flow based coordination.

Transformation

From the pattern depicted in LHS of Figure 3.5, we can see that the loop control variable (V_2) is only used to read a set of input files (data products), but is not used to manipulate them. However, there exists control dependencies between activities reading a particular input file, i.e., first, read `'sampleFile_1'` and then `'sampleFile_2'` and so on. To transform such control dependencies, we need to remove the processing element referring to the loop, P_2, and the corresponding loop control variable V_2 from the sub-graph pattern shown in LHS of Figure 3.5. Removing these nodes allows the source processing element (SP_1) to take one value/tuple at a time from V_1 as input which determines the input file to be read and thus transforms control dependencies into data dependencies. RHS of Figure 3.5 shows the data dependencies after the transformation where the execution of SP_1 only depends on the available data products in V_1. In this case, the window size defined over V_1 is 1 time unit. The trigger interval of SP_1 is 1 time unit and the *input-output ratio* is 'one to one'.

Example

A sample program having a looping construct and the corresponding initial workflow provenance graph is shown in the top part of Figure 3.6. The given program iterates over a set of input files, i.e., `'month_1'`, `'month_2'`

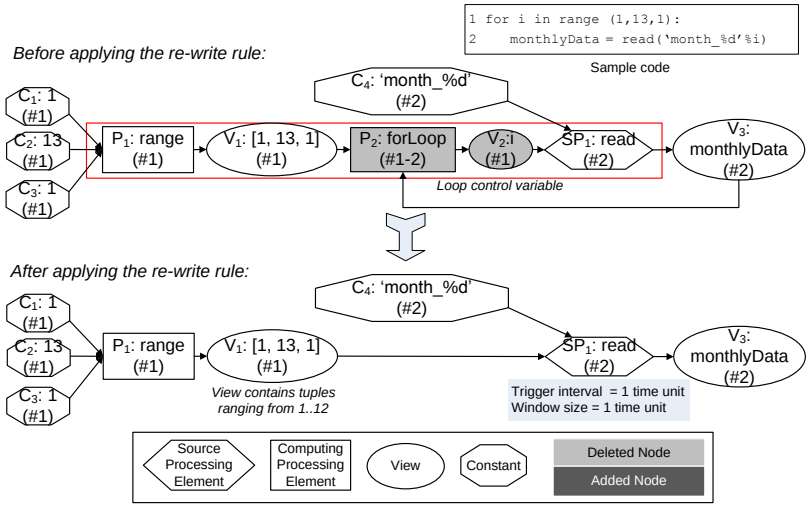


Figure 3.6: After applying the re-write rule for a loop iterating over files (data products) on the given graph

and so on, to read data products inside those files. The `read` method performs this file read operation, denoted by the source processing element SP_1, based on the loop control variable `i` (V_2) and the appropriate input file, i.e., `'month_i'`, represented by the constant node C_4. Later, SP_1 assigns data products of an input file which has been read into variable `monthlyData`, represented by the view node V_3.

Applying the aforesaid re-write rule on the initial workflow provenance graph shown in the top part in Figure 3.6 transforms the control dependencies occurred due to the looping constructs into data dependencies and the transformed graph is shown in the bottom part in Figure 3.6. The re-write rule finds the isomorphic sub-graph equivalent to the LHS of the rule shown by the nodes within a surrounding rectangle in Figure 3.6. The rule replaces this sub-graph pattern with the one which is mentioned in the RHS of the rule. The transformed graph is now data dependent where the source processing element SP_1 has trigger interval of 1 time unit and *input-output ratio* of 'one to one'. The window defined over V_1 has size 1 time unit.

Loop manipulating input data products and producing new data products

LHS of Figure 3.7 shows the sub-graph pattern that could be found in the initial workflow provenance graph if a loop is used to manipulate input

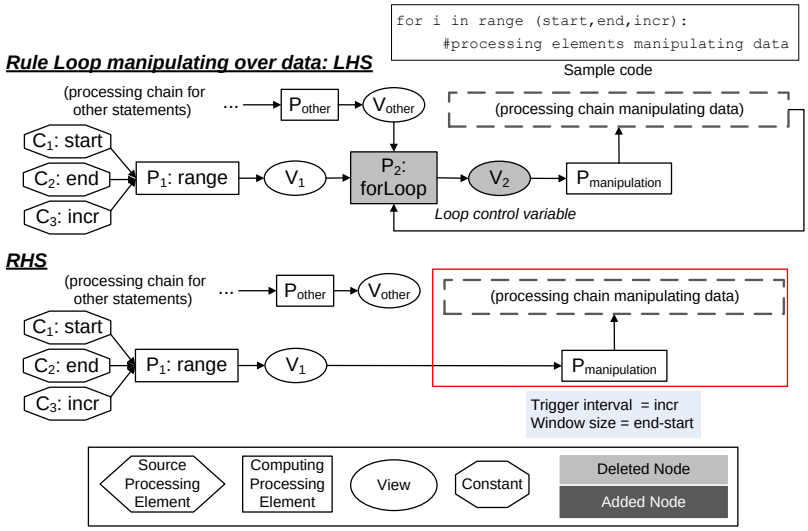


Figure 3.7: Re-write rule for a loop that manipulates data products

Pattern

data products to produce new data products. In Figure 3.7, P_2 represents the loop and V_2 is the view node created for representing the loop control variable, i. P_2 controls the value of the loop control variable, i, based on the outputs of *range* function (P_1) which are hold by view V_1. V_1 contains the values ranging from `start` (inclusive) to `end` (exclusive) based on the given increment value (`incr`). In this case, a processing element $P_{manipulation}$ manipulates input data products based on the value of the loop control variable i (V_2) and produces new data products. The processing element $P_{manipulation}$ and other processing elements within the processing chain could be executed several times and thus they exhibit control-flow based coordination.

Transformation

We can transform control dependencies into data dependencies by inferring not only window size defined over views, contributing to $P_{manipulation}$ and other processing elements in the chain but also the trigger interval of appropriate processing elements in the chain. A window size refers to the subset of data products within a view, participating in an activity. In this case, the first two parameters of the *range* function (P_1), `start` (inclusive) and `end` (exclusive) define the boundary [`start`, `end`), of the values hold by the loop control variable i. Since the loop control variable i is facilitated by $P_{manipulation}$ and other subsequent processing elements in the chain to manipulate input data products, the window size of participating views in the processing chain is `end`−`start`. The last parameter `incr` refers to the

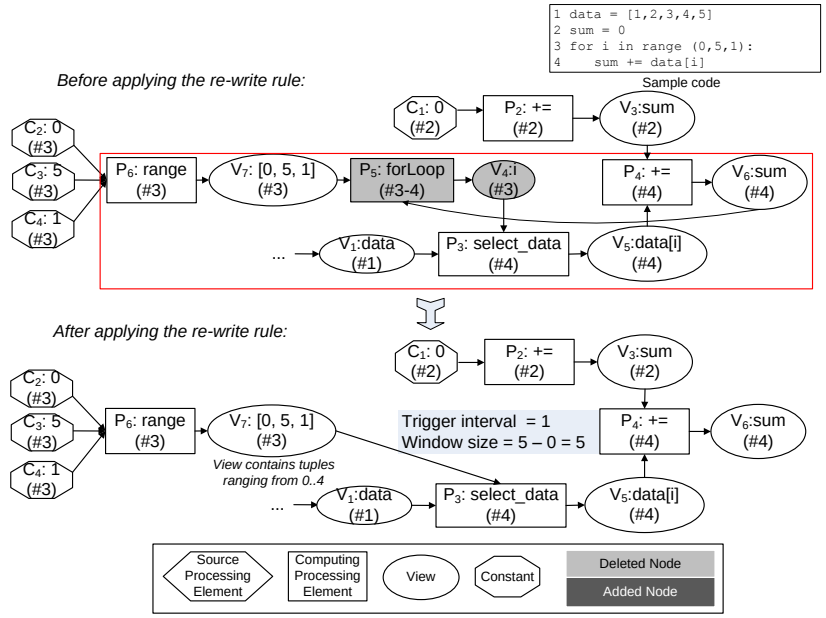


Figure 3.8: After applying the re-write rule for a loop manipulating data on the given graph

increment of the value of i and thus, it represents the trigger interval of the manipulating processing element $P_{manipulation}$ and other successive processing elements in the chain. The processing elements enclosed within the rectangle in RHS of Figure 3.7 have the inferred window size and triggers. The *input-output ratio* of these processing elements depends on the nature of their processing.

Example

The top part in Figure 3.8 shows an initial workflow provenance graph which is created based on a given program with a looping construct, producing a `sum` value by manipulating data products in an array, named as `data`. First, the given program initializes the array, `data` and the variable, `sum`. Afterward, the program executes a looping constructs that iterates over data products within the array `data` to calculate the sum of all data products. The processing element node P_5 represents the defined loop and the view node V_4 represents the loop control variable i. The boundary of the values of i starts from the value 0 (inclusive) to 5 (exclusive) and is incremented by 1 after the end of every iteration depending on the parameters given to the *range* function (P_6).

The initial workflow provenance graph shown in the top part in Figure 3.8 does not show the initialization of `data` array, represented by V_1, to

reduce the complexity of the graph. As we can see from the top part in Figure 3.8, V_4 is connected to the processing element node P_3 which is used to select a particular data product from the defined array `data`. The index of the selected data product depends on the loop control variable, i.e., P_3 selects i^{th} element from the array `data`. Please note that, P_3 has a constant *input-output ratio* ('one to one') unlike a selection operation applied over a view in a database. Later, the selected data product `data[i]`, represented by V_5, contributes to the addition operation, represented by P_4. P_4 takes one data product from `data` array at a time as input, calculates a sum of these values incrementally and assigns the computed value into variable `sum`, represented by V_6. Therefore, P_4 maintains control-flow based coordination during its execution.

Applying the aforesaid re-write rule on the initial workflow provenance graph transforms control dependencies occurred due to the looping constructs into data dependencies. The transformed graph is shown in the bottom part in Figure 3.8. The re-write rule finds the isomorphic sub-graph pattern equivalent to the LHS of the rule (see top part in Figure 3.7), as shown by the nodes within a surrounding rectangle in top part of Figure 3.8. It replaces this sub-graph with the one which is mentioned in the RHS of the rule (see bottom part in Figure 3.7). In this case, the processing element P_4 has trigger interval of 1 tuple/data product and the window size defined over the view V_5 is 5 tuples which is calculated based on the parameters of the *range* function, represented by P_6.

3.6.3 User-defined Function/Subroutine Call

A function or subroutine call executes a set of statements defined in the body of the function and afterward, the flow of control usually returns to the activity which calls that particular function. In this case, the successful execution of the caller activity and other activities to be executed after the caller activity depends on the successful completion of the activities defined within the function body. Therefore, a function call exhibits control-flow based coordination between activities.

Pattern

To transform the control dependencies into data dependencies in a user-defined function call, we replicate the nodes found inside the function body into the place where the caller activity calls the function. LHS of Figure 3.9 shows the sub-graph pattern for a function call which could be found in the initial workflow provenance graph. The user-defined function

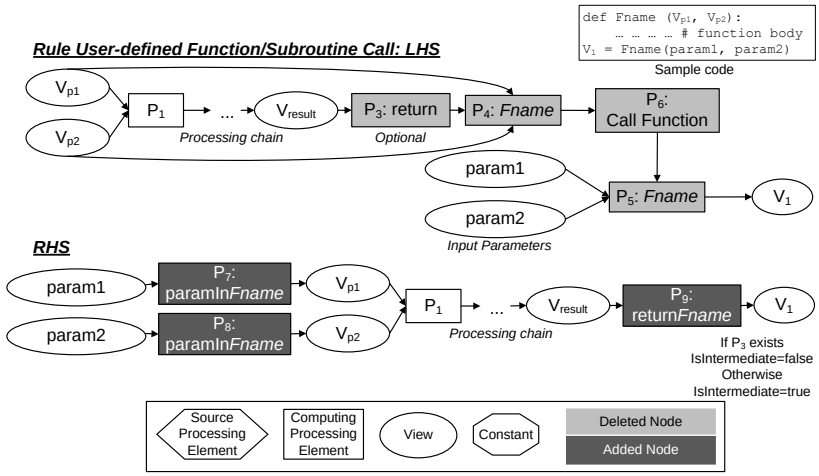


Figure 3.9: Re-write rule for a user-defined function/subroutine call

`Fname` is defined and later in the code it is called. The processing element P_5 represents the caller activity and passes input parameters, represented by `param1` and `param2` nodes, to the function `Fname`, denoted by the processing element P_4. P_6 connects the caller P_5 to the function body hold by P_4.

Transformation

To transform control dependencies into data dependencies, we introduce two specific activities: *paramIn* and *return*. The *paramIn* activities take parameters from the caller P_5 as input and then connects them to the parameters mentioned in the function definition hold by P_4. Then, the processing chain defined within P_4 is replicated. The *return* activity takes the returned value from the function if there is any and assigns the value into the view V_1 with *IsIntermediate=false*. Otherwise, it assigns an intermediate value into V_1 (*IsIntermediate=true*). RHS of Figure 3.9 shows the pattern after the transformation.

Example

The top part in Figure 3.10 shows a sample program having a user-defined function call and the corresponding initial workflow provenance graph. The given program includes a user-defined function, named as `add`, which takes two input parameters: `x` and `y`. The function `add` calculates the sum of these two values and returns the value to the caller activity. The main body of the program calls the function `add` with the values 10 and 5 and assigns the return value into variable `a`.

The initial workflow provenance graph shown in the top part in Figure 3.10 bridges the caller activity/processing element P_5 and the called activ-

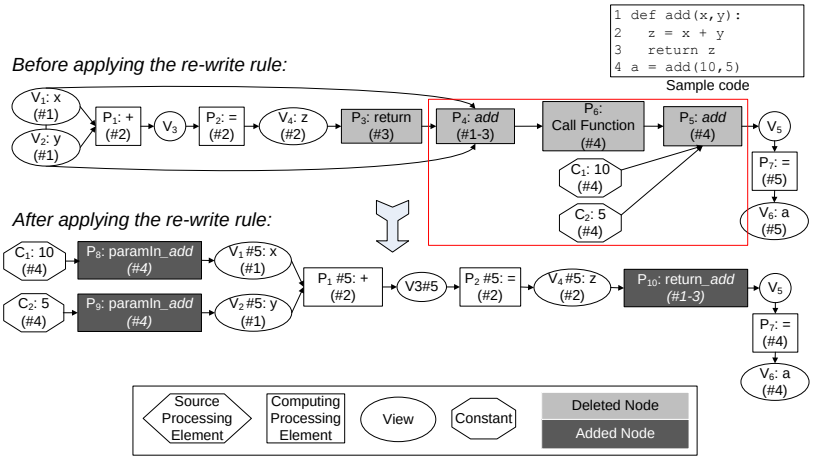


Figure 3.10: After applying the re-write rule for a user-defined function call on the given graph

ity/processing element P_4 (the node holds the body of the function), by introducing the node P_6 in between them. Once the aforesaid re-write rule is applied over this initial workflow provenance graph, the rule detects a match to its LHS sub-graph pattern. It then introduces the nodes P_8 and P_9, representing the *paramIn* activity, which take the constant nodes C_1 and C_2, representing the value 10 and 5 and assign these values to the nodes within the function body, V_1 and V_2, respectively. From this point onwards, the nodes within the function body are replicated and placed accordingly till the node P_3 is reached, representing the `return` activity within the function. The replicated nodes also include the *Line#* property just after the node identifier, i.e., P_1#5, indicating the line number in the main body of the program where the caller processing element calls the function. Eventually, by introducing the node P_{10}, representing the `return` activity from the function, the return value is assigned into view node V_6, representing variable a. In this case, the trigger interval is 1 for all processing elements and the window size of associated views is also 1. However, the *input-output ratio* of the processing elements defined in the function must be given by users before starting the execution of workflow provenance inference method because the *input-output ratio* of a user-defined function cannot be inferred by analyzing the given program.

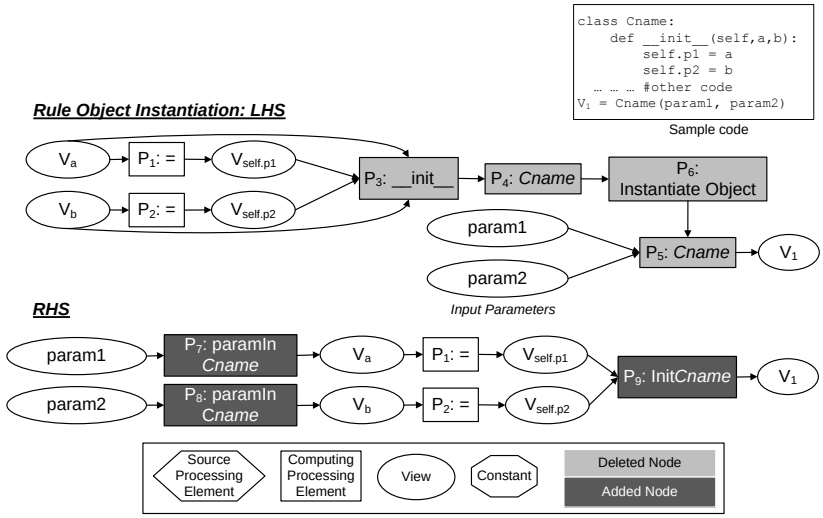


Figure 3.11: Re-write rule for an object instantiation of a user-defined class

3.6.4 Object Instantiation of a User-defined Class

Python supports object-oriented programming where a class can be defined and the object, instance of the class, can be instantiated. Instantiating an object of a particular class is accomplished by executing the constructor method of the class, after which the control flow returns to the caller activity. In Python, the constructor method is usually known as `_init_`. The successful execution of the caller activity and other activities to be executed after the caller activity depends on the successful completion of the constructor of the class instantiating an object of that class. Therefore, object instantiation of a user-defined class exhibits control-flow based cooperation between activities.

Pattern

LHS of Figure 3.11 shows the sub-graph pattern for an object instantiation activity, represented by the processing element P_5. Similar to the LHS sub-graph pattern shown in Figure 3.9, P_5 passes input parameters, represented by `param1` and `param2` nodes, to the `_init_` function (denoted by P_3) of the class defined by P_4. P_6 connects the caller activity/processing element P_5 to the class definition hold by processing element P_4.

Transformation

RHS of Figure 3.11 shows the transformed graph that exhibits data dependencies between processing elements. We follow the same approach which transforms control dependencies occurring due to a function/subroutine call into data dependencies as discussed in Section 3.6.3. The *paramIn*

activities take parameters from the caller P_5 as input and then connects them to the parameters mentioned in the `__init__` function defined within the body of the class, represented by P_4. Then, the input parameters are assigned into the instance variables of the class represented as $V_{self.p1}$ and $V_{self.p2}$. The `_init_` method assigns these values into the view V_1 which represents the instantiated object of the class `Cname`. In Figure 3.11, the default trigger interval is 1 for all processing elements and the default window size of associated views is also 1. The *input-output ratio* of object instantiation activity/processing element (P_9) is $n : 1$ where n is the number of parameters to the constructor method.

Another type of statements which exhibits control dependencies is accessing a member function of a particular class. Since this statement also operates in a similar fashion as the object instantiation does, the re-write rule to transform the control dependencies occurred due to this statement is same as the aforesaid rule shown in Figure 3.11. Therefore, it is not discussed here. However, unlike object instantiation processing element, the *input-output ratio* of the processing elements representing the function must be given by users beforehand.

Example

The top part in Figure 3.12 shows source code of a given program with the example of object instantiation of a particular class followed by the example of accessing a member function of the same class in line 8 and 9 respectively. The first 7 lines are used to define the class `person` which has two instance variables: `self.first` and `self.last`, representing the first and last name of a person, respectively. Furthermore, the class definition also contains a member function `show` which concatenates the first and last name of an object of type `person` class and returns the value to the caller activity/processing element.

The top part in Figure 3.12 also shows the initial workflow provenance graph created for the given program. The processing element node P_{10} represents the object instantiation of the class `person` in the main body of the program. The caller activity is represented by the node P_9 and the class `person` is represented by the node P_8. These three nodes are surrounded by a rectangle with solid line in the top part in Figure 3.12, representing the LHS sub-graph pattern shown in Figure 3.11. Applying the re-write rule over this initial provenance workflow graph results into the nodes surrounded by a rectangle with solid line, depicted in the bottom part in Figure 3.12.

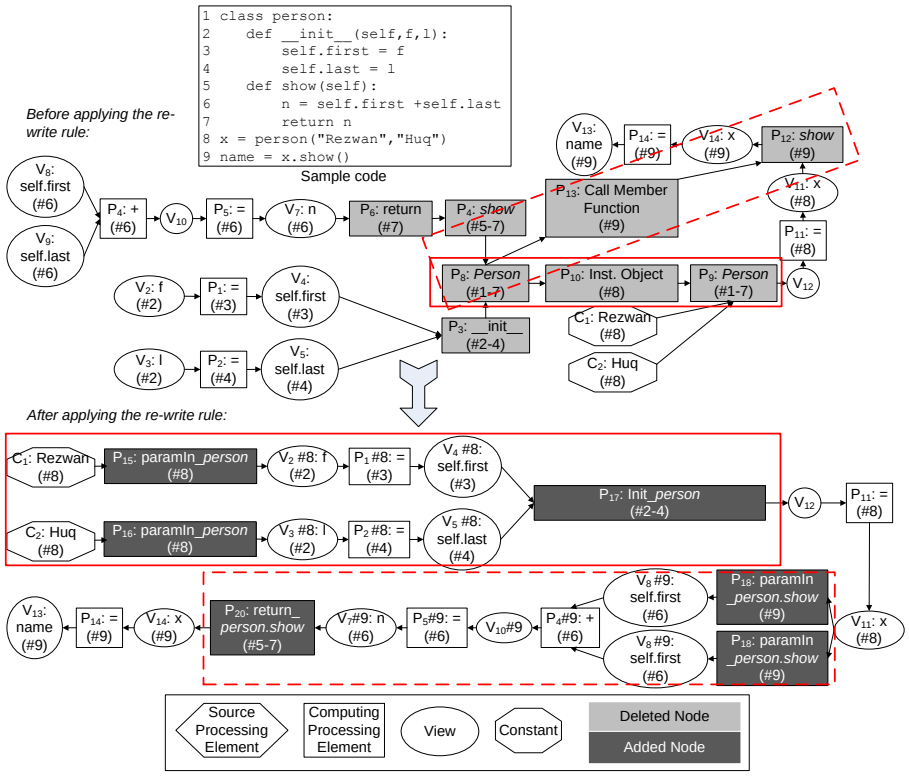


Figure 3.12: After applying the re-write rule for an object instantiation on the given graph

Moreover, the given program also contains a caller activity, denoted by the node P_{12}, which calls the member function `show`, represented by the node P_4. The nodes surrounded by a rectangle with dashed line in the top part in the Figure 3.12 represent this activity. The control dependencies in the initial workflow provenance graph (top part in Figure 3.12) can be transformed into data dependencies by applying the re-write rule as shown in Figure 3.11. After the transformation, it results into the nodes surrounded by a rectangle with dashed line in the bottom part in Figure 3.12. The provenance graph shown in the bottom part in Figure 3.12 is the transformed graph for the given program.

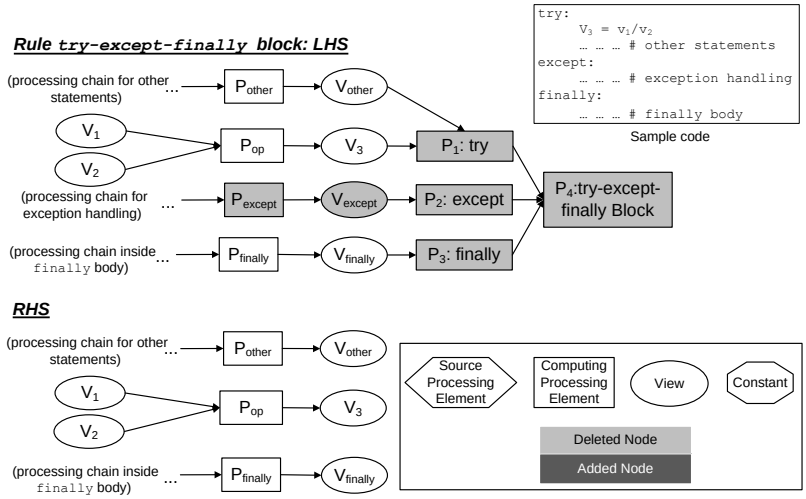


Figure 3.13: Re-write rule for Exception Handling using `try-except-finally` block

3.6.5 Exception Handling using `try-except-finally` block

It is possible to write a Python program that can handle selected exceptions using `try`, `except` and `finally` keywords. In such type of a Python program, the `try` block contains a set of statements that will be executed by the program. If an exception is raised during the execution, the set of statements within the `except` block is executed as a *fail-safe* approach. Eventually, the set of statements defined within the `finally` block is executed irrespective of whether an exception is raised or not. Statements inside a `finally` block can also update variables defined within corresponding `try` block. Furthermore, execution of a statement/activity within an `except` block depends on a specific condition that is triggered by executing statements within corresponding `try` block. Therefore, the activities defined within these blocks exhibit control-flow based cooperation.

Pattern

LHS of Figure 3.13 shows the sub-graph pattern that could be found in the initial workflow provenance graph in case the given program contains any exception handling block. The processing element node P_4 represents the complete `try-except-finally` block, consisting of statements within `try`, `except` and `finally` block, represented by processing elements P_1, P_2 and P_3, respectively.

If an exception is raised by one of these statements within the `try` block, the intended result of the program cannot be achieved and thus, the program execution becomes unsuccessful. Since a workflow provenance graph represents the data dependencies of a successful program execution, we decide not to consider the case when an exception is raised. Therefore, the defined re-write rule, depicted in Figure 3.13, only considers the statements within the `try` and the `finally` block and transforms control dependencies into data dependencies within these two blocks. The re-write rule replaces any isomorphic sub-graph equivalent to the LHS of the Figure 3.13 with the sub-graph pattern mentioned in the RHS of Figure 3.13. As already mentioned, only the nodes associated with the `try` and the `finally` block retain in the transformed graph.

Transformation

An alternative approach to deal with the `except` block could be representing activities within an `except` block as a parallel branch to the activities defined within corresponding `try` block which is similar to the approach taken to represent conditional branching statements (see Section 3.6.1). This alternative approach could be more useful to examine a situation where an exception is raised so that debugging becomes easier. However, the resulting workflow provenance graph produced by this alternative approach could be larger and more complex to understand in some cases where the `except` block is comprised of many statements.

Figure 3.14 shows a sample program with an exception handling block and the corresponding initial workflow provenance graph and the transformed provenance graph after applying the re-write rule. In the given program, two variables `a` and `b`, represented by the nodes V_1 and V_2, are assigned with the values 12 and 3, denoted by the constant nodes C_1 and C_2, respectively, shown in the top part in the Figure 3.14. In the `try` block, the program calculates the division of these two variables and assigns the outcome into variable `c`, represented by the view node V_3. If the aforesaid operation raises any exception such as *divide by zero* error, the statement within the `except` block will be executed. Eventually, variable `c` is reassigned with the value `0` in `finally` block.

Example

The bottom part in the Figure 3.14 shows the transformed graph after applying the aforesaid re-write rule, depicted in Figure 3.13. As discussed in the re-write rule, only the nodes associated with the `try` block, and the `finally` block retain and the other nodes are deleted from the transformed graph. Since, variable `c` is reassigned within `finally` block, the updated version of `c` is represented by view V_5. Later, if any activity/state-

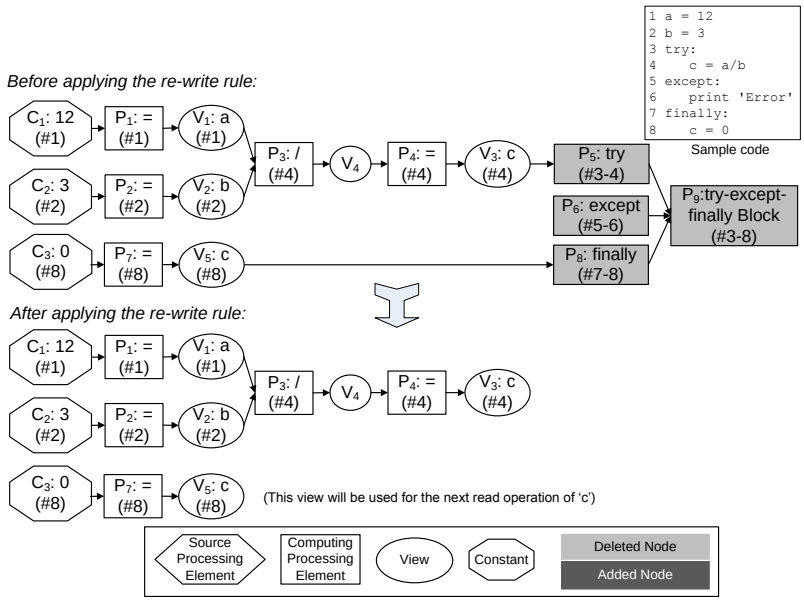


Figure 3.14: After applying the re-write rule for exception handling using `try-except-finally` block on the given graph

ment wants to access variable `c`, view V_5 will participate in that particular activity. All processing elements in Figure 3.14 have default trigger interval of 1 tuple and all associated views have default window size of 1 tuple. The *input-output ratio* depends on the nature of the processing.

3.6.6 Handling `with` statements

The Python programming language incorporates `with` statement from version 2.7. The `with` statement is used to wrap the execution of a block with methods defined by a context manager which handles the desired runtime context for the execution of that block of code. In other words, a `with` statement guarantees to enter the `_enter_` method (a runtime context) defined within the scope of the object to bind this method's return value to the target specified in the `as` clause of the statement, if any. It also guarantees to call the `_exit_` method (another runtime context) defined within the context of the object to ensure that irrespective of whether an exception is raised by the block defined within the `with` statement or not, the program continues to execute the statements found within the scope of the `_exit_`

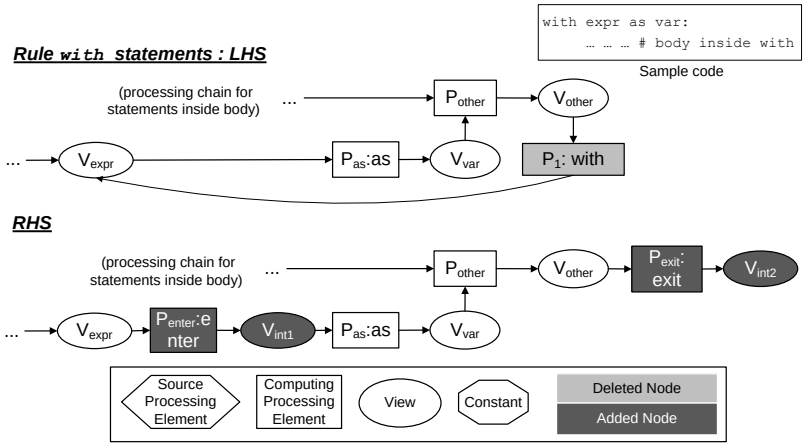


Figure 3.15: Re-write rule for handling `with` statements

method. As an example, a `with` statement can be used with a file object to ensure the closing of the file in case of a raised exception or not.

Since, a `with` statement calls at least two member functions- `_enter_` and `_exit_`, it exhibits control dependencies which has to be transformed into data dependencies. Figure 3.15 shows both LHS and RHS of the re-write rule handling a `with` statement. LHS of Figure 3.15 depicts the sub-graph pattern which is to be matched with any isomorphic sub-graph in the initial workflow provenance graph. If an isomorphic graph equivalent to this pattern is found, the corresponding part is replaced by the sub-graph pattern shown in RHS of Figure 3.15. In the RHS, two processing element nodes, P_{enter} and P_{exit}, and their corresponding output view nodes V_{int1} and V_{int2}, are added for the `_enter_` and the `_exit_` method respectively assuming that the body of these methods is not defined in the given program. However, if the body of these methods are defined within the given program, we apply the re-write rule for a user-defined function call, shown in Figure 3.9 in the next step. Since the use of a `with` statement is very rare, we do not provide any sample program and its corresponding initial workflow provenance graph and transformed graph.

Patter & Transformation

3.7 GRAPH MAINTENANCE RE-WRITE RULES

After applying the transformation function, consisting of a set of *flow transformation re-write rules*, over the initial workflow provenance graph, the

maintenance function is applied next. The maintenance function contains a set of re-write rules, called *graph maintenance re-write rules*. This set of re-write rules are defined to ensure the propagation of persistence of views from one to another as well as to discard unnecessary intermediate views followed by the assignment processing element. Furthermore, one of the rules in this set helps scientists to identify the computing processing element which generates persistent output data products.

Propagating persistence of views

Figure 3.16 shows three graph maintenance re-write rules. The first re-write rule, *GM 1*, propagates the persistence of a view to the next one if some conditions hold. This re-write rule ensures that if scientists use Python methods such as `read` to read input data products from persistent storage and to assign this data into a variable, the corresponding view created for the variable will be also persistent (*IsPersistent=true*). LHS of the re-write rule *GM 1* in Figure 3.16 shows the sub-graph pattern for such an activity. The intermediate view V_1 is produced by the `read` method and contains persistent data. Later, the persistent data hold by V_1 is assigned into a variable represented by V_2 through the processing element node P_1. In this case, the value of *IsPersistent* property of view V_1 is propagated towards view V_2 and V_2 becomes also persistent *(IsPersistent=true)*. RHS of the rule *GM 1* in Figure 3.16 shows the updated property of V_2. The nodes with updated property values are highlighted with a dark shade, applicable for all graph maintenance re-write rules.

Discarding intermediate views

Rule *GM 2* minimizes the size of the workflow provenance graph. It deletes all intermediate views *(IsIntermediate=true)* and subsequent assignment process nodes *(name = '=')* if they are followed by a view representing a variable defined in the program, i.e., *(IsIntermediate=false)*. It has two variants depending on the type of the node which produces the intermediate view (either a source processing element, SP_1 or a computing processing element, P_1) shown in the LHS of rule *GM 2.a* and *GM 2.b*, respectively in Figure 3.16. Executing these re-write rules, discards the light-shaded nodes from the initial workflow provenance graph and makes a connection between SP_1 and V_2 as well as between P_1 and V_2 for the rules *GM 2.a* and *GM 2.b*, respectively.

Identifying processing element produced output

Rule *GM 3* identifies the computing processing element which generates a persistent result, i.e., the result that is written into the disk. In Python, there are a few methods such as `write`, `report` which write data into the disk. However, these methods do not compute the data rather they write the data produced by another processing element. Therefore, the process-

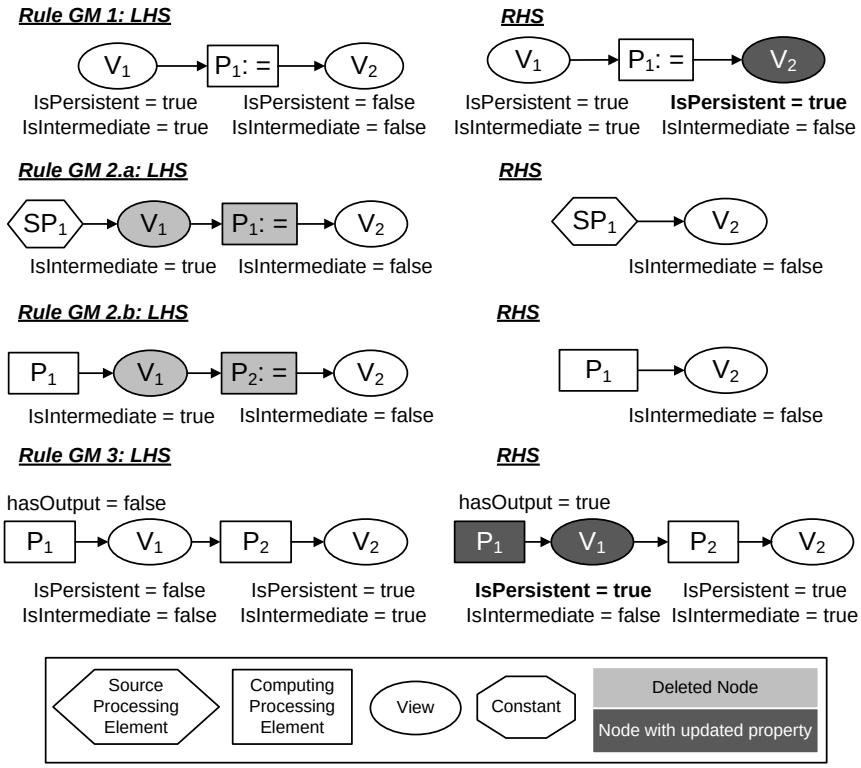


Figure 3.16: Re-write rules for graph maintenance

ing element which produces the data that is written into the disk later, is the computing processing element generating persistent output. LHS of the rule *GM 3* in Figure 3.16 shows the sub-graph pattern for the aforesaid activities. P_2 is the processing element representing the method like `write` or `report` and generates a view node V_2 which refers to the data written into the disk. Before this activity takes place, P_2 had taken V_1 as input and V_1 is a non-intermediate view, indicating that V_1 represents a defined variable in the program which contains the data written by P_2. Therefore, the processing element P_1 which produces V_1 is the computing processing element having persistent output data. It is represented by *hasOutput=true* value. RHS of the rule *GM 3* in Figure 3.16 shows the processing chain with the updated values of relevant properties.

Step-by step transformation

The outcome of the actual execution of these graph maintenance re-write rules are shown in Figure 3.18. First, the initial workflow provenance graph created for the given Python program shown in Figure 3.2 is depicted in Figure 3.17. In Figure 3.17, the view nodes V_2 and V_5 are both intermediate,

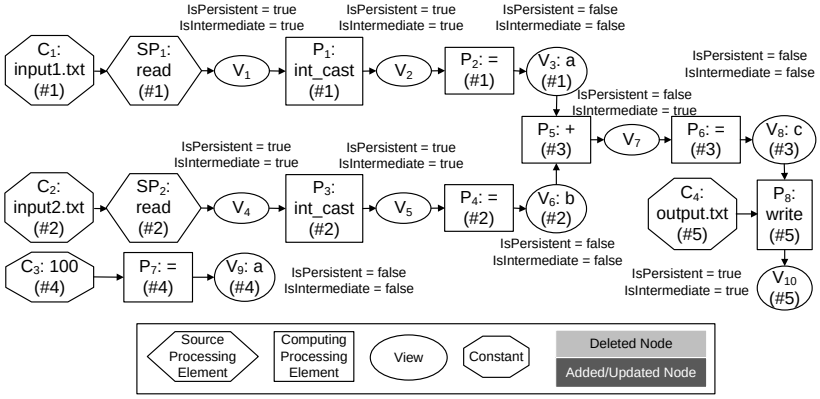
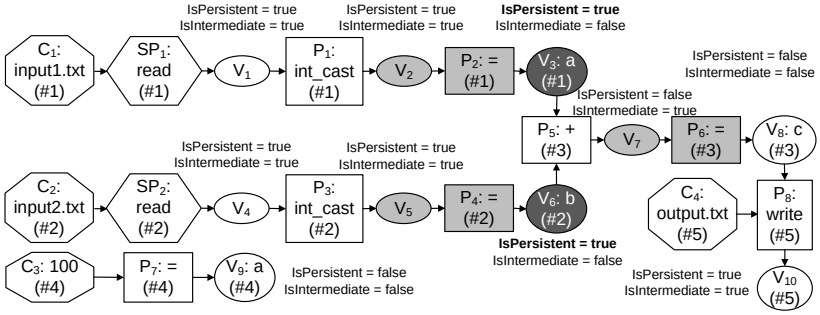


Figure 3.17: Initial workflow provenance graph (before applying graph maintenance re-write rules)

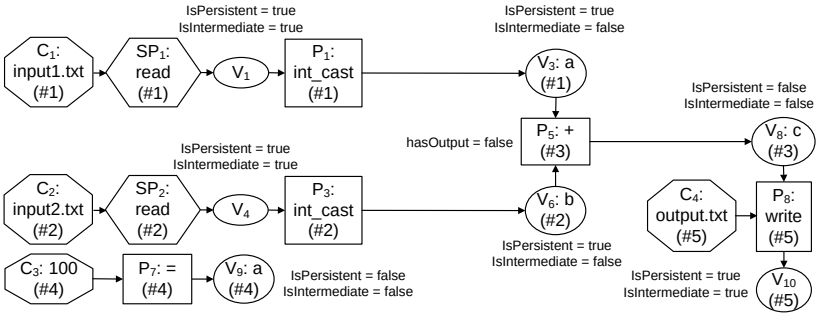
but are persistent as both nodes contain the result of the `read` method, represented by the source processing element nodes SP_1 and SP_2 respectively. The values hold by these intermediate and persistent views are assigned into the view nodes V_3 and V_6. Since these nodes used to represent the defined variables in the given program, they are non-intermediate *(IsIntermediate=false)* and by default are non-persistent *(IsPersistent=false)* also. This information on the nodes properties will be used to apply the first graph maintenance re-write rule *GM 1* that propagates the persistence of views from one to the other.

Applying the re-write rule *GM 1* over the initial workflow provenance graph shown in Figure 3.17, updates the value of the *IsPersistent* property of nodes V_3 and V_6 from `false` to `true`. The provenance graph after applying this re-write rule is shown in Figure 3.18a.

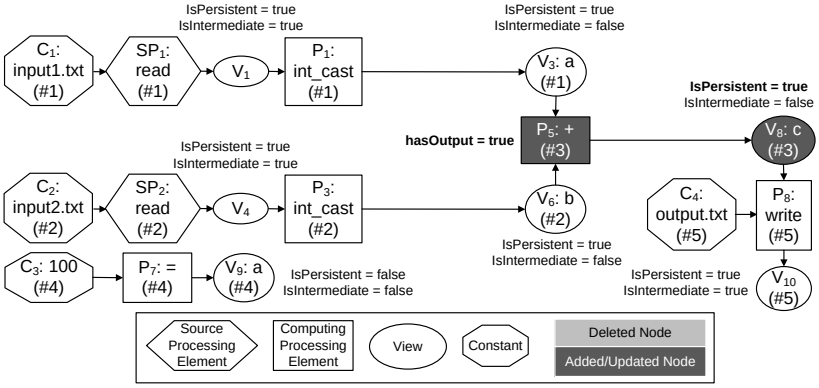
Next, the re-write rule *GM 2* is applied over the provenance graph shown in Figure 3.18a. This re-write rule discards intermediate views followed by a processing element performing an assignment operation. In Figure 3.18a, there are three isomorphic sub-graphs which are equivalent to the sub-graph pattern mentioned in the LHS of the re-write rule *GM 2.b* (see Figure 3.16). These are: i) $P_1 \rightarrow V_2 \rightarrow P_2 \rightarrow V_3$, ii) $P_3 \rightarrow V_5 \rightarrow P_4 \rightarrow V_6$ and iii) $P_5 \rightarrow V_7 \rightarrow P_6 \rightarrow V_8$. Therefore, applying the aforesaid re-write rule discards the nodes V_2, P_2 and V_5, P_4 and V_7, P_6 for the first, second and third isomorphic sub-graph, respectively. The provenance graph after applying this re-write rule is shown in Figure 3.18b.



(a) After applying the re-write rule, *GM 1*



(b) After applying the re-write rule, *GM 2*



(c) After applying the re-write rule, *GM 3*

Figure 3.18: Step-by-step transformations of the initial workflow provenance graph

Finally, the last re-write rule among this set of graph maintenance rules, *GM 3*, is applied over the provenance graph shown in Figure 3.18b. In Figure 3.18b, the node P_8, representing the `write` method, produces a view V_{10}, which is persistent as the result hold by this view is written into the

disk. Since the node P_8 only takes the value hold by the view node V_8 as an input the re-write rule *GM 3* concludes that the processing element node P_5 is the node which actually produces a result, that is made persistent afterward by P_8. Therefore, applying this rule updates the value of a few properties of the node P_5 and V_8. The *hasOutput* of P_5 becomes *true* as well as the *IsPersistent* of V_8 also becomes *true*. The transformed provenance graph is shown in Figure 3.18c. This provenance graph will be considered by the next re-write rules if available.

3.8 GRAPH COMPRESSION RE-WRITE RULES

Finally, we execute the compression function, consisting of a set of graph compression re-write rules, over the initial workflow provenance graph. The main purpose of applying graph compression re-write rules is to reduce the number of nodes in the initial workflow provenance graph.

In the proposed workflow provenance model, described in Section 3.1, the *workflow provenance graph*, represented as a bipartite graph, has two major types of nodes. These are: i) data products, represented as view and constant nodes and ii) activities/processing elements, represented as source processing element and computing processing element nodes. Since the workflow provenance graph is a bipartite graph, no two data products as well as no two processing elements can be connected together. This property of a workflow provenance graph can be facilitated to reduce the size of a workflow provenance graph in two ways. First, a constant node which acts as an input to a source/computing processing element node, can be unified with the corresponding processing element and then, the constant node will be discarded from the graph. Second, a view node which is produced by a source/computing processing element can be unified with the corresponding source/computing processing element and afterward, the view node will be deleted from the graph. To ensure that no information is lost, we copy the properties of the output view or input constant nodes to the corresponding source or computing processing element node. This mechanism triggers a change of representing the workflow provenance model to reduce the complexity of the workflow provenance graph and to increase the readability. However, the model itself is not affected by this change in representation.

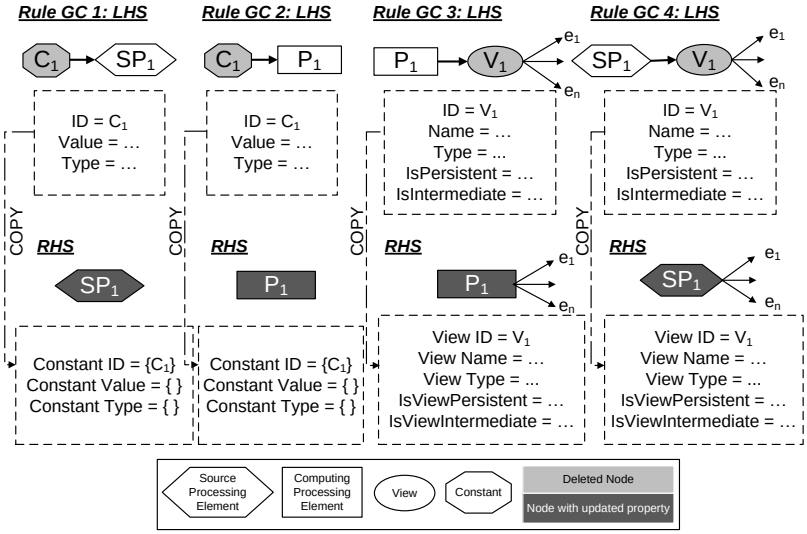


Figure 3.19: Re-write rules for graph compression

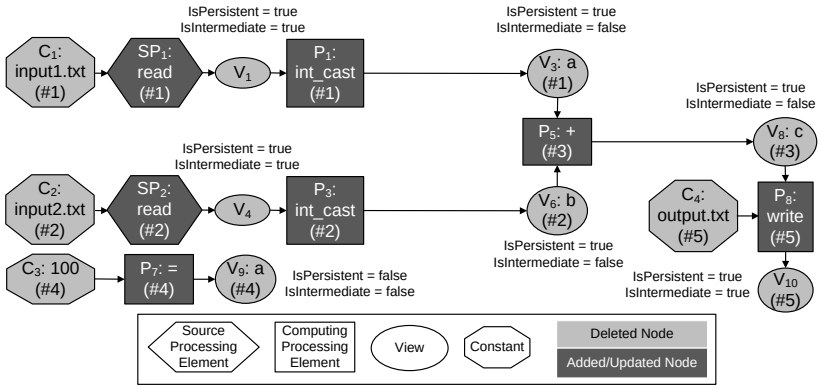
Therefore, to apply these graph compression re-write rules, we change the representation of the workflow provenance model and the new representation has two types of nodes: i) source processing element and ii) computing processing element. Both source and computing processing elements include properties of constant and view nodes so that we can copy these values of constants and views to the corresponding properties in the corresponding processing element.

Unifying input constants

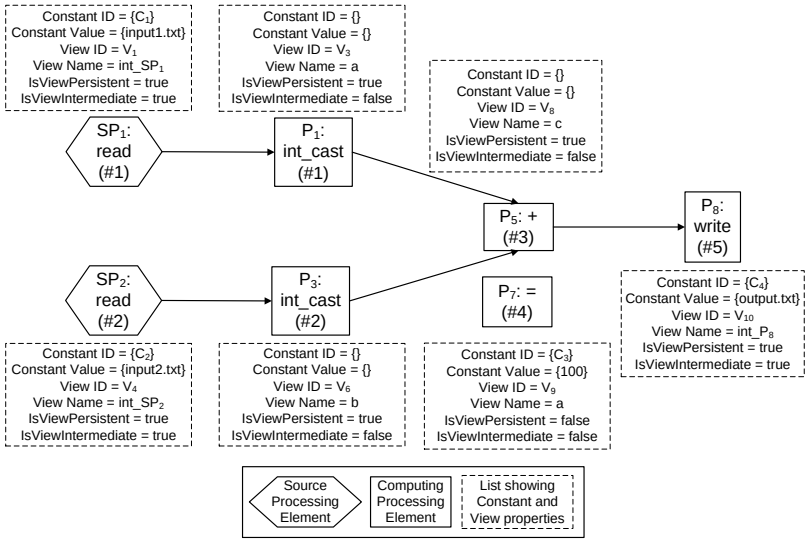
Figure 3.19 shows all four graph compression re-write rules. Rule *GC 1* unifies a constant node with the following source processing element and deletes the constant node. If a match is found, the rule *GC 1* copies all properties of the constant node C_1 to the corresponding properties of the source processing element node SP_1 and deletes the constant node C_1 eventually. Since several constant nodes might be connected with the same source processing element, the source processing element maintains an array or a list for keeping the values of all constant node properties. Rule *GC 2* unifies a constant node with the following computing processing element node. The computing processing element also maintains an array of values of the constant nodes properties.

Unifying output view

The other two rules, *GC 3* and *GC 4*, unify a view node with the preceding computing processing element and source processing element node, respectively and discard the view node. Rule *GC 3* and *GC 4* also ensure



(a) Before applying graph compression re-write rules



(b) The workflow provenance graph (after applying graph compression re-write rules)

Figure 3.20: Transformation to the Workflow provenance graph

that the outgoing edges from the view node, i.e., $e_1, ..., e_n$, are now connecting from the respective processing elements. Figure 3.19 also shows the copied properties of constants or a view node into the corresponding source/computing processing element node. Please note that, in Figure 3.19, the list of copied properties only contains mandatory properties of a constant or a view node. In practice, all properties including optional ones are copied while applying graph compression re-write rules.

Figure 3.20 shows an example of applying these graph compression re-write rules over a given provenance graph. The initial workflow provenance graph shown in Figure 3.2 has been considered to apply the graph maintenance re-write rules discussed in Section 3.7. After applying these rules, the outcome is shown in Figure 3.18c. This provenance graph is considered for applying the graph compression re-write rules to infer the final *workflow provenance graph*. *Example*

Figure 3.20a shows the input provenance graph over which the graph compression re-write rules are applied. According to these re-write rules, all constants and views will be unified with the corresponding source and computing processing element nodes and the properties of those constant and view nodes are copied into the processing element nodes to ensure no loss of information. In Figure 3.20a, there are 7 processing elements. Therefore, applying the model modification re-write rules will return the final workflow provenance graph with 7 nodes. The final workflow provenance graph is shown in Figure 3.20b alongside the properties of deleted constant and view nodes which are now copied into the corresponding source and computing processing element nodes.

After applying all these re-write rules, we have the workflow provenance graph. Figure 3.20b shows the workflow provenance graph after being transformed from the initial workflow provenance graph shown in Figure 3.2. The workflow provenance graph is significantly more compact than the initial one and it also transforms all control dependencies into data dependencies.

3.9 EVALUATION

The goal of the workflow provenance inference is to capture workflow provenance information automatically for a scientific model that has been developed in a provenance-unaware platform such as a scripting environment, general purpose programming languages like Python, Java etc. In this chapter, the workflow provenance inference method is demonstrated over Python programs. One of the design factors of the workflow provenance inference method is that the method will be *generic* in nature as discussed in Section 1.4. The *generic* nature of the workflow provenance inference method refers to the capability of handling Python programs with different types of programming constructs. Therefore, the main eval-

uation parameters are: i) *applicability* and ii) *accuracy* of the workflow provenance inference method. Furthermore, we would like to keep the workflow provenance graph as compact as possible to increase the readability of the graph. Therefore, we also measure the *compactness ratio* of the workflow provenance graph and we consider *compactness ratio* as the third evaluation parameter.

Evaluation parameters

Applicability refers to the percentage of Python programs which can be handled by the workflow provenance inference method and as a result, a *workflow provenance graph* can be generated. *Accuracy* refers to the percentage of programs for which the generated workflow provenance graph is accurate. *Accuracy* of a workflow provenance graph is determined by experts in Python programming. Experts compare a Python program and the corresponding workflow provenance graph line by line. A workflow provenance graph is considered to be an accurate one, if the captured provenance graph can show the exact relationship between variables (data products) and operations (activities) for each line of the program. Otherwise, the workflow provenance graph for that program is considered as an inaccurate one. Since the *accuracy* is determined by means of manual checking, the *accuracy* reported in this section is approximate. The last evaluation parameter, *compactness ratio* refers to the ratio of the total number of nodes between the initial workflow provenance graph and the final workflow provenance graph. This parameter can be a rough indicator of the performance of the rewrite-rules in terms of reducing the graph size.

3.9.1 Test cases

Programs source

We collect Python programs from three different sources. The first set of programs are used in the Data2Semantics[9] project. In this project, researchers are developing approaches that enable scientists to more easily publish, share and reuse data. The set of Python programs that we present as test cases for this evaluation, are used for converting raw data such as comma-separated values (csv), tab-separated values (tsv) and other custom formats into RDF[10] triples and for managing a triple store as well. We consider 14 Python programs from this toolset which are also available online[11].

9 Available at `http://krr.cs.vu.nl/author/data2semantics/`

10 Available at `http://www.w3.org/TR/rdf-concepts/`

11 Available at `https://github.com/Data2Semantics/raw2ld/tree/master/src`

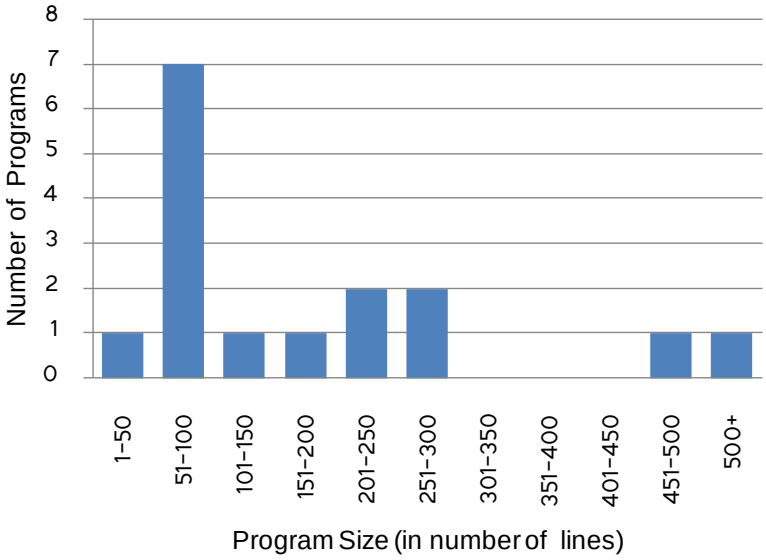


Figure 3.21: Distribution of programs based on their size (in number of lines)

We also collect two other Python programs used from two other research groups. One of the programs estimate the total water demand on the global level used by the hydrologists in the Utrecht University, The Netherlands. The other program is collected from one of our colleagues in the University of Zurich, Switzerland. This program is used for data manipulation such as barometric pressure and air temperature and the program generates a file with metadata.

In total, we present 16 Python programs as test cases for the evaluation. This collection of programs is diverse in nature as they are used in different domains. The size of the programs in lines of code also varies from 34 to 991, with average size of around 200 lines of code. Figure 3.21 shows the distribution of programs size in terms of number of lines.

Programs size

All these programs contain most primitive assignment and arithmetic operations as well as built-in/library function calls. A majority of the programs contain conditional branching and looping constructs. Table 3.1 shows the break down for different types of statements found in the collection of programs used in this evaluation.

Programs type

Table 3.1: Different types of statements found in the collection of programs

TYPE OF STATEMENTS	PERCENTAGE OF TOTAL PROGRAMS
Conditional branching	88%
Looping constructs	88%
User-defined function call	44%
Object Instantiation	19%
`try-except-finally` block	13%
`with` statements	7%

3.9.2 Applicability and Accuracy

Applicability

We use all 16 Python programs in the evaluation to measure the *applicability* and the *accuracy*. Among these 16 programs, the workflow provenance inference method is applicable to all programs and the method infers the corresponding workflow provenance graphs. Therefore, the *applicability* of the workflow provenance inference is 100%. This high applicability of the inference mechanism is achieved because of its capability to address most of the Python programming constructs. However, there are a few valid Python statements for which the inference mechanism cannot generate the provenance graph due to the limitation of the facilitated Python grammar in the implementation. As an example, the statement `a = b = c` is a valid Python statement given that a value is already assigned to `c`. The workflow provenance inference method cannot handle this statement. However, decomposing this statement into the following statements can overcome this problem: `b = c` and `a = b`.

Accuracy

Next, we determine the *accuracy* of the workflow provenance inference method. To calculate the *accuracy*, inferred workflow provenance graphs have to be checked manually by experts in Python programming. An accurate workflow provenance graph captures the exact relationships between variables (data products) and operations (activities) for each line in the corresponding program. However, it becomes a tedious and a time consuming task to check the generated provenance graphs manually especially for programs with more than 2 levels of nested compound statements (e.g.

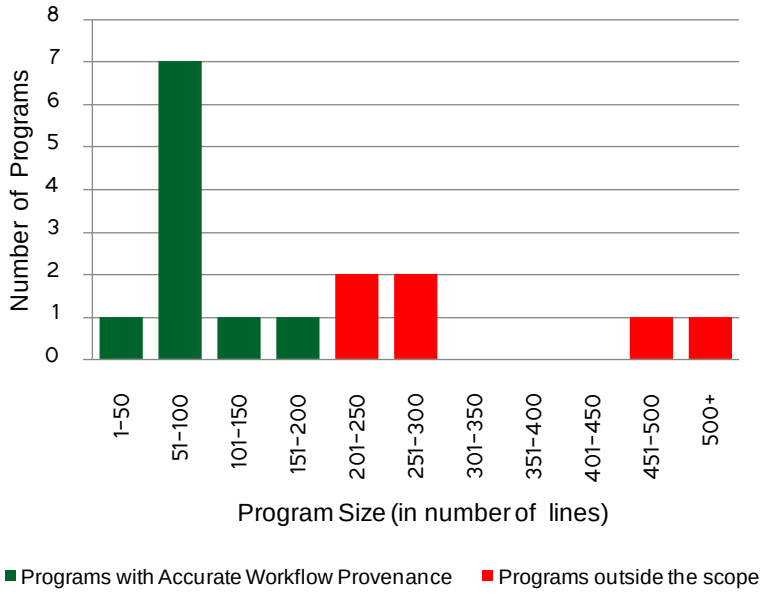


Figure 3.22: Distribution of programs (with accurate provenance graphs/outside the scope) based on their size (in number of lines)

conditional branching) exhibiting control-flow based coordination. Therefore, we narrow down the scope of our experiments. We only consider the programs which have at most 2 levels of nested compound statements to measure the accuracy. We have found that there are 10 programs satisfying this criterion. Then, the workflow provenance graphs of those 10 programs are generated and are manually checked by experts in Python programming. We find all 10 provenance graphs are accurate.

Figure 3.22 shows the distribution of programs with accurate workflow provenance or programs falling outside the defined scope based on their size in number of lines. From Figure 3.22, we observe that all 10 programs considered during the accuracy calculation process have less than 200 lines of code. The other 6 programs outside the scope of the experiment, have more than 200 lines of code.

3.9.3 Compactness Ratio

Compactness ratio is a rough indicator of the performance of the re-write rules used in the workflow provenance inference method in terms of reduc-

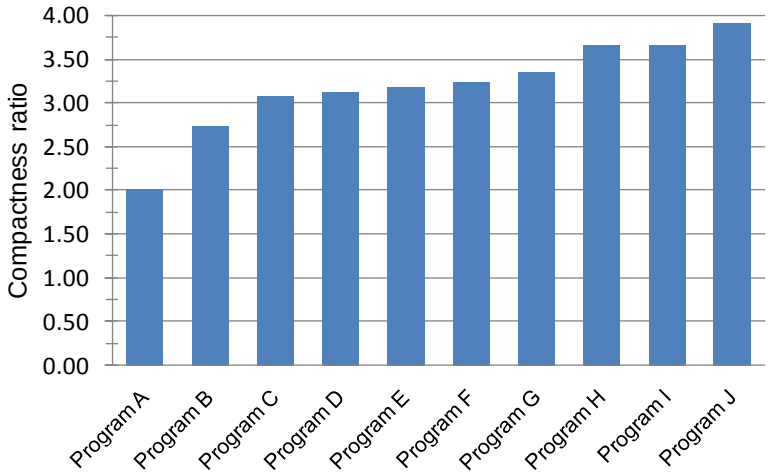


Figure 3.23: Compactness ratio of accurate workflow provenance graphs for corresponding programs in ascending order

ing the graph size. The size of a workflow provenance graph is calculated by only considering the total number of nodes in that graph. It is quite common that the size of a provenance graph can be enormous especially if the graph is created by analyzing a program due to the syntactic details of the programming language. In such a case, it becomes difficult to interpret the provenance graph for a scientist which could limit the application of the provenance graph. Therefore, one of the key design factors during the inference of workflow provenance graph is that the final workflow provenance graph should be compact in size.

Compactness ratio is measured by calculating the ratio between the total number of nodes in the initial workflow provenance graph and the workflow provenance graph. To measure this parameter, we consider the 10 accurate workflow provenance graphs and their corresponding initial workflow provenance graphs.

Figure 3.23 shows the *compactness ratio* of 10 workflow provenance graphs in ascending order. The minimum compactness ratio is 2.02 : 1 which means that the size of an initial workflow provenance graph is at least 2 times bigger than the size of a workflow provenance graph. The maximum compactness ratio achieved is 3.92 : 1. The average compactness ratio measured over these 10 workflow provenance graphs and initial workflow provenance graphs is 3.19 : 1. In general, the graph re-write rules can reduce the initial workflow provenance graph size by more than 66%

Table 3.2: Summary of the compactness ratio of the workflow provenance graphs

Minimum compactness ratio	2.02:1
Maximum compactness ratio	3.92:1
Average compactness ratio	3.19:1

when transformed into the final workflow provenance graph. The maximum compactness is achieved for the program which has only a few compound statements such as conditional branching and looping. On the other hand, the minimum compactness is achieved for the program which has higher number of statements exhibiting control-flow based operations. Therefore, from this observation, we can conclude that if the program has comparatively fewer explicit control dependencies, the compactness of the final workflow provenance graph will be higher than the average. Table 3.2 summarizes the results.

3.10 DISCUSSION

The proposed workflow provenance inference method can capture workflow provenance based on a given Python program automatically. We argue that the general principle of this method can be extended to address other scripting and programming languages such as MATLAB[12], R[13], Answer Set Programming [52] etc. Furthermore, we think that the workflow provenance inference method can infer provenance information with the limited amount of knowledge available on the programs which are used to realize a scientific model. However, there exist a few limitations of the workflow provenance inference method which is discussed in this section.

3.10.1 Platforms having Coordination between Users/Components

The proposed workflow provenance inference method is demonstrated over Python programs. However, the core idea of the proposed approach

12 Available at `http://www.mathworks.nl/products/matlab/`

13 Available at `http://www.r-project.org/`

is independent of the used programming language. It is to automatically translate a control-flow coordinated program into a data-flow coordinated program. The mechanisms to perform this transformation as proposed in this chapter are limited to certain control-flow based coordination mechanisms like conditional branching, looping, modularization etc. However, we do not address coordination mechanisms including interaction between multiple users, components using messages instead of function calls, which are often done in Business Process Execution Language (BPEL).

3.10.2 Interpretation of Provenance Graphs

The workflow provenance inference method infers workflow provenance by static analysis of the program. One of our main purposes to develop this inference method is to put minimum burden to the users to achieve provenance information. However, acquired workflow provenance does not explicate the semantics of activities and data products. The user has to interpret and understand the meaning of the processing steps and the used data products. If semantic information, like e.g. metadata of the sources and their data structure, is available then this may help the user interpreting the provenance graph, but it is not part of the provenance graph as addressed in this chapter.

The approach is most beneficial to a user when there is little of the available data actually used for the calculation of an individual result, but these data originate from many different sources. The less sources are involved and the more data contributes to an individual result, the bigger the provenance graph gets and therefore the harder it is for the user to interpret the provenance graph and get useful information out of it.

3.10.3 Nested Iterative Activities

The proposed workflow provenance inference method can transform control dependencies, occurring due to a conditional branching or an iterative operation, into data dependencies. Currently, the method handles one looping operation/activity at a time and infer the window size and trigger interval of this activity based on the input parameters given to that looping activity. However, we do not consider the plausible dependencies between multiple nested looping. Due to this nested iterative structure, sometimes

the properties (e.g. window size, trigger etc.) of an inner loop can influence the properties of an outer loop. Identifying these influences requires in-depth analysis of involved control structures and possibly manual annotations. The workflow provenance inference discussed in this chapter does not address this problem. This is a potential future work to improve the workflow provenance inference method.

3.10.4 Recursive Functions

The workflow provenance inference method does not support any recursive function. The biggest challenge of handling recursive functions is to figure out the exact data-flow based coordination between associated activities. Since same variables with different values are used at different stages of a recursive function, it is difficult to transform control-flow based coordination into data-flow based coordination. Furthermore, it is also difficult to identify the end of execution of a recursive function without interpreting the values of participating data products. The ability to handle recursive functions is another future work that can enhance the proposed workflow provenance inference method.

3.11 SUMMARY

The workflow provenance inference method, presented in this chapter, can capture workflow provenance automatically based on a given program. This method becomes useful for collecting provenance traces for scientific models which are built in a provenance-unaware platforms like Python programming language. The first research question, *RQ 1*, introduced this challenge of extracting workflow provenance automatically in a provenance-unaware platform. Therefore, the workflow provenance inference method answers the first research question, *RQ 1*.

At the beginning of this chapter, we introduced the workflow provenance model which is one of the core concepts in the proposed workflow provenance inference method. Based on this model, we represented workflow provenance information as a bipartite graph, consisting of two types of nodes - data products and activities, and edges representing data dependencies between these nodes. Later, we explained the data-flow based

semantics between nodes in a workflow provenance graph followed by the discussion on the provenance representation scheme used.

Next, we explained the workflow provenance inference method. First, the method parses a given Python program to get an abstract syntax tree (AST) of that program. Afterward, objects of the appropriate class based on the object model of the Python are created by traversing through the AST. Having all the objects, the initial workflow provenance graph is created. The initial workflow provenance graph might exhibit control dependencies. Therefore, a set of functions are applied over the initial workflow provenance graph to achieve a more compact, data dependent provenance graph, called workflow provenance graph. First, we apply transformation function, consisting of some graph re-write rules. These re-write rules transform all control dependencies into data dependencies. Next, maintenance and compression functions, consisting of other re-write rules are applied over the graph to achieve the final workflow provenance graph. The workflow provenance graph we inferred by applying these re-write rules, can explicate data dependencies between activities and data products found in the given program.

Our evaluation shows that the proposed workflow provenance inference method is applicable to a wide variety of Python programs which ensure its generic nature. The proposed method captures accurate workflow provenance graphs for all test cases considered. The generated workflow provenance graphs are more compact than the corresponding initial workflow provenance graphs. In average, the method can reduce the size of an initial workflow provenance graph by 66% of its original size.

The general principles of the workflow provenance inference method can be extended and adapted to build a similar tool for other scripting and general-purpose programming languages. Furthermore, the workflow provenance inference method infers workflow provenance information with minimal prior information. This method can save a lot of time and effort of scientists who currently manage provenance of their experiments using other provenance-aware systems. Currently, the method does not handle recursive functions and does not address plausible dependencies in a nested iterative structure. We would like to address these operations in future to improve the proposed method.

4

BASIC PROVENANCE INFERENCE

The workflow provenance of a scientific model can either be created and stored by using provenance-aware workflow engines, stream processing engines, complex event processing engines [84, 116, 102, 24, 8, 2, 3] or be captured automatically as discussed in Chapter 3 in case the model is developed using a general purpose programming tool such as Python. The workflow provenance of a scientific model provides the relationship among different operations at the design phase as discussed in Section 1.3.1. However, it cannot provide the provenance information during the execution of that scientific model.

Therefore, we need a mechanism that can achieve fine-grained data provenance, i.e., the relationship between data products generated during the execution of the model. Fine-grained data provenance can be used as a means of debugging the model to validate it as well as it allows to have reproducible results. Therefore, efficient management of fine-grained data provenance is of utmost importance to the scientific community especially to the scientists handling massive, continuous data streams.

Fine-grained data provenance can be explicitly documented and stored in a database during the execution of a scientific model. This is a feasible approach for a small amount of manually sampled data because the storage space consumed by provenance information remains constant and does not accumulate over time which is the case in stream data processing.

This chapter is based on the following work: Inferring Fine-Grained Data Provenance in Stream Data Processing: Reduced Storage Cost, High Accuracy. In *Database and Expert Systems Applications (DEXA'11)*, volume 6861 of *LNCS*, pages 118–127. Springer, 2011. & Facilitating fine grained data provenance using temporal data model. In *Proceedings of the Workshop on Data Management for Sensor Networks (DMSN'10)*, pages 8–13. ACM, 2010.

Challenges

To process data streams, a processing element is continuously executed on a subset of the data stream, also known as a window. Executing a processing element defined over a window, requires to document fine-grained data provenance at each window execution to have a complete provenance trace. The provenance trace allows scientists to debug as well as to reproduce results. If the window is large and subsequent windows have significant overlaps with each other, a particular input data product contributes to several output data products. These relationships between the input data product and multiple output data products need to be maintained in form of fine-grained data provenance. Therefore, the size of provenance data becomes a multiple of the size of the actual data stream. Since provenance data is another type of metadata and not frequently accessed by users, the explicit documentation of fine-grained data provenance seems to be too expensive and infeasible [64, 69].

Furthermore, documenting fine-grained data provenance explicitly requires *operator instrumentation* that explicitly adds a few lines of code into the actual model itself to capture provenance [45]. This technique has been used in the ES3 project providing computational provenance [44]. However, a drawback of operator instrumentation is the need to extend all operators which exist in a model and thus, is a time consuming task.

Solution

Therefore, managing fine-grained data provenance in a *cost-efficient* manner in terms of storage and time is one of the key questions which need to be satisfied to develop a framework managing data provenance. It is also addressed as one of the research questions (*RQ 2*) in this thesis, mentioned in Section 1.4. To satisfy the research question *RQ 2*, we propose several methods to infer fine-grained data provenance to ensure cost efficiency. These inference-based methods can infer fine-grained data provenance based on the workflow provenance of the scientific model and the timestamps attached to the data products. Each of these methods has it's own pros and cons and is suitable for a particular set of system dynamics which includes input data products arrival pattern, processing delay etc. The suite of these inference-based methods to infer fine-grained data provenance makes the proposed provenance management framework more generic. A detailed guideline on the suitability of these inference-based methods is discussed in Chapter 7.

Basic provenance inference

In this chapter, we discuss the *basic provenance inference* method. The *basic provenance inference method* is applicable to both data streams and offline (non-stream) data products . It can handle input data streams with regular

and constant arrival pattern. The basic provenance inference method also assumes that the processing time required to execute a processing element, called processing delay, is constant. In cases with different system dynamics than the aforesaid one, the basic provenance inference method might provide inaccurate provenance information.

Chapter structure

This chapter is structured in the following way. First, we present a scenario based on a real project followed by the description of the example workflow associated with the scenario. Next, we describe a few basic concepts used to explain the basic provenance inference method. The overview of the basic provenance inference method is presented afterward. Then, we discuss the required information to be needed by the underlying system to execute this inference-based mechanism followed by the explanation of the working principle of the basic provenance inference method. Eventually, we evaluate this method based on the example workflow presented before followed by the discussion on the applicability of this inference-based method in different situations.

4.1 SCENARIO DESCRIPTION

RECORD[1] is one of the projects in the context of the Swiss Experiment[2], which is a platform to enable real-time environmental experiments. One objective of the RECORD project is to study how river restoration affects water quality, both in the river itself and in the groundwater [110].

Example

Figure 4.1 shows the overview of the scenario. As depicted in Figure 4.1, different types of input data products are acquired by several sensors which have been deployed to monitor river restoration effects. The type of input data products may vary from high resolution infrared image to a simple data tuple depending on the type of sensors. In this scenario, there are a few sensors measuring electrical conductivity of water which is a measure of the number of ions in the water. Increasing conductivity indicates the higher level of salt in water. Since scientists are interested to control the operation of a drinking water well by facilitating the available sensor data, this data stream of electrical conductivity is sent for the processing. The data processing mechanism provides output data products which is a

1 Available at `http://www.swiss-experiment.ch/index.php/Record:Home`
2 Available at `http://www.swiss-experiment.ch/`

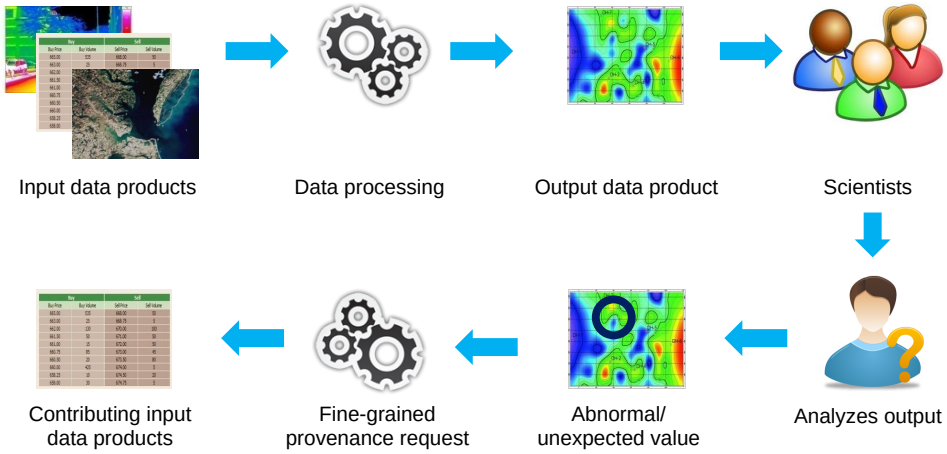


Figure 4.1: Scenario overview

contour map representing electrical conductivity in a known region of the river as shown in Figure 4.1.

As depicted in Figure 4.1, scientists could analyze the output data, i.e., a contour map, and in case of any abnormal or unexpected value exists in the map, they could request the derivation history of that particular output data, also known as fine-grained data provenance. The provenance request results into the set of input data products that contribute to produce the selected output data. The provenance information can be used as a means to debug the outcome of a scientific model as well as to validate the model.

4.2 WORKFLOW DESCRIPTION

To explain and evaluate the basic provenance inference method, we construct an artificial and simplified workflow in the light of the scenario discussed in Section 4.1. We assume that a region of the river is divided into 3×3 grid cells. Further, we also assume that there are three sensors measuring electrical conductivity in three different cells. Sensors send data tuples containing the `device id`, the `latitude` and the `longitude` of the device, the measured `electrical conductivity`, the `timestamp` of the measurement, also referred to as *valid time* [79], along with some other attributes. Scientists use these conductivity readings to compute an approximate electrical conductivity for all cells in the grid using a spatial interpolation operation. Later, based on the scenario, a contour map of electrical conductivity

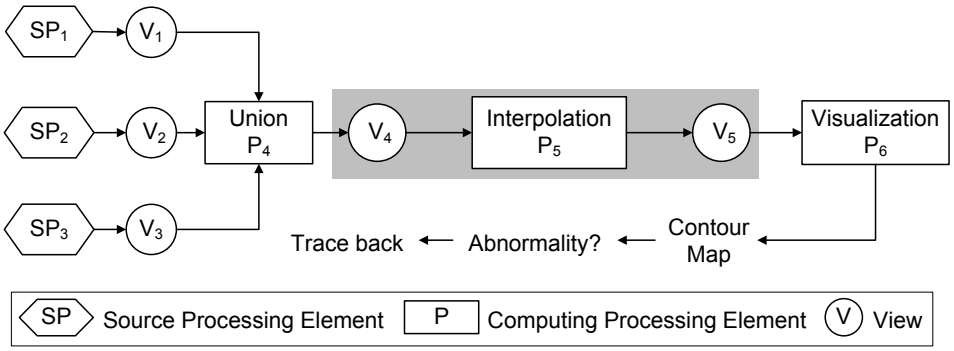


Figure 4.2: The example workflow

for that region is produced. Scientists can request the provenance of any of the points within the map if the value hold by that point seems abnormal.

As introduced in Chapter 3, a scientific workflow can be represented as a graph, known as workflow provenance graph. A workflow provenance graph has different types of nodes - i) constants, ii) views, iii) source processing elements and iv) computing processing elements as mentioned in Section 3.1. The definitions and notations of these nodes are used to represent the aforesaid workflow which is depicted in Figure 4.2.

Example

In Figure 4.2, for each sensor, there is a corresponding source processing element named SP_1, SP_2 and SP_3 which reads these data tuples and stores the data tuples in view V_1, V_2 and V_3, respectively. Later, these views act as inputs to the computing processing element P_4 which executes an *union* operation, i.e., combining all available data tuples, and produces a view V_4 as output. This view acts as an input to the computing processing element P_5. P_5 computes the interpolated values for all cells of the grid using the values sent by the three sensors and stores the interpolated values in the view V_5. All views holding data tuples in Figure 4.2 are persistent views (*IsPersistent=true*), i.e., data tuples are never deleted. Later, the view V_5 is used by the *Visualization* processing element to produce a contour map of the electrical conductivity. The shaded part of the workflow in Figure 4.2 is considered to evaluate the basic provenance inference method. Furthermore, for comprehensive evaluation, we also replace P_5 with other relevant computing processing elements executing different operations and report the results in Section 4.7.

4.3 BASIC TERMINOLOGY

In the previous section, we described a simple and artificial, scientific workflow as shown in Figure 4.2. The scientific workflow exhibits data-flow based coordination, i.e., a particular processing element will be executed based on the availability of input data products, similar to the semantics of the workflow provenance model, discussed in Chapter 3. Executing this scientific workflow produces output data products and the proposed basic provenance inference method can infer fine-grained data provenance of a selected output. The basic provenance inference method can reduce storage overhead to maintain provenance data especially in cases when a single input data product contributes several times, producing multiple output data products. The aforesaid situation often occurs while processing data streams. Therefore, the understanding of the concepts associated with stream data processing such as windows, triggers, sampling interval, processing delay is important. In this section, we restate the definitions of different types of nodes which exist in the example workflow shown in Figure 4.2 along with the related terminology like windows, triggers etc. Furthermore, we introduce the notations used for this terminology which are used to explain the working principle of the basic provenance inference method.

Views

As defined in Section 3.1, a *view* represents either any variable defined in the scientific model or a result generated by a processing element. From Database point of view, these variables can hold a set of tuples (e.g. a list). Furthermore, in the context of data streams, new data products/tuples can be added into the list referenced by a variable. Therefore, a view V_i can be defined as a set of tuples ${t_j}^i$ where j is the *transaction time* [79]. *Transaction time*, j refers to the system timestamp indicating the point in time when the tuple is inserted into the view V_i. Each tuple ${t_j}^i : \{v\}$ consists of a set of values of attributes, v, conforming to a schema S_i defined for a view V_i.

Depending on whether the tuples hold by the views are made persistent or not, views can be classified into two types: i) persistent and ii) non-persistent. Once inserted, data tuples are never deleted from a persistent view. As mentioned in Section 3.1, these views have a property *IsPersistent* which is set to *true*. On the contrary, in a non-persistent view, it is possible that tuples inserted into the view are not committed and therefore, are no longer available to access after they are processed. For non-persistent views

the *IsPersistent* is set to *false*. All views shown in Figure 4.2 are persistent views.

Processing elements

Views are produced by both source and computing processing elements. As defined in Section 3.1, a *source processing element* represents an operation that either assigns a constant value into a variable or an operation that acquires data from the disk or any other source. In the example workflow shown in Figure 4.2, SP_1, SP_2 and SP_3 are source processing elements since they acquire data products sent by the sensors. On the contrary, a *computing processing element* represents an operation that either computes a value/data product or writes data products into a file, database etc. In Figure 4.2, P_4, P_5 are computing processing elements. Views which are produced by the source processing elements are always persistent which is not necessarily true in case of views produced by computing processing elements.

Sampling interval

Tuples can be inserted into a view V_i either at a regular interval or in an arbitrary manner. The amount of time between two successive tuples insertion into a view V_i is referred to as *sampling interval*, λ_i. λ_i can vary depending on the nature of the source or computing processing elements which generates and inserts the tuples into the views. However, in this example workflow shown in Figure 4.2, we assume that for all views V_i, λ_i is constant, i.e., tuples are inserted at a regular interval.

Windows

A view can be used as an input to computing processing elements only. A computing processing element, P_k, requires a window to be defined over the input view for it's successful execution in the context of data streams. As defined in Section 3.1, a *window* specifies a subset of data products used by a computing processing element to produce output data products. Therefore, a window $(W_i{}^n)_k$ is a subset of tuples within a view V_i at the n^{th} execution of a computing processing element P_k. In case of multiple input views, the computing processing element P_k is executed over a set of windows $\mathbb{W} = \{(W_x{}^n)_k, (W_y{}^n)_k, ...\}$ which are a finite subset of a set of input views $\mathbb{V} = \{V_x, V_y, ...\}$.

Tuple-based window

A window $(W_i{}^n)_k$ can be either *tuple-based* or *time-based*. In tuple-based windows, the number of tuples within a window remains constant. A tuple-based window can be defined based on two parameters: i) window size m and ii) a point in time T. A tuple-based window $(W_i{}^n)_k$ is a finite subset of V_i containing the latest m number of tuples $t_j{}^i$ where $j \leqslant T$. The *window size* is represented as $WS_i{}^k$ where, $WS_i{}^k = m$ (number of tuples).

Time-based window

In a time-based window, tuples whose *transaction time* (system timestamp) falls into a specific boundary constitutes a window. A time-based window $(W_i{}^n)_k = [start, end)$ is a finite subset of V_i containing all tuples $t_j{}^i$ where $start \leqslant j < end$. In cases of time-based windows, the window size of a window is $WS_i{}^k = end - start$ (amount of time units).

Trigger interval

Moreover, the execution of a computing processing element, P_k, also depends on a trigger. As defined in Section 3.1, a *trigger interval*, TR_k, refers to the predefined interval between two successive executions of a computing processing element, P_k. The trigger interval of a computing processing element could be either tuple-based or time-based. Therefore, the trigger interval specifies either the number of newly arrived tuples required to execute P_k (tuple-based trigger) or the amount of time units between two successive executions of P_k (time-based trigger).

Processing delay

After a processing element P_k is triggered, it takes an amount of time to finish the processing. The amount of time to complete the execution of a processing element, P_k, is referred to as *processing delay* δ_k. The value of δ_k may vary at each execution of P_k depending on the nature of P_k. However, in the example workflow shown in Figure 4.2, we assume that for all computing processing elements P_k, δ_k remains constant.

The aforesaid terms are used to explain the working principle of the basic provenance inference method presented in this chapter.

4.4 OVERVIEW OF THE BASIC PROVENANCE INFERENCE

Basic provenance inference is one of the inference-based methods to infer fine-grained data provenance at reduced costs in terms of storage consumption. It is suitable for the scientific models having offline (non-stream) data as well as for the models processing data streams where the input data products arrive at a regular interval and the processing delay always remains constant. The basic provenance inference method infers fine-grained data provenance in three phases: i) Documentation of workflow provenance, ii) Backward computation and iii) Forward computation.

Documentation of workflow provenance

The *documentation of workflow provenance* phase is a one-time action, performed during the setup of the processing elements of a scientific model. In this phase, we document the workflow provenance of a scientific model, i.e., different values of properties of the processing elements and their relationship with each other including the views, as discussed in Section 3.1.

There might be some extra properties of a processing element which are not specified in Section 3.1, but are required to infer accurate fine-grained data provenance. We discuss these properties in Section 4.6.1 after motivating their necessity.

The next two phases are the main phases that infer fine-grained data provenance based on the documented workflow provenance and the timestamps (transaction time) of the data products. These phases will be executed only when the scientist is interested to know the fine-grained provenance information of an output data product, i.e., a tuple in the output view. The scientist will select an output data product which seems to have abnormal or unexpected value. After choosing the output data product, the *backward computation* phase is executed. During the backward computation phase, it takes the given workflow provenance and the timestamp of the chosen data product into consideration. By facilitating the workflow provenance information, i.e., window size over the input views, processing delay etc., it reconstructs the original processing window over the input views, which is referred to as the *inferred window*. The input data products within the inferred window might contribute to produce the selected output data product.

Backward computation

Afterward, the final phase is executed. In the *forward computation* phase, the method establishes the relationship between the input data products within the inferred window and the output data product based on the given workflow provenance of the scientific model, i.e., *input-output ratio* of the corresponding computing processing element. It refers to the ratio of the number of input data products contributed to the output data products over the number of output data products produced during the execution of a computing processing element as defined in Section 3.1.

Forward computation

The other two inference-based methods to infer fine-grained provenance presented in this thesis in Chapter 5 and Chapter 6 also have the aforesaid three phases. While the mechanism of documenting workflow provenance is exactly the same in all three inference-based methods, the mechanism of both backward and forward computation phases differ from one provenance inference method to the other to address different system dynamics such as regular vs. irregular input data arrival pattern, constant vs. variable processing delay, single-step vs. multiple-step workflows etc. Since all three inference-based methods depend on the workflow provenance of the given scientific model and the timestamps of the data products, there

are some common pieces of information required to apply these inference-based methods. We discuss this required information in the next section.

4.5 REQUIRED INFORMATION

To infer fine-grained data provenance using the inference-based methods, the following information is required.

- *Explicit System Timestamps* - System timestamp, also referred to as *transaction time* [79], is added to every data product representing the point in time when the data product/tuple is inserted into the view.
- *Temporal Ordering* - The inference-based methods require the processing elements to process data products/tuples based on their order on *transaction time* in the input view. If a tuple's *transaction time* is t, this tuple will be processed after the tuples with *transaction time* $<$ t and before the tuples with *transaction time* $>$ t.
- *Workflow Provenance* - Workflow provenance of a scientific model is required which documents the relationship between input views, processing elements and output views.

In Chapter 3, we define workflow provenance and the way of representing a workflow provenance of a scientific model. Workflow provenance of a scientific model can be represented as a graph, known as workflow provenance graph as discussed in Section 3.1. There are different types of nodes with their associated properties in a workflow provenance graph which are facilitated to apply the basic provenance inference method. There are a few extra properties of a computing processing element required to be documented which were not specified in Section 3.1. These properties of a computing processing element are discussed in the next section which are also used to classify the computing processing elements as shown in Table 4.1.

Table 4.1: Classification of the Computing Processing Elements implementing different operations

Operation	*Input-output ratio*	*No. of Contributing Input Tuples*	*No. of Produced Output Tuples*	*No. of Input Views*	*Contributing Views*
Project	constant ('one to one')	single	single	single	n/a
Cartesian product	constant ('one to many')	single	multiple (=window size)	multiple	all
Average	constant ('many to one')	multiple (=window size)	single	single	n/a
Interpolation	constant ('many to many')	multiple (=window size)	multiple	single	n/a
Union	constant ('one to one')	single	single	multiple	specific
Select	variable	single	variable	single	n/a

4.5.1 Classification of Computing Processing Elements

A scientific model is comprised of a variety of activities and operations. These activities/operations are transformed into a corresponding computing processing element during the execution of the model as discussed in Section 1.3.2. The computing processing elements execute operations including different types of SQL operations such as *select*, *project*, *aggregate functions*, *cartesian product*, *union*, generic functors like *interpolation*, *extrapolation* etc. A computing processing element takes a number of input data products/tuples and maps them to a set of output data products/tuples after the successful execution. The ratio between the number of input data products contributed to produce output data products over the number of output data products of a particular computing processing element is referred to as *input-output ratio* as defined in Section 3.1. Depending on this property, we can classify the computing processing elements into two major categories: constant and variable ratio computing processing elements as discussed in Section 1.3.1.

Input-output ratio

Constant ratio computing processing elements have a constant *input-output ratio* each time they execute. As for example, processing elements implementing *project*, *aggregate functions*, *interpolation*, *cartesian product* and *union* operations are constant ratio computing processing elements. *Variable ratio* processing elements do not have a constant *input-output ratio* at the time of each execution. A computing processing element implementing *select* operation is an example of variable ratio computing processing element. In this case, we observe a variation in input-output ratio due to the conditional clause of *select* which has to be satisfied by the input data products to be processed. The column *Input-output ratio* in Table 4.1 indicates the constant and variable ratio computing processing elements.

Number of input views

A computing processing element might have single or multiple input views. As for instance, computing processing elements executing *project*, *average*, *interpolation*, *select* operations have single input view as indicated in Table 4.1. On the other hand, a computing processing element performing *cartesian product* or *union* operation have multiple input views as shown in Table 4.1. The *number of input views* of a computing processing element is required to document in the workflow provenance to generate the *inferred window* over each participating input view during the backward computation phase.

In cases where a computing processing element has multiple input views, further classification is required based on the number of *contributing input views* as indicated by the last column in Table 4.1. It distinguishes whether an output tuple is produced by the contribution of the input tuples from a specific input view or all input views. As an example, computing processing elements implementing both *union* and *cartesian product* operations have multiple input views. While in *union* operation, one input tuple from a specific input view contributes to produce one output tuple, in a *cartesian product* operation, tuples from all input views contribute to generate an output tuple. This information should be also documented in the workflow provenance to establish the exact relationship between input and output data products during the forward computation phase.

Contributing input views

As already mentioned, we need to document the information on the participating input views of a computing processing element, i.e., number of input views, contributing input views as a part of the workflow provenance. Furthermore, there are a few assumptions which have to be satisfied by a subset of computing processing elements to infer accurate fine-grained data provenance. This subset includes computing processing elements which have either multiple input views or produce multiple output data products per window execution. The assumptions are listed in the next section.

4.5.2 Assumptions on Computing Processing Elements

The computing processing elements which have either multiple input views or produce multiple output data products per window execution (e.g. *cartesian product*, *union*, *project*, *interpolation* etc.) need to satisfy the following assumptions to infer accurate fine-grained data provenance.

- *Order of Input Views*: For a computing processing element with multiple input views where all input views contribute to produce a single output data product, the order in which the computing processing element iterates over the input views must be known to infer fine-grained data provenance. As an example, the participation of input views at the inner and outer loop used in a *cartesian product* operation must be known to the inference-based methods beforehand.
- *Contributing Input View*: For a computing processing element with multiple input views where a specific single input view contributes

at a time, it is required to document the name of the input view explicitly with the output data product. As an example, in case of a computing processing element implementing an *union* operation, for each tuple in the output view, the name of the input view from where that particular output tuple has been produced, must be documented in an additional column in the output view.

- *Order of Tuples in the Output View*: For a computing processing element with multiple output data products per window execution (e.g. *project* operation), it is required to ensure the following:
  1. All output data products/tuples produced by executing the same window have the same *transaction time* while inserting into the output view.
  2. The order of contributing input data products/tuples in the window must be preserved in the set of output data products/tuples produced from that window. Let, $(W_i{}^n)_k$ be the current window defined over the view V_i which is an input view to the processing element P_k as defined in Section 4.3. I be the set of input tuples within the window $(W_i{}^n)_k$ and O be the set of output tuples produced by P_k. Assuming that, $i_1, i_2 \in I$ and $o_1, o_2 \in O$, i_1 contributes to produce o_1 and i_2 contributes to produce o_2, *transaction time* of i_1 < *transaction time* of i_2 then, this requirement ensures that o_1 will appear before o_2 in O.

These assumptions are required to be satisfied to infer accurate provenance information when the processing steps within a workflow involves a computing processing element executing over multiple input views or producing multiple output data products per window execution. In other cases, these assumptions are not required (e.g. *average* operation).

4.6 WORKING PRINCIPLE OF BASIC PROVENANCE INFERENCE

As discussed in Section 4.4, the *basic provenance inference* method has three different phases. The first phase, *documentation of workflow provenance*, is the pre-requisite phase which has to be completed before the actual execution of the inference-based method. The basic provenance inference method

enters into the next two phases, only when fine-grained provenance information of an output data product is requested. Both of these phases, *backward* and *forward* computation facilitate the explicated workflow provenance information during their execution and eventually infer fine-grained data provenance of the selected output data product/tuple.

4.6.1 Documentation of Workflow Provenance

In this phase, the workflow provenance of the entire data processing is explicated. It includes documenting the values of the properties of different types of nodes present in the workflow provenance graph as discussed in Section 3.1. As shown in Figure 3.1, each type of nodes has the listed properties and during this phase, the value of these properties is set.

For computing processing elements, some extra properties such as number of input views, contributing input view are added based on the required information and assumptions discussed in Section 4.5. The explicated information of a computing processing element is quite similar to the *process provenance* reported in [116]. Process provenance describes the parameters of a single process to execute. The basic provenance inference method facilitates the documented properties of computing processing elements to infer fine-grained data provenance. Therefore, the following properties of a computing processing element must be documented during this phase based on the discussion in Section 3.1, Section 4.5 and a workflow model for continuous data [132].

- *Window type*: refers to a list of window types; one element for each input view. The value can be either *tuple* or *time*.
- *Window size*: refers to a list of window sizes; one element for each input view. The value represents the size of the window.
- *Trigger type*: specifies how a *computing processing element* will be triggered for execution; The value can be either *tuple* or *time*.
- *Trigger interval*: refers to the interval between successive executions of the same computing processing element.
- *Input-output ratio*: refers to the ratio of the number of input data products contributed to produce output data products over the number of output data products of a particular computing processing element.

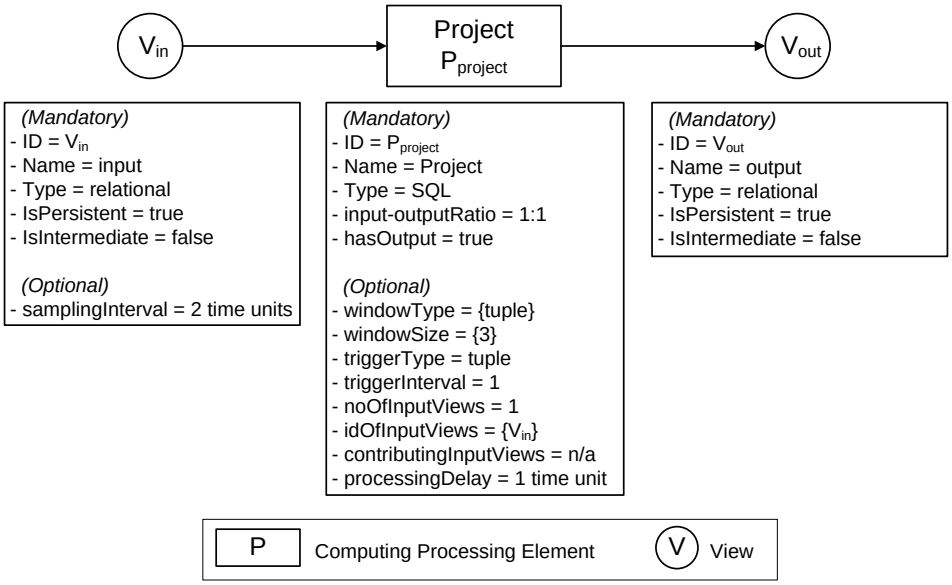


Figure 4.3: Example of the explicated workflow provenance

- *Number of input views*: refers to the total number of input views.
- *Identifier of input views*: refers to the list of ids (node identifiers) of input views.
- *Contributing input views*: refers to the fact that whether a computing processing element with multiple input views processes data products over *all* input views or a *specific* input view at a time. For computing processing elements with only one input view, it is set to *not applicable*.
- *Processing delay*: refers to the amount of time required by a computing processing element to complete the execution over the current window.

Furthermore, for each input view, *sampling interval* which refers to the amount of time between two successive tuples/data products insertion into that view, is also documented.

Example

Figure 4.3 shows an artificial, simple workflow and the explicated workflow provenance. In Figure 4.3, the workflow consists of a computing processing element implementing a *project* operation, $\mathrm{P}_{\mathtt{project}}$, which takes one input view $\mathrm{V}_{\mathtt{in}}$ as input and produces one output view, $\mathrm{V}_{\mathtt{out}}$. Moreover, we assume that, the sampling interval of the input view $\mathrm{V}_{\mathtt{in}}$, $\lambda_{\mathtt{in}} = 2$

time units and the window size, $WS_{in}^{project} = 3$ tuples. The processing element, $P_{project}$ will be executed after arrival of every tuple. Therefore, $TR_{project} = 1$ tuple. $P_{project}$ takes 1 time unit to process the current window which is the processing delay, $\delta_{project}$. A data tuple, t_j indicates that the *transaction time* of the tuple is j. Based on this settings of the workflow, the explicated workflow provenance of different nodes (e.g. V_{in}, $P_{project}$, V_{out}) is also depicted in Figure 4.3. The next two phases of the basic provenance inference method facilitates this documented workflow provenance information as shown in Figure 4.3 to infer fine-grained data provenance.

4.6.2 Backward Computation

The backward computation phase is executed based on the request initiated by a user to infer fine-grained data provenance of an output data product/tuple. Figure 4.4 depicts the working mechanism of the backward computation phase. The left-side of Figure 4.4 shows the available data products/tuples in both input and output view, i.e., V_{in} and V_{out}, respectively. The user chooses a tuple T from the output view V_{out}, for which he/she initiates the request to infer fine-grained data provenance. The tuple T is also referred to as the *chosen tuple*.

The backward computation phase reconstructs the original window over which the actual execution of the computing processing element, P_k, was taken place and the chosen tuple T was produced. To accomplish that, backward computation phase facilitates the explicated workflow provenance, shown in Figure 4.3, and the *transaction time* of the chosen tuple T. This reconstructed window is referred to as the *inferred window*. In Figure 4.4, the *transaction time* of the chosen tuple is 8 which is referred to as the *reference point* to calculate the boundary/interval of the inferred window.

The mechanism of the backward computation phase is given in Algorithm 4.1. First, the transaction time of the chosen tuple T, i.e., *reference point*, and number of input views are retrieved in line 1 and 2. The *processing delay* of P_k, δ_k is also retrieved in line 3. Then, for each input views, we retrieve it's *id*, *window type* and *window size* in line 5-7. Afterward, we compute the list of input tuples I_{C^i} for each input view V_i which form the inferred window. There are two specific functions to accomplish this task based on the type of the window. If the window is a tuple-based window, then the function `reconstructTupleWindow` in line 9-10 is executed. Otherwise, in case of a time-based window the other func-

Algorithm

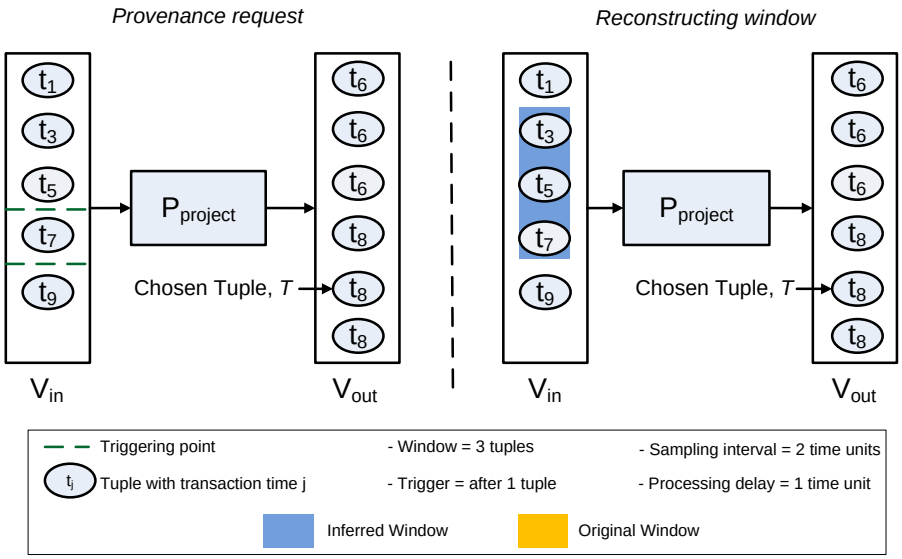


Figure 4.4: Illustration of the backward computation phase

tion `reconstructTimeWindow` in line 12-13 is executed. Both functions take exactly the same parameters.

Tuple-based windows

In case of a tuple-based window, the function `reconstructTupleWindow` calculates the *upper bound* or the ending edge of the inferred window defined over the input view V_i based on the following equation.

$$\mathrm{upperBound} = \mathrm{referencePoint} - \mathrm{processingDelay} \tag{4.1}$$

Please note that, the *upper bound* of the inferred window refers to a time value. After calculating the *upper bound*, the function `reconstructTupleWindow` considers only the latest $\mathrm{windowSize}$ number of tuples whose transaction time is less than or equal to the $\mathrm{upperBound}$ to form the inferred window.

Time-based windows

In case of a time-based window, the other function `reconstructTimeWindow` is executed. It calculates both the *upper bound* and the *lower bound* of the inferred window defined over the input view V_i based on the following equations.

$$\mathrm{upperBound} = \mathrm{referencePoint} - \mathrm{processingDelay} \tag{4.2}$$

$$\mathrm{lowerBound} = \mathrm{referencepoint} - \mathrm{processingDelay} - \mathrm{windowSize} \tag{4.3}$$

Algorithm 4.1: Backward Computation for Basic Provenance Inference

Input: An output tuple T produced by a processing element P_k, for which fine-grained provenance is requested

Output: Set of input tuples I_{C^i} for each input view V_i which form the inferred window producing T

```
1  referencePoint ← getTransactionTime(T);
2  noOfInputViews ← getNoOfInputViews(P_k);
3  processingDelay ← getProcessingDelay(P_k);
4  for i ← 1 to noOfInputViews do
5  |  inputView ← getInputViewID(P_k, i);
6  |  windowType ← getWindowType(inputView);
7  |  windowSize ← getWindowSize(inputView);
8  |  if windowType = "tuple" then     /* tuple-based windows */
9  |  |  I_Ci ← reconstructTupleWindow(inputView, windowSize,
10 |  |                    referencePoint, processingDelay);
11 |  else                             /* time-based windows */
12 |  |  I_Ci ← reconstructTimeWindow(inputView, windowSize,
13 |  |                    referencePoint, processingDelay);
14 |  end
15 end
```

Both *upper bound* and *lower bound* of the inferred window are time values. After calculating these values, the function `reconstructTimeWindow` considers all the tuples t_j satisfying the condition: $lowerBound \leqslant j < upperBound$ to construct the inferred window.

Example

The right-side of Figure 4.4 shows the *inferred window* based on Algorithm 4.1 for the workflow shown in Figure 4.3. In this workflow, the window is tuple-based. Therefore, according to Equation 4.1, the *upper bound* of the inferred window is calculated which is 7. Since the given window size is 3 tuples, the backward computation algorithm considers the latest 3 tuples having *transaction time* $\leqslant 7$ from the input view V_{in} to construct the inferred window and is shown by a light-blue shaded rectangle in the right-side of Figure 4.4. The list of tuples within the inferred window over an input view V_i is denoted as I_{C^i}. Therefore, the tuples enclosed within the light-blue shaded rectangle over V_{in} in Figure 4.4 represents the list of

tuples $I_{C^{in}}$ which is used in the forward computation phase of the basic provenance inference method.

4.6.3 Forward Computation

The last phase of the basic provenance inference method is the forward computation phase. In this phase, the inference-based method establishes the data-dependent relationship between the input data products/tuples and the chosen output data product/tuple. This data-dependent relationship is referred to as fine-grained data provenance. To infer accurate fine-grained data provenance, the forward computation phase facilitates the *input-output ratio* alongside some other properties of the processing element $P_{project}$ which have been already documented as workflow provenance information, discussed in Section 4.6.1.

Algorithm

Algorithm 4.2 describes the mechanism in the forward computation phase to infer accurate fine-grained data provenance. First, *number of contributing input tuples*, *number of produced output tuples*, *contributing input views* and *number of input views* are retrieved in line 2-5 from the explicated workflow provenance as shown in Figure 4.3. As mentioned in Section 4.6.2, the list of candidate input data products/tuples, I_{C^i}, for each input view V_i, might contribute to produce the chosen tuple *T*. The forward computation phase selects only the contributing input data products/tuples from I_{C^i} for each input view V_i and finally associates the selected input tuples to the chosen tuple *T*. This selection mechanism depends on the types of the computing processing elements as discussed in Section 4.5.1.

Union

Project & Cartesian product

For computing processing elements implementing operations where only one input tuple contributes to the output tuple such as a *project* or an *union* operation (line 6), we have to identify the relevant contributing input tuple. However, there are a few computing processing elements which involves multiple input views but only one input view is contributing at a time, i.e., *contributing input views* = "specific". As an example, a computing processing element executing an *union* operation falls into this category. To address these cases, line 8-10 in Algorithm 4.2 is executed. Otherwise, if the tuples in all the input views contribute at the same time to a particular output tuple, line 12-14 is executed. In both cases, we need to facilitate the assumptions on the *order of input views*, the *contributing input view* and *order of tuples in the output view*, discussed in Section 4.5.2, to determine the

Algorithm 4.2: Forward Computation for Basic Provenance Inference

Input: Set of input tuples I_{C_i} for each input view V_i which form the inferred window producing T
Output: Set of input tuples I which contributes to T

```
1  I = ∅;
2  noOfInputTuples ← getInputOutputRatio(PE,"input");
3  noOfOutputTuples ← getInputOutputRatio(PE,"output");
4  contributingInputViews ← getContributingInputViews(PE);
5  noOfInputViews ← getNoOfInputViews(PE);
6  if noOfInputTuples = 1 then            /* only one input tuple
   contributes */
7      if
       noOfInputViews > 1 ∧ contributingInputViews = "specific"
       then                                          /* e.g. union */
8          parentView ← getParentView(T);
9          tuplePosition ←
           getTuplePosition(T, parentView, noOfOutputTuples);
10         I ← selectTuple(I_{C parentView}, tuplePosition);
11     else          /* e.g. project, cartesian product etc. */
12         for i ← 1 to noOfInputViews do
13             tuplePostion ←
               getTuplePosition(T, i, noOfOutputTuples);
14             I ← selectTuple(I_{Ci}, tuplePosition) ∪ I;
15         end
16     end
17 else           /* all input tuples contribute (e.g. average,
   interpolation) */
18     for i ← 1 to noOfInputViews do
19         I ← I_{Ci} ∪ I;
20     end
21 end
```

position of the chosen tuple in the output view. Based on the position of the chosen tuple and the assumption on the *order of tuples in the output view* discussed in Section 4.5.2, we can select the input tuple which contributed to produce the output tuple T (in line 10 and 14).

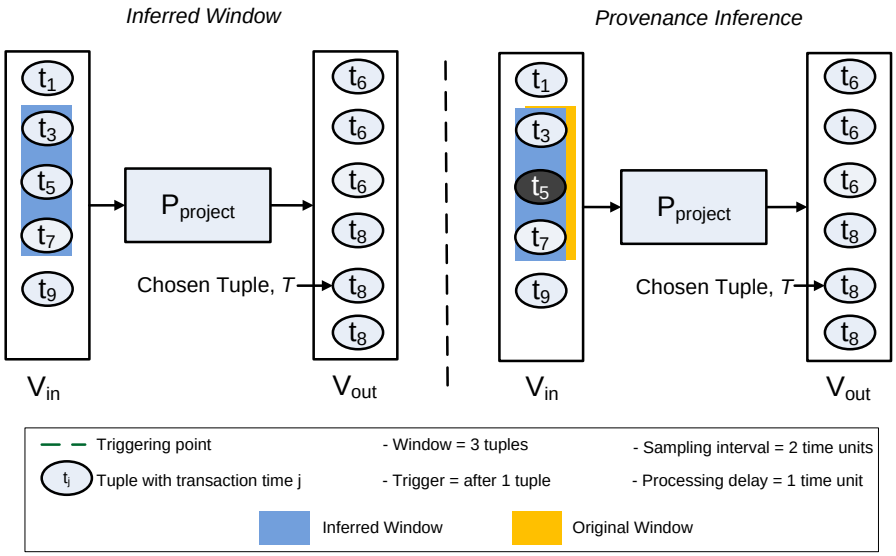


Figure 4.5: Illustration of the forward computation phase

Average & Interpolation

In cases where all input tuples contribute to the output tuple independent of the number of input views, all tuples accessible from all inferred windows are selected. Therefore, the set of contributing input tuples is the union of the set of candidate input tuples per input view (line 19).

Example

Figure 4.5 depicts the forward computation phase. In this example, since the computing processing element is performing a *project* operation, *number of contributing input tuples* and *number of produced output tuples* are 1 and only one input view participates to the processing (see Table 4.1). Therefore, we execute the segment reported in line 12-14 to infer the contributing input tuple and make a relationship between the contributing input tuple and the *chosen tuple*. At first, we determine the chosen tuple's *tuple position*. In this example, there are 3 tuples having *transaction time* equal to the chosen tuple's transaction time which is 8. Since the chosen tuple's position is second among them, $\texttt{tuplePosition} = 2$. Since, the value of *tuple position* is 2, we choose the 2nd tuple in the descending order of tuple appearance from the reconstructed window. The tuple, t_5 contributes to produce the chosen tuple T from the output view and is represented by the brown-shaded tuple within the inferred window in the right-side of Figure 4.5.

4.7 EVALUATION

The basic provenance inference method is evaluated based on the workflow presented in Section 4.2. The shaded part in Figure 4.2 is considered for this evaluation. The processing element P_5 in Figure 4.2 is implementing an *interpolation* operation which has the *input-output ratio* $= 3 : 9$. The view, V_4, is the input view of P_5 and the view, V_5 is the output view produced by P_5. The collection of tuples in both input and output view is referred to as *sensor data*. Since P_5 has the *input-output ratio* of $3 : 9$, the number of tuples in the output view, V_5, becomes three times the number of tuples in the input view, V_4. Therefore, in this case, the storage space consumed by the sensor data becomes quite high. To have a comprehensive evaluation considering different types of operations, we also evaluate the cases where P_5 implements other operations such as a *project* (input-output ratio $= 1 : 1$) and an *average* (input-output ratio $= n : 1$) operation where n be the window size.

4.7.1 Evaluation Criteria and Methods

The second research question (*RQ 2*) of this thesis is about the challenge of managing fine-grained data provenance under different system dynamics at reduced cost in terms of storage consumption as discussed in Section 1.4. The basic provenance inference method infers fine-grained data provenance at reduced storage cost assuming that the system has constant processing delay and regular arrival pattern of input data products, i.e., constant sampling interval. Therefore, the basic provenance inference method addresses *RQ 2* and provides a solution. Since the primary challenge introduced in *RQ 2* is to have fine-grained data provenance at reduced storage costs, the main evaluation criterion is the *storage consumption* of the basic provenance inference method. Furthermore, the overall goal of the inference-based framework is to provide accurate provenance information in a cost-efficient way. Therefore, the other evaluation criterion is the *accuracy* of the basic provenance inference method.

Explicit Provenance

Existing approaches [103, 26, 109, 108] record fine-grained data provenance explicitly in varying manners. Since details of these implementations are not available, the basic provenance inference method is compared with an implementation of a fine-grained data provenance documentation, re-

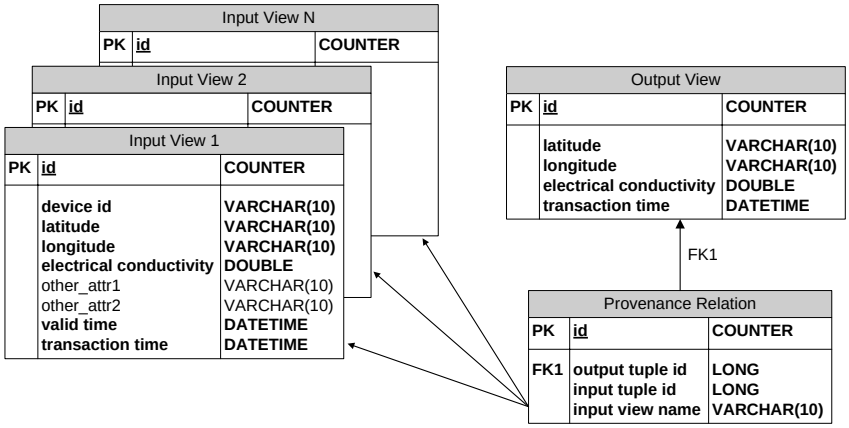


Figure 4.6: Schema diagram for Explicit Provenance method

ferred to as *explicit provenance* method. In the *explicit provenance* method, the derivation history (provenance) of each output tuple is annotated and inserted into a relation in a database. One relation per output view is maintained. Since tuples from multiple input views might contribute to produce tuples in the output view, we also need to keep the name of input view in provenance records. The annotation attributes to record provenance of an output tuple include: i) *output tuple id*, ii) *input tuple id* and iii) *input view name*. We also assign another attribute named as *id* which is auto incremental and serves as the primary key of this relation. Figure 4.6 shows the schema diagram of this method. In this case, if an output tuple is produced by the contribution of 3 input tuples from the same input view, the provenance relation contains 3 tuples with the same *output tuple id* but different *input tuple id*s having the same *input view name*. The size of the provenance relation shown in Figure 4.6, represents the storage consumption by the *explicit provenance* method.

Improved explicit provenance

The *explicit provenance* method is the simplest way to store provenance data. Therefore, we improve this explicit provenance collection system using the concept of *basic factorization* which is proposed in [26]. We refer to this technique as *improved explicit provenance* method. The main concept of this method is that if an input tuple from a particular input view contributes several times to produce multiple output tuples, only one record about input tuple's information, consisting of *input tuple id* and *input view name*, will be kept in a separate relation. Afterward, this particular record can be pointed by provenance relation to attach the input tuple's informa-

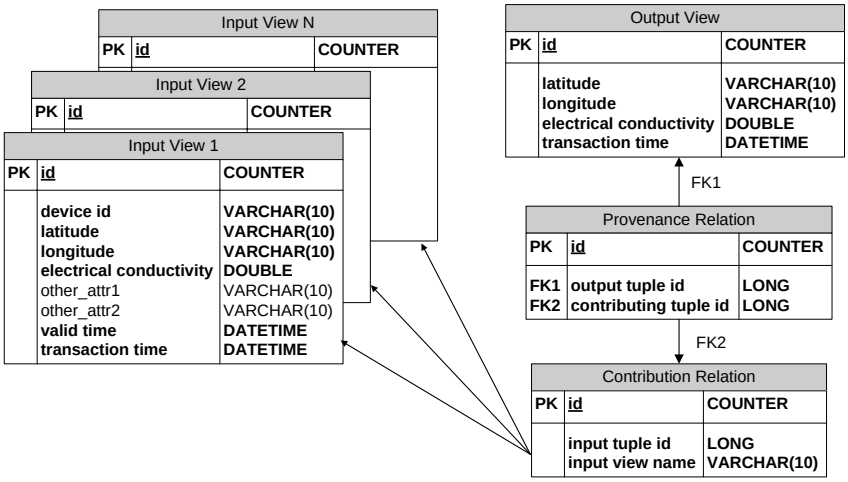


Figure 4.7: Schema diagram for Improved Explicit Provenance method

tion with appropriate output tuples' information. The higher the number of output tuples and the higher the number of contribution of one input tuple to produce output tuples, the more the *improved explicit method* can reduce storage consumption to maintain provenance data compared to the *explicit provenance* method.

Figure 4.7 shows the schema diagram of this method. In this method, the contributing input tuples are stored into a separate relation, called contribution relation. Then, using the concept of *foreign key*, we associate output data tuples to their corresponding input data tuples. Therefore, in this case, two relations per output view are maintained. Contribution relation holds the reference of input data tuples with an *id* (auto incremental), *input tuple id* and *input view name* as depicted in Figure4.7. The provenance relation associates these input tuples to output tuples containing *id*, *output tuple id* and *contributing tuple id* where *contributing tuple id* is the foreign key referred to the contribution relation. The size of these two relations represent the storage consumption by the *improved explicit provenance* method.

For the *basic provenance inference* approach, the storage consumption due to the inclusion of the *transaction time*, i.e., system timestamps, to each tuple in both input and output views is considered as the storage cost to infer data provenance. We compare the storage cost of these three approaches in different test cases described in Section 4.7.3.

Finally, it is also important to check whether the *basic provenance inference* method infers accurate provenance information or not. To check the

Table 4.2: Parameters of Different Test Cases used for the Evaluation

PARAMETERS	TEST CASES			
	Tuple-based Windows		Time-based Windows	
	Overlap	*Non-overlap*	*Overlap*	*Non-overlap*
Window size	3	3	6 s	6 s
Trigger interval	1	3	2 s	6 s
Sampling interval	2 s	2 s	2 s	2 s
processing delay	1 s	1 s	1 s	1 s

accuracy of the inference-based method, the fine-grained data provenance provided by the *explicit provenance* method is used as the ground truth and it is compared with the fine-grained data provenance inferred by the *basic provenance inference* method.

4.7.2 Dataset

For the evaluation, a real dataset[3] measuring electrical conductivity of water, collected by the RECORD project, discussed in Section 4.1, is used. The workflow operating on this dataset has been discussed in Section 4.2. The experiments are performed on a underlying PostgreSQL 8.4[4] database and the Sensor Data Web[5] platform. The input dataset contains 30000 tuples representing a six-month period from July-December 2009 and requires 7200 KB of storage space.

4.7.3 Test cases

To compare the storage consumption among the *basic provenance inference*, the *explicit provenance* and the *improved explicit provenance* method, we use a few test cases. All these test cases are based on sliding windows. However, there is a variation in the type of windows as well as the amount of slide.

3 Availableathttp://data.permasense.ch/topology.html#topology
4 Available at http://www.postgresql.org/
5 Available at http://sourceforge.net/projects/sensordataweb/

Table 4.2 shows the window size, trigger interval and sampling interval of the 4 test cases, which are considered for this evaluation. Test cases based on the tuple-based windows have window size 3 with overlapping and non-overlapping variation. We have chosen the window size of 3 based on the artificial workflow described in Section 4.2, where we assumed that input data products were sent by 3 different sensors located in different cells of a 3×3 grid to calculate the interpolated values for all cells within the grid. The other two test cases are defined based on time-based windows having window size of 6 seconds with overlapping and non-overlapping variation. In all cases, data products at the input view arrive after every 2 seconds which is referred to as the sampling interval. Evaluation using these test cases allows us to report results covering different system settings.

4.7.4 Storage Consumption

Interpolation Operation

Firstly, the storage consumption managing fine-grained provenance data by the *explicit provenance*, *improved explicit provenance* and the *basic provenance inference* method are investigated. In this experiment, we measure the storage overhead to maintain fine-grained data provenance for the *Interpolation* processing element based on the workflow described in Section 4.2 using the test cases mentioned in Section 4.7.3.

In the non-overlapping tuple-based window case, each window contains 3 tuples and the computing processing element performing an interpolation operation is executed for every third arriving tuple. This results into $30000 \div 3 \times 9 = 90000$ output tuples since the interpolation operation is executed for every third input tuple and it produces 9 output tuples at a time because the area of the river is divided into 3×3 cells based on the scenario discussed in Section 4.1. It produces the output view which requires about 4200 KB of storage space. In the overlapping tuple-based window case, the window contains 3 tuples and the operation is executed for every tuple. This results into $30000 \times 9 = 270000$ output tuples since for every input tuple the computing processing element is executed and producing 9 output tuples, which requires about 12650 KB. As already mentioned, the collection of both input and output tuples together is referred to as the *sensor data*, the size of sensor data becomes 7200KB+4200KB = 11400KB and

Sensor data

7200KB+12650KB = 19850KB for the non-overlapping and the overlapping tuple-based window case, respectively.

Explicit provenance

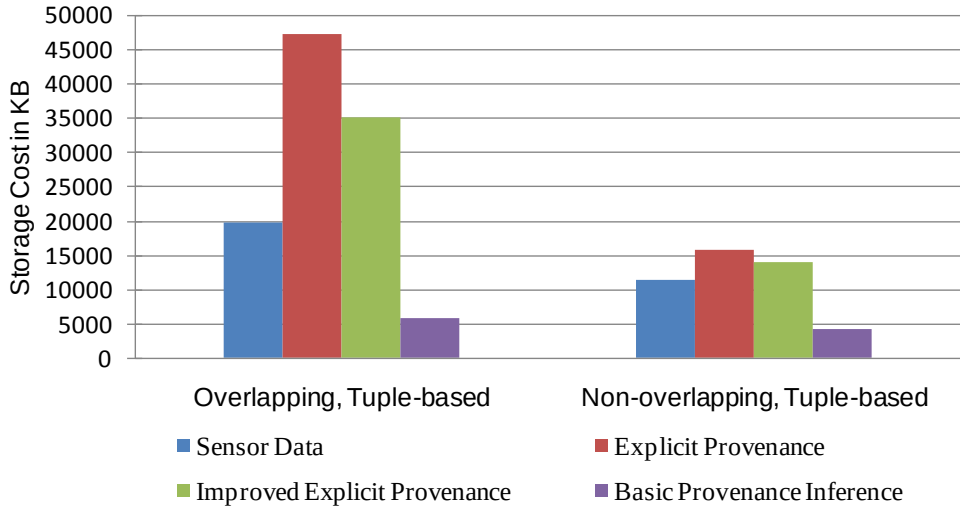
For the *explicit provenance* method to maintain fine-grained data provenance, the relation between input and output tuples are enumerated as shown in Figure 4.6. In the non-overlapping tuple-based window case, the relation contains $90000 \times 3 = 270000$ tuples since each output tuple is produced by the contribution of 3 (window size) input tuples. It requires about 16000 KB of storage space. In the overlapping tuple-based window case, the relation maintaining provenance contains $270000 \times 3 = 810000$ tuples and requires about 47500 KB of storage space. Therefore, the *explicit provenance* method takes about 3 times more storage space in the overlapping case than in the non-overlapping case.

Improved explicit provenance

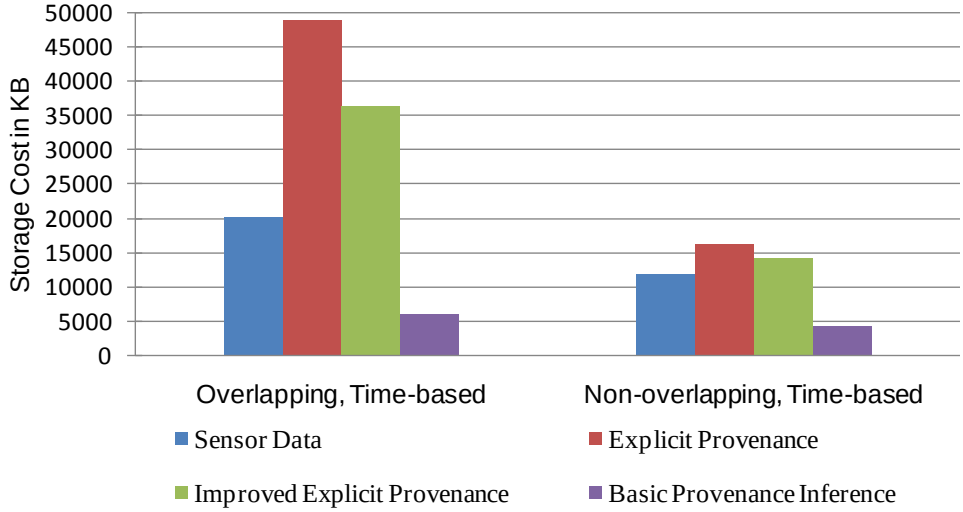
The *improved explicit provenance* method takes less storage space than the *explicit provenance* method in these two cases. It means that there are usually higher number of input tuples which contributed several times to produce output tuples. It happens for two reasons. Firstly, in the overlapping tuple-based window case, an input tuple contributed several times to produce output tuples because of the overlapping between two successive windows. Secondly, in both overlapping and non-overlapping cases, the computing processing element produced multiple output tuples from the same set of input tuples (window) because of the nature of the *interpolation* operation. The *improved explicit approach* takes 14100 KB and 35200 KB of storage space, which are 89% and 74% of storage space consumed by the *explicit provenance* method for non-overlapping and overlapping tuple-based window case, respectively.

Basic provenance inference

The *basic provenance inference* method has the least storage overhead among these three methods. The storage consumption of the *basic provenance inference* method is calculated by the space required to store the *transaction time* for all tuples in the *sensor data*, i.e., tuples in both input and output view. The *basic provenance inference* method requires 4220 KB and 5880 KB of storage space for the non-overlapping and the overlapping tuple-based window case, respectively. The *explicit provenance* takes 3.75 and 8.07 times more space than the *basic provenance inference* method for non-overlapping and overlapping tuple-based window case, respectively. The *improved explicit provenance* takes around 3.35 and 6 times more space than the *basic provenance inference* method for non-overlapping and overlapping tuple-based window case, respectively. Therefore, the *basic provenance inference* method clearly outperforms the other two methods. The ba-



(a) Tuple-based Window executing *Interpolation* operation



(b) Time-based Window executing *Interpolation* operation

Figure 4.8: Storage cost associated with *Interpolation* operation for different test cases

sic provenance inference approach requires only one attribute, *transaction time*, to be attached to each input and output tuple, to apply the inference mechanism. Therefore the storage consumption associated with the basic provenance inference method is independent on the window size and the amount of overlaps between windows unlike the other two methods. The

storage consumption of the basic provenance inference depends on the amount of *sensor data*, i.e., both input and output tuples, only.

Results of Tuple-based windows

Figure 4.8a shows the aforesaid numbers representing the storage cost associated with different methods for tuple-based windows. From Figure 4.8a, one can observe that in both overlapping and non-overlapping cases, the storage space consumed by the *explicit provenance* and the *improved explicit provenance* method are bigger than the size of the sensor data. Especially in the overlapping tuple-based window case, the size of provenance data becomes almost 2.5 and 1.8 times bigger than the size of the sensor data. The ratio of the size of provenance data over the size of sensor data depends on the following factors: i) window size, ii) amount of overlaps between windows, iii) input-output ratio of the computing processing element and iv) the size of an input tuple. The bigger the window size and overlapping between windows, the higher the ratio of the provenance data size over the sensor data size. If a computing processing element has an *input-output ratio* of $n : m$ where $m > n$, i.e., implementing an *interpolation* operation, the ratio becomes higher. Finally, the ratio also depends on the size of an input tuple. A tuple in the input dataset may contain other values except the value of interest, i.e., electrical conductivity in this case. The higher the number of other values in the input dataset, the lower the ratio becomes since the increasing size of an input tuple eventually increases the size of sensor data.

Results of Time-based windows

Figure 4.8b shows the storage consumption to maintain provenance by different methods in case of the overlapping and the non-overlapping time-based windows. Figure 4.8b represents numbers which are similar to the ones shown in Figure 4.8a. Since the window size of time-based window is 6 seconds and the sampling interval is 2 seconds (see Table 4.2), each window of 6 seconds can hold 6 seconds$\div$2 seconds/tuple $=$ 3 tuples exactly which is same to the window size of the tuple-based windows. Furthermore, in the overlapping time-based window case, the computing processing element triggers after 2 seconds, i.e., after 2 seconds$\div$2 seconds/tuple $=$ 1 tuple, which is the same to the trigger interval of the overlapping tuple-based window case. In the non-overlapping time-based window case, the computing processing element triggers after 6 seconds, i.e., after 6 seconds$\div$2 seconds/tuple $=$ 3 tuples, which is also the same to the trigger interval of the non-overlapping tuple-based window case. Therefore, the result of time-based windows is also similar to the one of tuple-based windows in both overlapping and non-overlapping cases. We keep similar

settings in both tuple-based and time-based windows that is keeping 3 tuples per window, because of the artificial workflow discussed in Section 4.2 where we assumed that input data products were sent by 3 different sensors located in different cells of a 3×3 grid to calculate the interpolated values for all cells within the grid.

Project operation

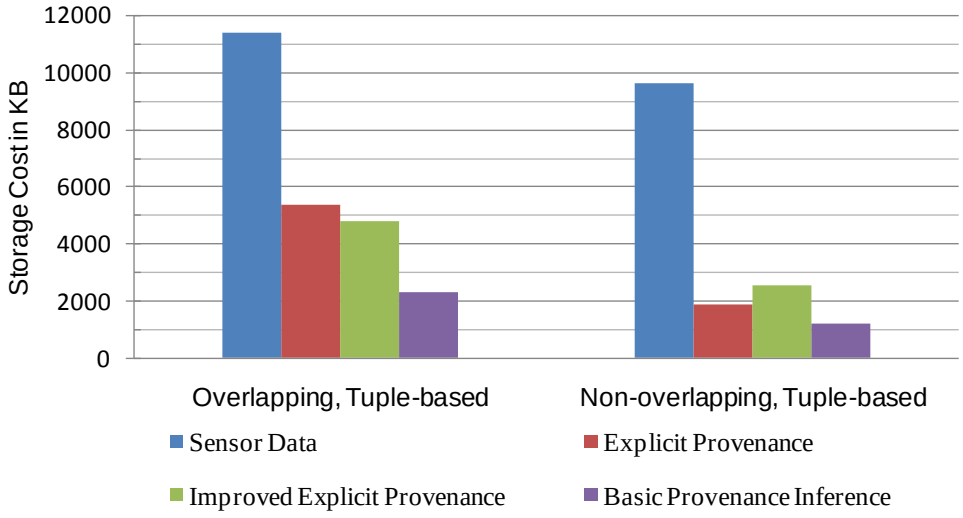
Additional tests for different operations are also performed. Figure 4.9 shows the storage consumption comparison among the *explicit provenance*, *improved explicit provenance* and the *basic provenance inference* method for *project* and *average* operations. We report the result of both *project* and *average* operation using the test cases with tuple-based windows only. The results using the test cases with time-based windows are not reported since they are similar to the ones using the tuple-based windows. It occurs due to the fact that the window size and the trigger interval are similar in both time-based and tuple-based windows as explained before.

Results of Project operation

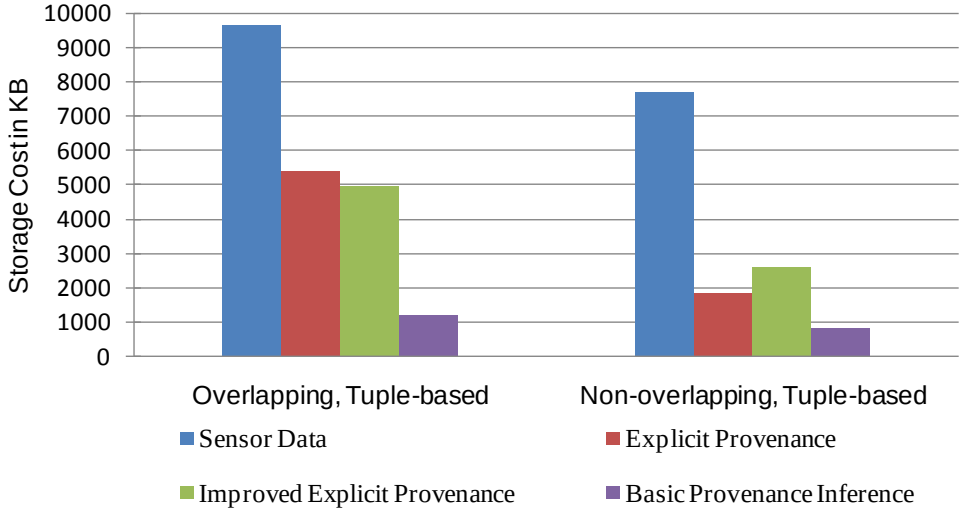
Figure 4.9a shows the storage space consumed by different methods for both overlapping and non-overlapping tuple-based windows, executing a *project* operation. A *project* operation with window size of 3 tuples in overlapping case produces $30000 \times 3 = 90000$ output tuples. In the non-overlapping case, it produces 30000 tuples$\div$3 tuples$\times$3 tuples = 30000 output tuples. In both cases for a *project* operation, the *basic provenance inference* approach has the lowest storage overhead compared to the other methods. The *basic provenance inference* method requires 1200 KB and 2320 KB of storage space for non-overlapping and overlapping tuple-based window case, respectively. The *explicit provenance* takes 1.55 and 2.33 times more space than the *basic provenance inference* method for non-overlapping and overlapping tuple-based window case, respectively. The *improved explicit provenance* takes 2 times more space than the *basic provenance inference* method for both non-overlapping and overlapping tuple-based window cases.

Analysis on size of Sensor Data

There are a few interesting observations that can be made from Figure 4.9a. The size of sensor data in both overlapping and non-overlapping cases are bigger than the storage space consumed by different methods to maintain provenance data. The reasons for this are twofold. Firstly, sensor data includes the complete input dataset which might contain values not used at all in the current workflow. Secondly, the number of output tuples produced by a computing processing element implementing a *project* oper-



(a) Tuple-based Window executing *Project* operation



(b) Tuple-based Window executing *Average* operation

Figure 4.9: Storage cost associated with *Project* and *Average* operation for different test cases

ation, is lower than the number of output tuples produced by a computing processing element, implementing an *interpolation* operation. It means that the provenance managing methods have less provenance data tuples to maintain and therefore, it reduces the storage consumption. One may argue that since the number of output tuples produced in this case is lower,

the storage space consumed by the output tuples also gets lower and so does the size of sensor data. However, this is not true since the output tuples have relatively very small payload than the input tuples and thus, the decreasing number of output tuples do not have any visible effect on the size of sensor data.

Analysis on Improved explicit provenance

Another observation is that, the *improved explicit provenance* takes more space than the *explicit* method in non-overlapping cases. Since windows have no overlaps between each other and the computing processing element implementing the *project* operation does not produce multiple output data products, there are no input tuples that contribute to produce multiple output data products. Therefore, the *improved explicit provenance* method cannot reduce the storage costs in non-overlapping cases.

Average Operation

Results of Average operation

We also perform the experiment with an *average* operation which is an aggregate function producing only one output tuple per window. Figure 4.9b shows the storage space consumed by the different methods for both overlapping and non-overlapping tuple-based windows, executing an *average* operation. An *average* operation with window size of 3 tuples in the overlapping case produces 30000 output tuples. In a non-overlapping case, it produces $30000 \div 3 = 10000$ output tuples. In both cases for an *average* operation, the *basic provenance inference* approach has the lowest storage overhead compared to the other methods. The *basic provenance inference* method requires 820 KB and 1220 KB of storage space for non-overlapping and overlapping tuple-based window case, respectively. The *explicit provenance* takes 2.25 and 4.5 times more space than the *basic provenance inference* method for non-overlapping and overlapping tuple-based window case, respectively. The *improved explicit provenance* takes 3.17 and 4.13 times more space than the *basic provenance inference* method for both non-overlapping and overlapping tuple-based window cases.

Results analysis

Based on this results, we can observe that the *improved explicit provenance* method takes more space than the *explicit provenance* method to maintain provenance data in the non-overlapping case, performing an *average* operation. A similar observation has been also made for a *project* operation over the non-overlapping case. The reason remains the same as discussed before. Therefore, as mentioned in Section 4.7.1, the *improved explicit provenance* method can perform better than the *explicit provenance* method when

an input tuple contributes several times, producing multiple output tuples. In all test case, the proposed *basic provenance inference* method outperforms the other two methods keeping explicit provenance records.

The reported ratio depends on the window size and overlap between windows. If the window size is larger and there is a big overlap between windows, the *basic provenance inference* method performs even better. We have not compared the storage consumption between the proposed method and any standard data compression technique though. However, an obvious advantage of the proposed method is that it does not require any pre-processing on data unlike a data compression method.

4.7.5 Accuracy

The *accuracy* of the basic provenance inference method is measured by comparing the inferred fine-grained data provenance of each output data product to the ground truth provided by the explicit provenance method for a particular test case. As discussed in Section 4.4, the basic provenance inference method assumes that the computing processing elements have constant processing delay and the input data tuples arrive at a regular interval as shown in Table 4.2. Because of these two assumptions, the proposed method infers 100% accurate provenance traces for all test cases.

4.8 DISCUSSION

The inference-based methods have a few requirements to satisfy. Most of the requirements are already introduced to process data streams in existing literature. In [103], authors propose to use transaction time on incoming stream data. Ensuring temporal ordering of data tuples is one of the main requirements in stream data processing. Moreover, several studies [46, 116] have proposed to maintain process level provenance which correspond to the documentation of workflow provenance.

Assumptions

There are a few assumptions which need to be satisfied based on the type of computing processing elements to infer accurate provenance information. These assumptions are discussed in Section 4.5.2 indicating that the inference mechanism must know the order of input views and the contributing input view in cases the operation to be performed has multiple input views. Furthermore, one of these assumptions also has to be satis-

fied to ensure that the order of tuples in the output view follows the same order found in input views. If these assumptions are not fulfilled by the underlying system, the proposed method cannot be applied.

Variable ratio operations

The basic provenance inference method infers provenance for *constant ratio* operations. A *variable ratio* operation does not satisfy the assumptions on the order of tuples in the output view discussed in Section 4.5.2 and hence, the proposed method cannot be applied on these operations directly. However, transforming a variable ratio operation into a constant ratio operation can overcome this difficulty. This transformation is possible by introducing *NULL* tuples in the output.

Solution for high selectivity

Suppose, for a *select* operation, the input tuple which does not satisfy the selection criteria will produce a *NULL* tuple in the output view, i.e., a tuple with a *transaction time* attribute and the remaining attributes are *NULL* values. This is how, it is ensured that the *select* operation now has 'one to one' input-output ratio and therefore, it is a constant ratio operation which satisfies all the requirements to apply the inference-based method. However, if the selectivity rate is very low for a given condition on a particular dataset, this transforming incurs storage overhead by introducing many *NULL* tuples. Based on the experiments, we have observed that if the selectivity rate is more than 60%, it is cost-efficient in terms of storage to apply the concept of *NULL* values for a variable ratio operation.

Solution for low selectivity

If the selectivity rate is less than 60%, the aforesaid approach could take more storage space than the *explicit provenance* method. To address variable ratio operations with low selectivity, we outline an alternative approach. In this approach, a *select* operation is considered to have a default 'many to many' input-output ratio. In this case, based on a fine-grained provenance request for a selected output tuple, the basic provenance inference method can infer a set of input tuples (the *inferred* window). One of these input tuples actually contributes to produce the selected output tuple. In this approach, the inferred provenance information might not be as precise as it is in the aforesaid approach ('one to one' ratio) but it certainly reduces storage consumption and provides all necessary information.

4.9 SUMMARY

The basic provenance inference method infers fine-grained data provenance accurately at reduced storage costs. The design and development

of this method was motivated by the second research question (*RQ 2*) which mentioned the challenge of managing fine-grained data provenance at reduced storage consumption. To explain the working principle of the proposed method, we facilitated a simple workflow that captures sensor measurements on electrical conductivity and produces interpolated values to generate a contour map. We also introduced a few basic concepts that help to explain the inference-based method.

The basic provenance inference method infers fine-grained data provenance by facilitating the workflow provenance and the timestamps, also referred to as the *transaction time*, attached to all input and output data products/tuples. The method has three major phases. In the first phase, the workflow provenance is documented. This phase is executed only once during the setup of the workflow. The next two phases are executed only once the user requests provenance for an output data product. In the second phase, the basic provenance inference method reconstructs the actual window which had taken part during the execution. The reconstructed window is referred to as *inferred window*. During this phase, the method exploits the values of different properties of the particular computing processing element such as window size, processing delay etc. Finally, the basic provenance inference method associates the selected output data product with the contributing input data products. During this phase, the method takes a few parameters of associated computing processing elements such as *input-output ratio* into account.

We evaluated the storage consumption and the accuracy of the basic provenance inference method by comparing it to a few other methods such as the explicit provenance and the improved explicit provenance method for different types of operations. Our evaluation shows that the basic provenance inference method takes least storage space to maintain fine-grained data provenance in all test cases compared to the other methods. The basic provenance inference can reduce storage consumption at higher magnitude if the window size and the overlaps between windows is bigger. The accuracy of the basic provenance inference method is calculated by comparing the inferred provenance information to the ground truth provided by the explicit provenance collection method. The basic provenance inference method infers 100% accurate provenance information when the processing delay remains constant and input data products arrive at a regular interval.

5

PROBABILISTIC PROVENANCE INFERENCE

INFERENCE of fine-grained data provenance allows the scientists to achieve accurate provenance data at reduced storage costs. Scientists could exploit this provenance data to debug a scientific model, to validate a model as well as to reproduce results. In Chapter 4, we presented the *basic provenance inference* method that can infer fine-grained data provenance at reduced storage consumption handling both offline (non-stream) data and data streams. Moreover, the basic provenance inference method infers 100% accurate provenance information under a particular system dynamics. The system dynamics refers to the set of parameters that control the nature of how the data products are arriving into the system for processing and when the input data products are processed. To be more specific, the system dynamics depends on the following parameters:

1. Processing delay or δ_k refers to the amount of time, required to complete the execution of a computing processing element, P_k, over the current window defined on the input view V_i.

2. Sampling interval or λ_i refers to the amount of time between two successive input data products insertion into an input view, V_i.

Challenges

The basic provenance inference method infers 100% accurate provenance under the assumption that the underlying system has constant δ_k and con-

This chapter is based on the following work: Probabilistic Inference of Fine-Grained Data Provenance. In *Database and Expert Systems Applications (DEXA'12)*, volume 7446 of *LNCS*, pages 296–310, Springer, 2012. & Adaptive Inference of Fine-grained Data Provenance to Achieve High Accuracy at Lower Storage Costs. In *Proceedings of the IEEE International Conference on E-Science (e-Science'11)*, pages 202–209, IEEE Computer Society, 2011.

stant λ_i. However, in a typical system, due to other workload and the nature of the particular computing processing element, the processing delay may vary. As an example, if a computing processing element performs a *greatest common divisor* operation, the amount of time required to finish the execution depends on the number of iterations the computing processing element should perform and thus, it could result in a variable processing delay for the repeated execution of the same computing processing element. Furthermore, the sampling interval of input data products could also vary due to a number of reasons such as broken sensors, network delay etc.

Solution criteria

Therefore, we need a more sophisticated method that can infer fine-grained data provenance under variable processing delay and sampling interval. Moreover, it has to be ensured that the new inference-based method can infer fine-grained data provenance at a reduced storage costs like the basic provenance inference method. The second research question (*RQ 2*), introduces this challenge to infer fine-grained data provenance under different system dynamics at reduced cost in terms of storage consumption as discussed in Section 1.4.

Probabilistic Provenance Inference

In this chapter, we present the *probabilistic provenance inference* method that infers fine-grained data provenance by facilitating the given distribution of δ_k and λ_i meeting the challenge of the ability to infer under different system dynamics as mentioned in *RQ 2*. The *probabilistic provenance inference* method reconstructs the *inferred window* in such a way that the accuracy of the inferred fine-grained data provenance is optimized. At the time of reconstructing the inferred window, the method calculates an *offset* value which determines the distance of the shift of the window guaranteeing optimal accuracy. The *offset* value is calculated based on the relationship between the δ_k and the λ_i distribution.

The probabilistic provenance inference method has further advantages to offer. Since this inference-based method depends on the given processing delay and sampling interval distribution of the system, the probabilistic provenance inference method can estimate the achievable accuracy of the inferred provenance at design time. Therefore, the scientists could have an estimation on the performance of the inference-based method before deciding to apply it to infer fine-grained data provenance.

Chapter structure

This chapter is structured in the following way. First, we present a scenario based on a real project followed by the description of the example workflow associated with the scenario. Next, we describe a few basic con-

cepts used to explain the probabilistic provenance inference method. Based on these concepts, we present a few cases which explains the limitations of the basic provenance inference method based on which the probabilistic provenance inference method has to be designed. The overview of the probabilistic provenance inference method is given afterward followed by a brief discussion on the requirements and assumptions to be fulfilled to apply this inference-based method. Next, we explain the working principle of the probabilistic provenance inference method. Eventually, we evaluate this method using both real datasets and simulations followed by a brief discussion on the applicability of this method in different situations.

5.1 SCENARIO AND WORKFLOW DESCRIPTION

We use the scenario introduced in Section 4.1 to explain the probabilistic provenance inference method. In this section, we provide a brief outline of the scenario and the simplified workflow, defined based on this scenario.

RECORD[1] is one of the projects in the context of the Swiss Experiment[2], which is a platform to enable real-time environmental experiments. In this project, different types of input data products are acquired by several sensors which have been deployed to monitor river restoration effects. Among these data products, electrical conductivity of the water is also measured which represents the level of salt in water. Scientists are interested to control the operation of a drinking water well by facilitating the available sensor data reporting electrical conductivity.

In the context of the aforesaid scenario, we construct an artificial and simplified workflow which is used to explain the mechanism of the probabilistic provenance inference method. Figure 5.1 shows the simplified workflow. We assume that there are three sensors measuring electrical conductivity in three different locations. These sensors send data tuples containing the `device id`, the `latitude` and the `longitude` of the location, the measured `electrical conductivity`, the `timestamp` of the measurement, also referred to as *valid time* [79], along with some other attributes. Tuples sent by these sensors are acquired by the source processing elements SP_1, SP_2 and SP_3 (see Figure 5.1). Scientists combine these sensor readings and store these data tuples in the view V_4. Later, scientists define a window

Workflow description

1 Available at `http://www.swiss-experiment.ch/index.php/Record:Home`
2 Available at `http://www.swiss-experiment.ch/`

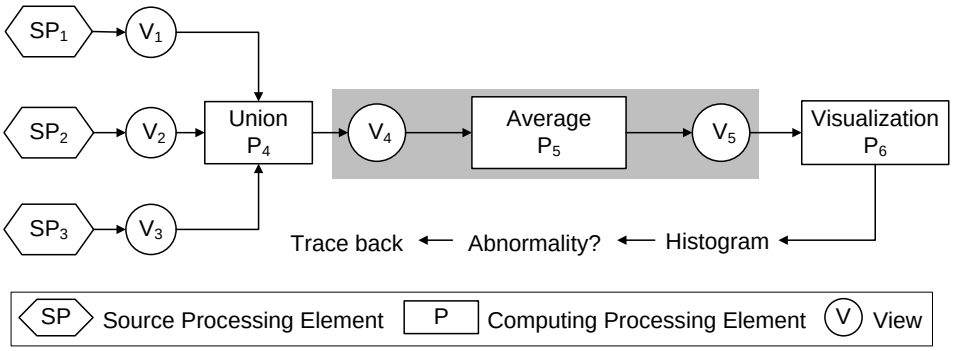


Figure 5.1: The example workflow

(tuple-based or time-based) over the view V_4 and calculate an average of the electrical conductivity over the defined window by executing the computing processing element P_5. P_5 stores the result in the view V_5. P_5 can be executed repeatedly based on a specific trigger interval, producing average of electrical conductivity over a particular period defined by the associated window. Later, a histogram, representing the average electrical conductivity, can be prepared. Scientists can request the provenance of any of the values within the histogram if the value seems to be an abnormal/unexpected one. The shaded part of the workflow in Figure 5.1 is considered to evaluate the probabilistic provenance inference method.

5.2 BASIC TERMINOLOGY

In this section, the definition of the terms which are used to explain the probabilistic provenance inference method is given. First, we restate the definitions of some of these terms which have already been introduced in Section 4.3.

- *Views:* A view V_i can be defined as a set of tuples ${t_j}^i$ where j is the *transaction time* [79]. The transaction time, j, refers to the system timestamp indicating the point in time when the tuple is inserted into the view V_i.
- *Sampling Interval:* Tuples can be inserted into a view V_i either at a regular interval or in an arbitrary manner. The amount of time between two successive tuples insertion into a view V_i is referred to as sampling interval, λ_i.

- *Computing Processing Elements:* A computing processing element, P_k, represents an operation that either computes a value/data product or writes data products into a file, database etc. It takes views as input and produces another view as output.

- *Windows:* A computing processing element, P_k, requires a window to be defined over the input view for its successful execution in the context of data streams. A window $(W_i{}^n)_k$ is a subset of tuples within a view V_i at the n^{th} execution of P_k. A window could be either tuple-based or time-based. A tuple-based window can be defined based on two parameters: i) window size m and ii) a point in time T. A tuple-based window is a finite subset of V_i containing the latest m number of tuples $t_j{}^i$ where $j \leqslant T$. The *window size* is represented as $WS_i{}^k$ where, $WS_i{}^k = m$ (number of tuples). In a time-based window, tuples whose *timestamp* falls into a specific boundary constitutes a window. A time-based window $(W_i{}^n)_k = [start, end)$ is a finite subset of V_i containing all tuples $t_j{}^i$ where $start \leqslant j < end$. In cases of time-based windows, the window size $WS_i{}^k = end - start$ (amount of time units).

- *Trigger Interval:* A trigger interval, TR_k, refers to the predefined interval between two successive executions of a computing processing element, P_k. The trigger interval of a computing processing element could be either tuple-based or time-based.

- *Processing Delay:* The amount of time to complete the execution of a processing element, P_k, after it is triggered, is referred to as processing delay δ_k.

Additional Terms

Unlike the *basic provenance inference* method, the *probabilistic provenance inference* method can infer fine-grained data provenance with variable processing delay (δ_k) and variable sampling interval (λ_i). The probabilistic provenance inference method takes the given distribution of processing delay and sampling interval into account to infer provenance information. Moreover, while inferring provenance, the probabilistic provenance inference also exploits two other variables and their corresponding distributions. Therefore, the following terms are required to be defined in addition to the aforesaid terms.

- *Sampling Interval Distribution:* The value of λ_i can change over time period if tuples are inserted into the view V_i in an arbitrary man-

ner. Therefore, in this case, λ_i becomes a discrete random variable which has *integer* values, defined over time domain. The distribution of the values of λ_i is referred to as the sampling interval distribution, denoted as $P(\lambda_i)$ and the probability of $\lambda_i = x$ is represented as $P(\lambda_i = x)$.

- *Processing Delay Distribution:* The value of δ_k can also change over time due to the system workload and the nature of the computing processing element P_k. Therefore, in this case, δ_k is a discrete random variable which has *integer* values, defined over time domain. The distribution of the values of δ_k is referred to as the processing delay distribution, denoted as $P(\delta_k)$ and the probability of $\delta_k = y$ is represented as $P(\delta_k = y)$.

- *First-tuple appearance Interval:* It refers to the amount of time between a particular window starts which is defined over the view V_i to execute P_k and arrival of the first tuple within that window. First-tuple appearance interval is denoted as ${\alpha_i}^k$. Since the appearance of the first tuple within a window depends on the sampling interval λ_i and λ_i could be variable as discussed, the value of ${\alpha_i}^k$ also changes over the time. Therefore, ${\alpha_i}^k$ becomes a discrete random variable over time domain. The distribution of the values of ${\alpha_i}^k$ is denoted as $P({\alpha_i}^k)$.

- *Last-tuple disappearance Interval:* It refers to the amount of time between the arrival of the last tuple within a particular window which is defined over the view V_i to execute P_k and the triggering point of that window. Last-tuple disappearance interval is denoted as ${\beta_i}^k$. Like ${\alpha_i}^k$, ${\beta_i}^k$ can be also deduced from the sampling interval λ_i. Therefore, the value of ${\beta_i}^k$ changes over the time which indicates that ${\beta_i}^k$ becomes a discrete random variable over time domain. The distribution of the values of ${\beta_i}^k$ is denoted as $P({\beta_i}^k)$.

The aforesaid terms are used to explain the working principle of the probabilistic provenance inference method presented in this chapter.

5.3 INACCURACY IN BASIC PROVENANCE INFERENCE

The *basic provenance inference* might infer inaccurate provenance under the system dynamics with variable processing delay and variable sampling

interval. The particular situation of an actual window for which the inaccuracy occurred is referred to as a *Failure Condition*. We illustrate a couple of cases, considering both tuple-based and time-based windows, where the basic provenance inference method cannot infer accurate provenance and define the corresponding failure condition accordingly.

5.3.1 Failure Condition for Tuple-based Windows

In case of tuple-based windows, the basic provenance inference method may infer inaccurate provenance information if a new input data product/tuple arrives and is inserted into the input view before completing the execution of the current window.

Figure 5.2 shows examples of the execution of the computing processing element P_{avg}. We assume that P_{avg} takes tuples for processing from input view V_{in} and after completing the execution, it produces output tuples, inserted into the view V_{out}. The window size of V_{in}, $WS_{in}{}^{avg} = 3$ tuples. P_{avg} triggers after the arrival of every tuple which means $TR_{avg} = 1$ tuple. Furthermore, we also assume that at every window execution, the processing delay of P_{avg} (δ_{avg}) remains constant which is 1 time unit. The sampling interval, $\lambda_{in} = 2$ time units which is also constant.

Inference with constant processing delay

The left side of Figure 5.2 shows the case where the basic provenance inference method infers accurate provenance. In this case, the first three tuples t_1, t_3 and t_5 form the original window. Based on our assumptions, the execution over this window takes 1 time unit and thus, the *transaction time* of the output tuple is 6. The output tuple is shown by t_6 in the view V_{out}. If we infer provenance of this tuple based on the basic provenance inference method, 6 is the *reference point* for calculating the *upper bound* of the *inferred window* based on the Equation 4.1 which is given below:

$$\begin{aligned} \mathtt{upperBound} &= \mathtt{referencePoint} - \mathtt{processingDelay} \\ &= 6 - 1 \\ &= 5 \end{aligned}$$

After calculating the *upper bound*, the basic provenance inference method considers only the latest 3 (=$\mathtt{windowSize}$) tuples, satisfying *transaction time* $\leqslant \mathtt{upperBound}$, to form the inferred window. Therefore, the tuple t_1, t_3 and t_5 is included in the inferred window which is the same to the original window.

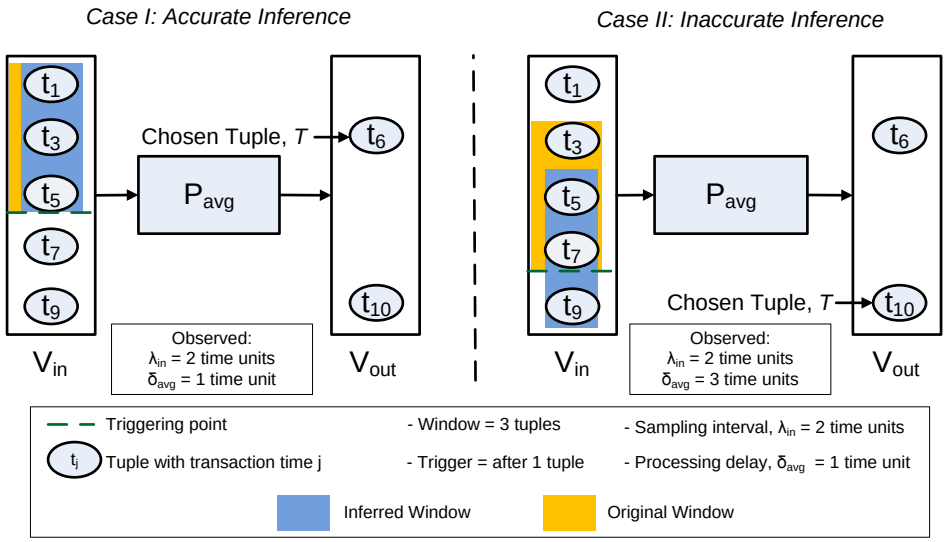


Figure 5.2: Examples of accurate and inaccurate provenance inference in a tuple-based window

Inference with variable processing delay

The right side of Figure 5.2 depicts the other case where the basic provenance inference method infers inaccurate provenance because of the variation in the processing delay of P_{avg}, δ_{avg}. In this case, the assumption of having constant processing delay, i.e., $\delta_{avg} = 1$ time unit, is violated. In the right side of Figure 5.2, the tuple t_3, t_5 and t_7 form the original window which is triggered just after the arrival of t_7. It is possible that the execution over this window could take longer than our assumption of 1 time unit due to some other workload in the system. In this case, the processing delay of P_{avg}, δ_{avg} is 3 time units and thus, the *transaction time* of the output tuple becomes 10. The output tuple is shown by t_{10} in the view V_{out}. If we infer provenance of this tuple based on the basic provenance inference method, 10 is the *reference point* for calculating the *upper bound* of the *inferred window* and 1 is the value of processing delay since the basic provenance inference method always count on the given constant processing delay and does not observe the actual delay during the execution. Based on the Equation 4.1:

$$\begin{aligned} \mathtt{upperBound} &= \mathtt{referencePoint} - \mathtt{processingDelay} \\ &= 10 - 1 \\ &= 9 \end{aligned}$$

After calculating the *upper bound*, the basic provenance inference method retrieves only the latest 3 (=$windowSize$) tuples, satisfying *transaction time* $\leqslant$ $upperBound$, to form the inferred window. Therefore, the tuple t_5, t_7 and t_9 is included in the inferred window which differs from the original window, resulting into inaccurate provenance information. Looking at these examples, we can observe that in the later case, a new tuple t_9 is inserted into the view V_{in} before the completion of the current original window which results into the inaccuracy. From this observation, the following *Failure Condition* is defined.

Failure Condition 5.1 *Inclusion of a non-contributing input tuple, at the ending edge of the inferred window, may occur if that particular non-contributing tuple is inserted into the input view,* V_i*, before completing the execution of a computing processing element,* P_k*, on the current original window defined over the same input view,* V_i*. If the following condition holds, we have a failure:* $\lambda_i \leqslant \delta_k$.

5.3.2 Failure Conditions for Time-based Windows

In case of time-based windows, the basic provenance inference method may infer inaccurate provenance information if there is a variation in both processing delay (δ_k) and sampling interval (λ_i) during the repeated executions of a computing processing element.

Figure 5.3 shows the examples of the execution of a particular computing processing element P_{avg} with a time-based window. P_{avg} has input view V_{in} and after completing the execution, it produces output tuples, inserted into the view V_{out}. We assume that the window size of V_{in}, $WS_{in} = 6$ time units. P_{avg} triggers after every 2 time units which means $TR_{avg} = 2$ time units. Furthermore, we also assume that at every window execution, the processing delay, δ_{avg}, remains constant which is 1 time unit. The sampling interval, $\lambda_{in} = 2$ time units which is also constant.

Inference with constant processing delay

The left side of Figure 5.3 shows the case where the basic provenance inference method infers accurate provenance. In this case, the original window holds tuples with the *transaction time* within the range of $[0, 6)$. Please note that the upper bound of the window is exclusive and the lower bound of the window is inclusive. Based on the given constant processing delay δ_{avg}, the execution over this window takes 1 time unit and thus, the *transaction time* of the output tuple is 7. The output tuple is shown by t_7 in the view V_{out}. If we infer provenance of this tuple based on the basic prove-

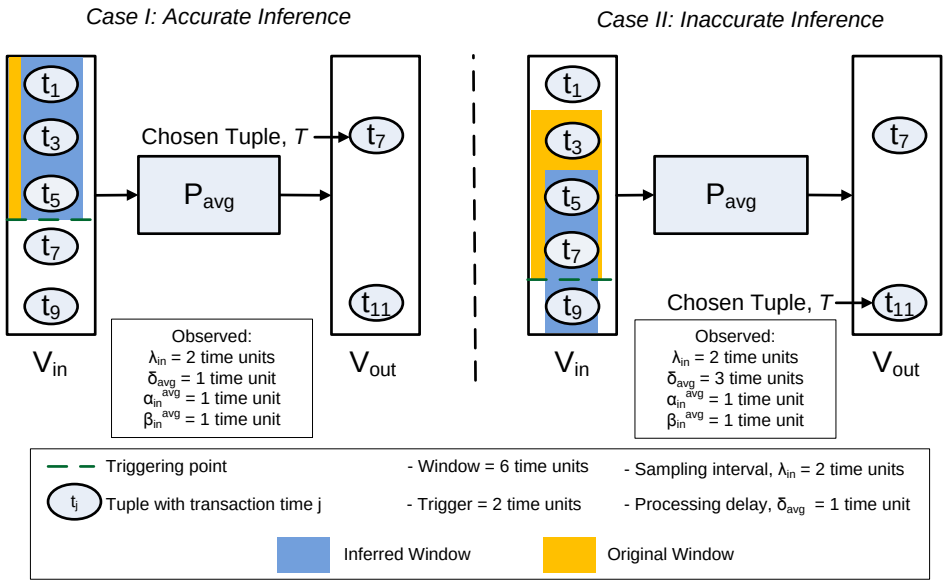


Figure 5.3: Examples of accurate and inaccurate provenance inference in a time-based window

nance inference method, 7 is the *reference point* for calculating the *upper bound* and *lower bound* of the *inferred window* based on the Equation 4.2 and 4.3, respectively, which are given below:

$$\begin{aligned} \mathtt{upperBound} &= \mathtt{referencePoint} - \mathtt{processingDelay} \\ &= 7 - 1 \\ &= 6 \end{aligned}$$

$$\begin{aligned} \mathtt{lowerBound} &= \mathtt{referencePoint} - \mathtt{processingDelay} - \mathtt{windowSize} \\ &= 7 - 1 - 6 \\ &= 0 \end{aligned}$$

After calculating the *upper bound* and *lower bound*, the basic provenance inference method includes the tuples with the *transaction time* within the range of $[0, 6)$ to form the inferred window. In this case, the *inferred window* is the same to the original window and therefore, the basic provenance inference method infers accurate provenance.

The right side of Figure 5.3 depicts the other case where the basic provenance inference method infers inaccurate provenance because of the variation in the processing delay of P_{avg} (δ_{avg}). In this case, the assumption of

having constant processing delay of P_{avg}, i.e., $\delta_{avg} = 1$ time unit, is violated. In the right side of Figure 5.3, the tuples with *transaction time* within the range of $[2, 8)$ are included in the original window which is triggered at time 8. It is possible that the execution over this window could take longer than our assumption of 1 time unit due to some other workload in the system. In this case, the processing delay of P_{avg}, $\delta_{avg} = 3$ time units, and thus, the *transaction time* of the output tuple becomes 11. The output tuple is shown by t_{11} in the view V_{out}. If we infer provenance of this tuple based on the basic provenance inference method, 11 is the *reference point* for calculating the *upper bound* of the *inferred window* and 1 is the value of processing delay since the basic provenance inference method always count on the given constant processing delay and does not observe the actual delay during the execution. Based on the Equation 4.2 and 4.3:

Inference with variable processing delay

$$\begin{aligned} \mathtt{upperBound} &= \mathtt{referencePoint} - \mathtt{processingDelay} \\ &= 11 - 1 \\ &= 10 \end{aligned}$$

$$\begin{aligned} \mathtt{lowerBound} &= \mathtt{referencePoint} - \mathtt{processingDelay} - \mathit{windowSize} \\ &= 11 - 1 - 6 \\ &= 4 \end{aligned}$$

After calculating the *upper bound* and *lower bound*, the basic provenance inference method includes the tuples with *transaction time* within the range of $[4, 10)$ to form the inferred window. In this case, the *inferred window* is different than the original window and therefore, the basic provenance inference method infers inaccurate provenance.

There are two possible reasons of inferring inaccurate provenance information in time-based windows. Firstly, exclusion of a contributing input tuple/data product from the inferred window might cause inaccuracy. Secondly, inclusion of a non-contributing input tuple into the inferred window might also result into wrong provenance information. To explain these reasons further, we need to exploit two variables, ${\alpha_{in}}^{avg}$ and ${\beta_{in}}^{avg}$, as introduced in Section 5.2. ${\alpha_{in}}^{avg}$ refers to the amount of between the original window start and the arrival of the first tuple within that window. In the first case shown in the left side of Figure 5.3, the window starts at time 0 and the first tuple arrives at time 1. Therefore, ${\alpha_{in}}^{avg} = 1$ time unit. In the second case shown in the right side of Figure 5.3, the window starts at

Exclusion of contributing input tuples

time 2 and the first tuple arrives at time 3 which means that ${\alpha_{in}}^{avg} = 1$ time unit. However, the value of δ_{avg} differs in these cases. In the first case, $\delta_{avg} = 1$ time unit while in the second case, the value of δ_{avg} is 3 time units. Since the basic provenance inference method only counts on the given constant value of δ_{avg} to calculate the boundary of the *inferred window*, it could possibly exclude a contributing tuple from the starting edge of the *inferred window* in the second case shown in the right side of Figure 5.3 where ${\alpha_{in}}^{avg}$ is less than the actual processing delay δ_{avg}. Based on this observation, the following *Failure Condition* is defined:

Failure Condition 5.2 *Exclusion of a contributing input tuple from the starting edge of the inferred window may occur if the processing delay δ_k of a computing processing element P_k is longer than the amount of time between the original window starts and the arrival of the first tuple in the original window which is defined over the view V_i, an input view to P_k. If the following condition holds, we have a failure: ${\alpha_i}^k < \delta_k$*

Inclusion of non-contributing input tuples

In time-based windows, it is also possible to include a non-contributing input tuple in the *inferred window* which results into inaccurate provenance information. This could happen if an input tuple arrives before completing the execution over the current window. In a time-based window with a time-based trigger, since the window can trigger after a while once the last tuple within the window has arrived, this amount of time is needed to be considered to decide whether such an inaccuracy can occur or not. In the right side of Figure 5.3, ${\beta_{in}}^{avg}$ refers to the amount of time between the arrival of the last tuple within the window and the trigger of the window which is 1 time unit. If the difference between the sampling interval of the non-contributing input tuple, represented as λ_{in}, and ${\beta_{in}}^{avg}$ is less than the actual processing delay δ_{avg} for that window, the non-contributing tuple t_9 is included in the *inferred window*. Based on this observation, the following *Failure Condition* is defined:

Failure Condition 5.3 *Inclusion of a non-contributing input tuple, at the ending edge of the inferred window, may occur if that particular non-contributing tuple is inserted into the input view, V_i, before completing the execution of a computing processing element, P_k, on the original window defined over the same input view, V_i. If the following condition holds, we have a failure: $\lambda_i - {\beta_i}^k < \delta_k$.*

5.4 OVERVIEW OF THE PROBABILISTIC PROVENANCE INFERENCE

The *probabilistic provenance inference* method is an extension of the *basic provenance inference* method, discussed in Chapter 4. While the general principle remains the same, the probabilistic provenance inference method is capable of handling variation in the processing delay and the sampling interval by facilitating the given distributions of these parameters. The major advantage of the probabilistic provenance inference method is that it can infer comparatively more accurate provenance information than the basic provenance inference method at the same amount of storage consumption.

Like the basic provenance inference method, the *probabilistic provenance inference* method has three phases to infer fine-grained data provenance. These three phases are: i) Documentation of workflow provenance, ii) Backward computation and iii) Forward computation. As already discussed in Section 4.6.1, *documentation of workflow provenance* is a one-time action documenting values of different properties of nodes within a given workflow, performed during the setup of the processing elements. In addition, both processing delay and sampling interval distributions, i.e., $P(\delta_k)$ and $P(\lambda_i)$, respectively, need to be documented in this phase to apply the probabilistic provenance inference method.

Documentation of workflow provenance

The next two phases will be executed only when the scientist is interested to know the fine-grained provenance information of an output data product, i.e., a tuple in the output view. The scientist will select an output data product which seems to have abnormal/unexpected value. After choosing the output data product, the *backward computation* phase is executed. The *backward computation* phase reconstructs the original window, referred to as the *inferred window*. Unlike the basic provenance inference, the probabilistic provenance inference method facilitates the documented $P(\delta_k)$ and $P(\lambda_i)$ distributions alongside the window size defined over the input views to compute the *inferred window* in this phase. The input data products within the inferred window might contribute to produce the selected output data product.

Backward computation

Afterward, the forward computation is executed. Like the basic provenance inference method, the probabilistic provenance inference method establishes the relationship between contributing input data products within the inferred window and the selected output data product by facilitating the given workflow provenance in this phase.

Forward computation

5.5 REQUIRED INFORMATION

Applying the probabilistic provenance inference method, inferring fine-grained data provenance, depends on a few parameters and assumptions, like the basic provenance inference method. Details on this required information and assumptions have already been discussed in Section 4.5. In this section, we present a summary on this required information and assumptions.

First, it is required to attach the *transaction time* (system timestamp) to every data products/tuples. Second, the probabilistic provenance inference method requires the processing elements to process data products/tuples based on their order on *transaction time* in the input view, following *temporal ordering* of tuples during the processing. Moreover, like any other inference-based methods, the probabilistic provenance inference method also considers the documented workflow provenance of a scientific model to infer fine-grained data provenance.

Furthermore, there are a few assumptions required to be fulfilled to apply the probabilistic provenance inference method. These assumptions which are applicable for a computing processing element with multiple input views or producing multiple output data products, have been introduced in Section 4.5.2. One of these assumptions indicates that the inference mechanism must know the *order of input views* which participate in an activity, realized by a computing processing element. Another assumption mentions that the name of the *contributing input view* shall be documented explicitly with the output tuple in cases the activity to be performed has multiple input views. The other assumption also has to be satisfied to ensure that the *order of tuples in the output view* follows the same order found in input views. If these assumptions are not fulfilled by the underlying system, the probabilistic provenance inference method cannot be applied.

5.6 DOCUMENTATION OF WORKFLOW PROVENANCE

The *documentation of workflow provenance* is the pre-requisite phase which has to be completed before the actual execution of the inference-based method. In this phase, the workflow provenance of the entire data processing is explicated. In Section 4.6.1, we have provided a list of properties of a computing processing element which are required to be documented to

apply the basic provenance inference method. This set of properties are also required to be documented to apply the probabilistic provenance inference method. Therefore, we again provide this set of properties in the following.

- *Window type*: refers to a list of window types; one element for each input view. The value can be either *tuple* or *time*.
- *Window size*: refers to a list of window sizes; one element for each input view. The value represents the size of the window.
- *Trigger type*: specifies how a *computing processing element* will be triggered for execution; The value can be either *tuple* or *time*.
- *Trigger interval*: refers to the interval between successive executions of the same computing processing element.
- *Input-output ratio*: refers to the ratio of the number of input data products contributed to produce output data products over the number of output data products of a particular computing processing element.
- *Number of input views*: refers to the total number of input views.
- *Identifier of input views*: refers to the list of ids (node identifiers) of input views.
- *Contributing input views*: refers to the fact that whether a computing processing element with multiple input views processes data products over *all* input views or a *specific* input view at a time. For computing processing elements with only one input view, it is set to *not applicable*.

Additional properties

Unlike the basic provenance inference method, the probabilistic provenance inference method can handle variation in the processing delay of a computing processing element as well as variation in the sampling interval of an input view. Therefore, in addition to the aforesaid properties, the distribution of processing delay of a computing processing element and the distribution of sampling interval of an input view are required to be explicated in the workflow provenance.

- *Processing delay distribution*: refers to the distribution of amount of time required by a computing processing element to complete the

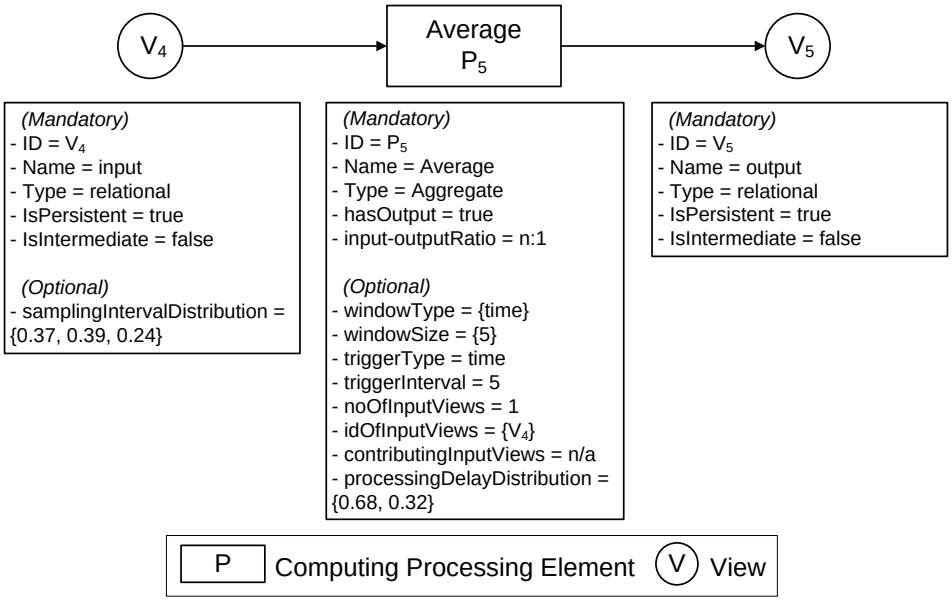


Figure 5.4: Example of the explicated workflow provenance

execution over the defined window. For a computing processing element P_k, the processing delay distribution is denoted as $P(\delta_k)$.

- *Sampling interval distribution*: refers to the distribution of amount of time between two successive tuples insertion into the view V_i, which is an input view of a computing processing element P_k. The sampling interval distribution is denoted as $P(\lambda_i)$.

Example

Figure 5.4 shows the simplified workflow described in Section 5.1 and its explicated workflow provenance. In Figure 5.4, the workflow consists of a computing processing element P_5, implementing an *average* operation. P_5 takes one input view V_4 and produces one output view, V_5. The given sampling interval distribution of the input view V_4, $P(\lambda_4) = \{0.37, 0.39, 0.24\}$ for time unit 1, 2 and 3, respectively. It means that 37% tuples arrive after 1 time unit from the previous tuple arrival and so on. The window size, ${WS_4}^5 = 5$ time units. The processing element, P_5 will be executed after every 5 time units. Therefore, $TR_5 = 5$ time units. The processing delay distribution of P_5, $P(\delta_5) = \{0.68, 0.32\}$ for time unit 1 and 2, respectively. It means that 68% times of the processing over the current window take 1 time unit and so on. These distributions are given by scientists based on the analysis of historical data which can be obtained by observing the

execution of the workflow of a model for at least 100 times. The next two phases of the probabilistic provenance inference method facilitates this documented workflow provenance information, shown in Figure 5.4.

5.7 BACKWARD COMPUTATION OF PROBABILISTIC PROVENANCE INFERENCE

The backward computation phase is executed based on the request initiated by a user to infer fine-grained data provenance of an output data product/tuple T. The backward computation phase reconstructs the original window by facilitating the explicated workflow provenance, shown in Figure 5.4 and the *transaction time* of the tuple T which is referred to as the *reference point*. This reconstructed window is referred to as the *inferred window*. Unlike the basic provenance inference, the probabilistic provenance inference method reconstructs the inferred window based on an *offset* value to handle the variation in processing delay and sampling interval. The *offset* value represents the distance in time between the *reference point* and the *upper bound* of the window. The *offset* value is calculated in such a way that the probabilistic provenance inference method can achieve the optimal accuracy. Therefore, the mechanism to calculate the *offset* value is the center of focus in the backward computation phase.

The mechanism of the backward computation phase is given in Algorithm 5.1. First, the transaction time of the chosen tuple T, i.e., *reference point*, and number of input views are retrieved in line 1 and 2. The *processing delay* distribution of P_k, $P(\delta_k)$ is also retrieved in line 3. Then, for each input view V_i, we retrieve it's *id*, *window type*, *window size* and *sampling interval distribution* ($P(\lambda_i)$) in line 5-8. Afterward, we compute the list of the input tuples I_{C^i} for each input view V_i which form the inferred window. Forming the inferred window depends on the type of the window. If the window is a tuple-based window, line 10-13 are executed. Otherwise, in case of a time-based window, line 15-23 are executed. In both cases, the backward computation phase returns a set of tuples, I_{C_i}, for each input view V_i. These tuples form the corresponding inferred window which is used by the forward computation phase to infer exact relationship between input data products/tuple and the chosen output data product/tuple. The details of the mechanism of reconstructing the original window for both tuple-based and time-based windows are explained in next sections.

Algorithm

Algorithm 5.1: Backward Computation of Probabilistic Provenance Inference

Input: A tuple T produced by a processing element P_k, for which fine-grained provenance is requested

Output: Set of input tuples I_{C^i} for each input view V_i which form the inferred window producing T

```
1  referencePoint ← getTransactionTime(T);
2  noOfInputViews ← getNoOfInputViews(P_k);
3  processingDelayDist ← getProcessingDelayDist(P_k);
4  for i ← 1 to noOfInputViews do
5      inputView ← getInputViewID(P_k, i);
6      windowType ← getWindowType(inputView);
7      windowSize ← getWindowSize(inputView);
8      samplingIntervalDist ←
       getSamplingIntervalDist(inputView);
9      if windowType = "tuple" then      /* tuple-based windows */
10         offset ← calculateOffset(samplingIntervalDist,
11                          processingDelayDist);
12         I_Ci ← reconstructTupleWindow(inputView, windowSize,
13                          referencePoint, offset);
14     else                                /* time-based windows */
15         triggerInterval ← getTriggerInterval(P_k);
16         alphaDist
           ← calculateAlphaDist(windowSize, triggerInterval,
17                          samplingIntervalDist);
18         betaDist
           ← calculateBetaDist(windowSize, triggerInterval,
19                          samplingIntervalDist);
20         offset
           ← calculateOffset(samplingIntervalDist, alphaDist
21                      betaDist, processingDelayDist);
22         I_Ci ← reconstructTimeWindow(inputView, windowSize,
23                          referencePoint, offset);
24     end
25 end
```

5.7.1 Forming the Inferred Window in Tuple-based Windows

Forming the inferred window requires to calculate the *offset* value based on the given $P(\delta_k)$ and $P(\lambda_i)$ distributions as mentioned in line 10 of Algorithm 5.1. Afterward, the calculated *offset* value is used to reconstruct the original window, i.e., the inferred window, containing a set of tuples, I_{C_i}, for each input view V_i (line 12 in Algorithm 5.1). The *upper bound* of the inferred window in case of a tuple-based window is calculated based on the following equation.

$$\mathtt{upperBound} = \mathtt{referencePoint} - \mathtt{offset} \tag{5.1}$$

Please note that, the *upper bound* of the inferred window refers to a time value. After calculating the *upper bound*, `reconstructTupleWindow` function considers only the latest $\mathtt{windowSize}$ number of tuples whose transaction time is less than or equal to the $\mathtt{upperBound}$ to form the inferred window.

Offset calculation

As already defined, the *offset* value refers to the distance in time between the *upper bound* of the inferred window and the *reference point*. Therefore, based on the definition, $0 \leqslant \mathit{offset} \leqslant max(\delta_k)$, where $max(\delta_k)$ refers to the maximum value of the random variable δ_k. The *offset* value has to be calculated in such a way that in most cases, the resulting inferred window contains all the potential input tuples which contributed to produce the chosen output tuple. The calculation of *offset* value involves the joint probability distribution of $P(\lambda_i)$ and $P(\delta_k)$ as well as the *Failure Condition* 5.1 which indicates that wrong provenance can be inferred by the inclusion of a non-contributing input tuple in the inferred window, i.e., if $\lambda_i \leqslant \delta_k$ holds. The following equation calculates the *offset* value.

$$\begin{aligned}
\mathtt{offset} &= \arg\max_{\mathtt{offset}} f(\mathtt{offset}) \\
&= \arg\max_{\mathtt{offset}} \{ \sum_{x=min(\lambda_i)}^{max(\lambda_i)} \sum_{y=min(\delta_k)}^{max(\delta_k)} \{P(\lambda_i = x,\ \delta_k = y - \mathtt{offset}) \\
&\quad \mid x > y - \mathtt{offset} \ \wedge\ y - \mathtt{offset} \geqslant 0 \ \} \ \}
\end{aligned} \tag{5.2}$$

Based on Equation 5.2, the *offset* value is the value for which $f(\mathtt{offset})$ has the maximum value. The value of $f(\mathtt{offset})$ represents the estimated

Table 5.1: Joint Probability Distribution of given $P(\lambda_i)$ and $P(\delta_k)$

$\lambda_i = x$	$\delta_k = y$	$P(\lambda_i = x, \delta_k = y)$
1	1	0.252 (a)
1	2	0.118 (b)
2	1	0.265 (c)
2	2	0.125 (d)
3	1	0.163 (e)
3	2	0.077 (f)

accuracy based on the given *offset*, joint probability distribution of $P(\lambda_i)$ and $P(\delta_k)$ as well as Failure Condition 5.1.

Table 5.1 shows the joint probability distribution of given $P(\lambda_i)$ and $P(\delta_k)$. In this case, we assume that $P(\lambda_i)$, has the following values: $P(\lambda_i = 1) = 0.37$, $P(\lambda_i = 1) = 0.39$, $P(\lambda_i = 1) = 0.24$ with $mean(\lambda_i) = 2$. Moreover, we also assume that $P(\delta_k)$, has the following values: $P(\delta_k = 1) = 0.68$, $P(\delta_k = 2) = 0.32$ with $mean(\delta_k) = 1$. Next, the value of $P(\lambda_i = x, \delta_k = y)$ is calculated which is shown in the right-most column of Table 5.1.

Example

First, we set *offset* value to 0 which means that the *reference point* and the *upper bound* of the inferred window represent the same point in time. In this case, based on *Failure Condition* 5.1, accurate provenance information can be inferred in 51% (c+e+f) cases only. However, if the value of *offset* is set to 1, the chance of inferring accurate provenance can be improved. Since *offset* refers to the distance between the *upper bound* of the inferred window and the *reference point*, setting the value of *offset* to 1 means that the value of δ_k is also subtracted by 1. According to the *Failure Condition* 5.1 and Table 5.1, the chance of accurately inferred provenance is increased to 88% (a+c+d+e+f). We cannot set *offset* value to 2 because it violates one of the conditions, i.e., $y - \mathtt{offset} \geqslant 0$, of Equation 5.2. In this example, setting *offset* to 1 returns the maximum value of $f(\mathtt{offset})$ which is the estimated accuracy of applying the probabilistic provenance inference method. Therefore, the *offset* value is set to 1 in Equation 5.1 calculating the *upper bound* of the inferred window.

5.7.2 Forming the Inferred Window in Time-based Windows

In case of a time-based window, the backward computation phase reconstructs the original window, i.e., forming the inferred window, by calculating the *upper bound* and the *lower bound* of the window based on the following equations.

$$\mathtt{upperBound} = \mathtt{referencePoint} - \mathtt{offset} \tag{5.3}$$

$$\mathtt{lowerBound} = \mathtt{referencePoint} - \mathtt{offset} - \mathtt{windowSize} \tag{5.4}$$

The probabilistic provenance inference method calculates the *offset* value in such a way that the inference-based method could infer accurate provenance in most cases. There are two possible ways to have a failure, i.e., inaccurate provenance, during the inference mechanism. First, a failure could occur if a contributing input tuple is excluded from the inferred window. According to *Failure Condition* 5.2, this could happen if the following holds: ${\alpha_i}^k < \delta_k$. Second, a failure could occur if a non-contributing input tuple is included into the inferred window. According to *Failure Condition* 5.3, this could happen if the following holds: $\lambda_i - {\beta_i}^k < \delta_k$. Since both ${\alpha_i}^k$ and ${\beta_i}^k$ values participate to estimate the accuracy of the inferred provenance information, it is necessary to compute these distributions to calculate the *offset* value.

We propose a novel *tuple-state graph*, that has to be constructed by facilitating Markov chain model [16], to compute both $P({\alpha_i}^k)$ and $P({\beta_i}^k)$ distributions which help us to infer fine-grained data provenance. A *tuple-state graph* represents the transitions from one state to another, where state is referring to the position of the incoming input tuple within a window, based on the sampling interval λ_i of the input view V_i.

To build the *tuple-state graph*, we consider the workflow provenance explicated in Figure 5.4. Figure 5.4 shows that the sampling interval distribution of input view V_4, $P(\lambda_4)$ has the following values: $P(\lambda_4 = 1) = 0.37$, $P(\lambda_4 = 2) = 0.39$ and $P(\lambda_4 = 3) = 0.24$ where $mean(\lambda_4) = 2$ time units. Figure 5.4 also shows that the processing delay distribution of P_5, $P(\delta_5)$ has the following values: $P(\delta_5 = 1) = 0.68$, $P(\delta_5 = 2) = 0.32$. In the next section, we discuss the mechanism to build the *tuple-state graph*, calculating the corresponding $P({\alpha_4}^5)$ and $P({\beta_4}^5)$ distributions using the given $P(\lambda_4)$, $P(\delta_5)$ distributions.

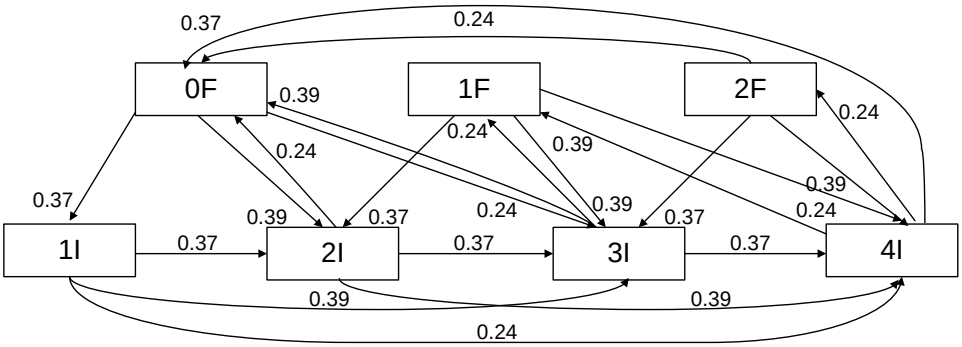


Figure 5.5: Tuple-state graph G_α

5.7.2.1 *Building Tuple-State Graph Calculating* $P({\alpha_4}^5)$ *Distribution*

A *tuple-state graph*, G_α, has to be built to compute the corresponding ${\alpha_4}^5$ distribution by facilitating the given $P(\lambda_4)$ and $P(\delta_5)$ distributions. Each vertex in a *tuple-state graph* represents a state, which identifies the position of a tuple within a window w.r.t. the start of the window. There are two different types of states in a *tuple-state* graph calculating $P({\alpha_4}^5)$ distribution. These are:

1. *First states*: These states represent that the current tuple is the first tuple of a particular window. These are denoted as the arrival timestamp of the tuple in the window w.r.t the start of the window, followed by a letter 'F' (e.g. 0F, 1F, 2F). In this case, the arrival timestamps indicate the *first-tuple appearance interval* as discussed in Section 5.2.

2. *Intermediate states*: These states represent the arrival of tuples within a window without being the first tuple. The states are represented by the arrival timestamp of the new tuple in the window w.r.t the start of the window, followed by a letter 'I' (e.g. 1I, 2I, 3I, 4I).

Set of vertices

The *tuple-state graph* G_α has a set of vertices and directed edges, i.e., $G_\alpha = (V_\alpha, E_\alpha)$. Therefore, first, a set of *first* and *intermediate* states are added as vertices/nodes. The number of vertices in both states is bounded by the window size. It can be expressed by the following formula.

$$V_\alpha = \bigcup_{j=0}^{ws_4{}^5-1} \{jF, jI\} \tag{5.5}$$

where $WS_4{}^5$ be the *window size* of the window defined over the input view V_4.

Next, we add directed edges from all vertices based on the values of $P(\lambda_4)$. A directed edge is defined via the start vertex (*from vertex*), the end vertex (*to vertex*), and the probability of this edge occurring (*weight*).

Set of directed edges

A directed edge can connect two states within the same window or it can cross the window boundary. Therefore, the set of directed edges is a collection of two other sets. The first set defines the directed edges from every point (state) in the window to a later point (state) in the window without crossing the window boundary. In this case, the start vertex could be a *first* or an *intermediate* state, while the end vertex must be an *intermediate* state. Assume that, TR_5 be the *trigger interval* of P_5, the formula below represents these directed edges, where the weight associated to a directed edge corresponds to the probability of two subsequent tuples arriving with a distance of $k-j$ time units.

$$E_\alpha{}^1 = \bigcup_{j=0}^{WS_4{}^5-1} \bigcup_{k=j+1}^{j+max(\lambda_4)} \{\{(jF, kI, P(\lambda_4 = k-j)),$$
$$(jI, kI, P(\lambda_4 = k-j))\} \mid k < WS_4{}^5 \wedge k < TR_5\} \tag{5.6}$$

The second set defines directed edges which are crossing the window boundary. In this case, the start vertex is either a first or an intermediate state, while the end vertex is a first state. The formula below represent these directed edges.

$$E_\alpha{}^2 = \bigcup_{j=0}^{WS_4{}^5-1} \bigcup_{k=j+1}^{j+max(\lambda_4)} \{\{(jF, k'F, P(\lambda_4 = k-j)),$$
$$(jI, k'F, P(\lambda_4 = k-j))\} \mid k \geqslant TR_5 \wedge k' = k \mod TR_5\} \tag{5.7}$$

The complete set of directed edges in G_α is the union of $E_\alpha{}^1$ and $E_\alpha{}^2$.

$$E_\alpha = E_\alpha{}^1 \cup E_\alpha{}^2$$

Example

Figure 5.5 depicts the simplified *tuple-state graph* G_α (vertices with no incoming edges are discarded). Given, $P(\lambda_4 = 1) = 0.37$, $P(\lambda_4 = 2) = 0.39$ and $P(\lambda_4 = 3) = 0.24$. Starting from the vertex 'oF', directed edges are added to '1I', '2I' and '3I' with weight 0.37, 0.39 and 0.24, respectively. These directed edges are the elements of set $E_\alpha{}^1$. As another example,

Table 5.2: Observed vs. Computed $P(\alpha_4{}^5)$ Distribution

$\alpha_4{}^5 = u$	Observed $P(\alpha_4{}^5 = u)$	Computed $P(\alpha_4{}^5 = u)$
0	0.532	0.535
1	0.347	0.337
2	0.121	0.128

starting from vertex '4I', we add directed edges to '0F', '1F' and '2F' with weight 0.37, 0.39 and 0.24, respectively. These directed edges are elements of set $E_\alpha{}^2$. This process will be continued for all vertices to get a complete G_α.

Steady-state

The long-term behavior of a Markov chain enters a steady state, i.e., the probability of being in a state will not change with time [47]. In the steady state, the vector s_α represents the average probability of being in a particular state based on the *tuple-state graph* G_α. To optimize the steady state calculation, vertices with no incoming edges are discarded (see Figure 5.5).

Assuming uniformly distributed initial probabilities, the steady state of the Markov model can be derived. The steady state distribution vector s_α for the *tuple-state graph*, G_α is given below.

$$s_\alpha = \left(\begin{array}{ccccccc} 0F & 1F & 2F & 1I & 2I & 3I & 4I \\ \hline 0.20 & 0.13 & 0.05 & 0.07 & 0.15 & 0.20 & 0.20 \end{array} \right)$$

The probabilities of states with suffix 'F' form the $\alpha_4{}^5$ distribution, i.e., the probabilities of the first tuple in a window arriving after a specific time interval. The components of the states '0F', '1F', '2F' represent the probability of the value of $\alpha_4{}^5$ = 0, 1 and 2, respectively. After normalizing the probability of these values, we get the computed $P(\alpha_4{}^5)$ distribution, shown in Table 5.2.

5.7.2.2 *Building Tuple-state Graph Calculating $P(\beta_4{}^5)$ Distribution*

Along the lines of the previous section, $P(\beta_4{}^5)$ distribution indicating the probability distribution on the distance between the last tuple in a window and the end of the window can be calculated.

For the computing processing element P_5 shown in Figure 5.4, a *tuple-state graph*, G_β is built to compute the corresponding $P({\beta_4}^5)$ distribution by facilitating the given $P(\lambda_4)$ and $P(\delta_5)$ distributions. As already mentioned in Section 5.7.2.1, each vertex in a *tuple-state graph* represents a state, which identifies the position of a tuple within a processing window w.r.t. the start of the window. However, unlike G_α, the *tuple-state graph* G_β has different types of states listed below.

1. *Intermediate states*: These states represent the arrival of tuples within a window without being the last tuple of that window. The states are represented by the arrival timestamp of the new tuple in the window w.r.t the start of the window, followed by a letter 'I' (e.g. 0I, 1I, 2I, 3I, 4I).

2. *Last states*: These states represent that the current tuple is the last tuple of a particular window. These are denoted as the arrival timestamp of the tuple in the window w.r.t the start of the window, followed by a letter 'L' (e.g. 2L, 3L, 4L). In this case, the arrival timestamps indicate the *last-tuple disappearance interval* as discussed in Section 5.2.

Set of vertices

The *tuple-state graph* G_β has a set of vertices and directed edges, i.e., $G_\beta = (V_\beta, E_\beta)$. Therefore, first, a set of *intermediate* and *last* states are added as vertices/nodes. The number of vertices in both states is bounded by the window size. It can be expressed by the following formula.

$$V_\beta = \bigcup_{j=0}^{{WS_4}^5 - 1} \{jL, jI\} \tag{5.8}$$

where ${WS_4}^5$ be the *window size* of the window defined over the input view V_4.

Set of directed edges

Next, we add directed edges between the vertices based on the values of $P(\lambda_4)$ distribution. There could be three different sets of directed edges in G_β. The first set includes directed edges connecting two points (states) within the same window. The second set includes directed edges representing that the current tuple is the last tuple in the window. The third set includes directed edges representing the transitions crossing the window boundary.

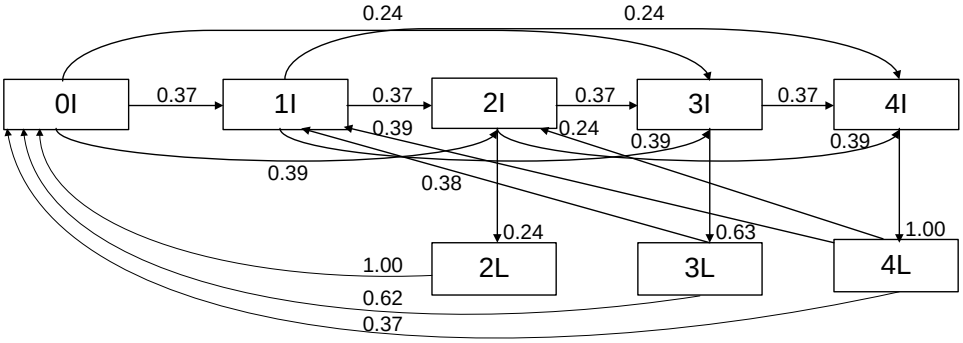


Figure 5.6: Tuple-state graph G_β

The first set defines directed edges, starting from every point (state) in the window to a later point (state) in the window without crossing the window boundary. In this case, both start and end vertex must be *intermediate* states. Assume that, TR_5 be the *trigger interval* of P_5, the formula below represents these directed edges, where the weight associated to an edge corresponds to the probability of two subsequent tuples arriving with a distance of $k-j$ time units.

$$E_\beta{}^1 = \bigcup_{j=0}^{{WS_4}^5-1} \bigcup_{k=j+1}^{j+max(\lambda_4)} \{\{(\,jI, kI, P(\lambda_4 = k-j)\,)\} \mid k < {WS_4}^5 \wedge k < TR_5\} \tag{5.9}$$

The second set defines directed edges, from an *intermediate* state to a *last* state, indicating that the current tuple could be the last tuple within the window. In this case, the arrival timestamps of both *intermediate* and *last* state remains the same. As an example, this set includes a directed edge from '3I' to '3L' representing that the last tuple within this window arrives at timestamp 3. The *weight* of this directed edge is the sum of the probability of crossing the window boundary from the *last* state ('3L') based on the values of $P(\lambda_4)$. The formula below represents this set of directed edges.

$$E_\beta{}^2 = \bigcup_{j={WS_4}^5-max(\lambda_4)}^{{WS_4}^5-1} \{\{(jI, jL, \sum_{k=j+1}^{j+max(\lambda_4)} P(\lambda_4 = k-j))\} \mid k \geqslant TR_5\} \tag{5.10}$$

The third set defines the directed edges that cross the window boundary, i.e., transition from a *last* state to an *intermediate* state based on the values of $P(\lambda_4)$. Since there could be multiple directed edges originating from a particular *last* state and connecting towards multiple *intermediate* states (e.g. from '3L' to 'oI' and '1I'), the *weight* of a particular directed edge is normalized w.r.t. the sum of the probability of directed edges originating from the same *last* state. The formula representing these directed edges is given below:

$$E_\beta{}^3 = \bigcup_{j=0}^{WS_4{}^5-1} \bigcup_{k=j+1}^{j+max(\lambda_4)} \{\{ (jL, k'I, \frac{P(\lambda_4 = k-j)}{\sum_{k=j+1}^{j+max(\lambda_4)} P(\lambda_4 = k-j)}) \mid k \geqslant TR_5 \ \wedge \ k' = k \mod TR_5\}\} \quad (5.11)$$

The complete set of edges in G_β is:

$$E_\beta = E_\beta{}^1 \cup E_\beta{}^2 \cup E_\beta{}^3$$

Figure 5.6 depicts the simplified *tuple-state graph* G_β (vertices with no incoming edges are discarded). Given, $P(\lambda_4 = 1) = 0.37$, $P(\lambda_4 = 2) = 0.39$ and $P(\lambda_4 = 3) = 0.24$. Starting from the vertex 'oI', directed edges are added to '1I', '2I' and '3I' with weight 0.37, 0.39 and 0.24, respectively. These directed edges are the elements of set $E_\beta{}^1$. As another example, consider the vertex '3I'. There is a directed edge from '3I' to '3L', indicating that this tuple is the last tuple within the window. The *weight* of this edge is the sum of the probability of having cross-boundary transition from state '3L' which is $P(\lambda_4 = 2) + P(\lambda_4 = 3) = 0.39 + 0.24 = 0.63$. This directed edge is an element to the set $E_\beta{}^2$. Directed edges are also added from the vertex '3L', connecting towards 'oI' and '1I' with weight $0.39 \div 0.63 = 0.62$ and $0.24 \div 0.63 = 0.38$, respectively. These directed edges are elements of the set $E_\beta{}^3$. This is how, all other edges based on the aforesaid formulas are added to have a complete G_β. *Example*

As already discussed in Section 5.7.2.1, the long-term behavior of a Markov chain enters a steady state. In the steady state, the vector s_β represents the average probability of being in a particular state based on the graph G_β. To optimize the steady state calculation, vertices with no incoming edges are discarded (see Figure 5.6). Assuming uniformly distributed initial probabilities, the steady state of the Markov model can be derived. *Steady-state*

Table 5.3: Observed vs. Computed $P({\beta_4}^5)$ Distribution

${\beta_4}^5 = v$	Observed $P({\beta_4}^5 = v)$	Computed $P({\beta_4}^5 = v)$
1	0.520	0.535
2	0.346	0.337
3	0.134	0.128

The steady state distribution vector s_β for the *tuple-state graph*, G_β is given below.

$$s_\beta = \left(\begin{array}{cccccccc} 0I & 1I & 2I & 3I & 4I & 2L & 3L & 4L \\ \hline 0.146 & 0.146 & 0.146 & 0.146 & 0.146 & 0.035 & 0.092 & 0.146 \end{array} \right)$$

The probabilities of states with suffix 'L' form the ${\beta_4}^5$ distribution, i.e., the probabilities of the last tuple in a window arriving after a specific time interval w.r.t the start of the window. For each last state, the corresponding value of ${\beta_4}^5$ can be calculated based on the following formula.

$${\beta_4}^5 = {WS_4}^5 - \textit{ arrival timestamp of the state}$$

Suppose, the last state is '2L'. Therefore, the value of the corresponding ${\beta_4}^5$ is $5 - 2 = 3$ time units. Based on this, the probabilities of the states '2L', '3L', '4L' represent the probability of the value of ${\beta_4}^5$ = 3, 2 and 1, respectively. After normalizing the probability of these values, we get the computed $P({\beta_4}^5)$ distribution which is shown in Table 5.3.

Both $P({\alpha_4}^5)$ and $P({\beta_4}^5)$ distributions are used to calculate the *offset* value. In next section, we explain the process of calculating *offset* value.

5.7.2.3 *Calculating Offset*

As defined in Section 5.7, the *offset* value refers to the distance in time between the *upper bound* of the inferred window and the *reference point*. Therefore, based on the definition, $0 \leqslant \textit{offset} \leqslant max(\delta_k)$, where $max(\delta_k)$ refers to the maximum value of the random processing delay of P_k. The *offset* value has to be calculated in such a way that in most cases, the resulting inferred window contains all the potential input tuples which contributed to produce the chosen output tuple. In case of a time-based window, the *offset* value is calculated by facilitating the joint probability distribution

of $P({\alpha_i}^k)$, $P({\beta_i}^k)$, $P(\lambda_i)$ and $P(\delta_k)$ based on *Failure Condition* 5.2 and 5.3 indicating situations where wrong provenance can be inferred. *Failure Condition* 5.2 indicates that a contributing input tuple could be excluded from the inferred window if the following condition holds: ${\alpha_i}^k < \delta_k$. *Failure Condition* 5.3 indicates that a non-contributing input tuple could be included into the inferred window if the following condition holds: $\lambda_i - {\beta_i}^k < \delta_k$. Considering these failure conditions, the following equation calculates the *offset* value for a time-based window.

$$
\begin{aligned}
offset &= \arg\max_{offset} f(offset) \\
&= \arg\max_{offset}\{ \sum_{u=min(\alpha_i{}^k)}^{max(\alpha_i{}^k)} \sum_{v=min(\beta_i{}^k)}^{max(\beta_i{}^k)} \sum_{x=min(\lambda_i)}^{max(\lambda_i)} \sum_{y=min(\delta_k)}^{max(\delta_k)} \\
&\{P({\alpha_i}^k = u, \delta_k = y - offset) \times P({\beta_i}^k = v, \lambda_i = x, \delta_k = y - offset) \\
&\mid u \geqslant y - offset \ \wedge \ x - v \geqslant y - offset \ \wedge \ y - offset \geqslant 0\}\}
\end{aligned}
\tag{5.12}
$$

Based on Equation 5.12, the *offset* value is the value for which $f(offset)$ has the maximum value. The value of $f(offset)$ represents the estimated accuracy based on the given *offset*, joint probability distribution of $P({\alpha_i}^k)$, $P({\beta_i}^k)$, $P(\lambda_i)$ and $P(\delta_k)$ as well as *Failure Condition* 5.2 and 5.3.

Figure 5.4 shows an example workflow based on a time-based window defined over the input view V_4. It also shows the given $P(\lambda_i)$ and $P(\delta_k)$ distributions. In previous sections, we have computed $P({\alpha_i}^k)$ and $P({\beta_i}^k)$ distributions based on these given $P(\lambda_i)$ and $P(\delta_k)$ distributions. To calculate the *offset* value for the given workflow shown in Figure 5.4, we use the aforesaid distributions.

First, we set *offset* value to 0 which means that the *reference point* and the *upper bound* of the inferred window represent the same point in time. In this case, based on Equation 5.12, estimated accuracy of inferred provenance information is around 30%. However, if the value of *offset* is set to 1, the chance of inferring accurate provenance is improved. Since *offset* refers to the distance between the *upper bound* of the inferred window and the *reference point*, setting the value of *offset* to 1 means that the value of δ_k is also subtracted by 1. According to Equation 5.12, the chance of accurately inferred provenance is increased to 86%. We cannot set *offset* value to 2 because it violates one of the conditions, i.e., $y - offset \geqslant 0$, of Equation 5.12.

Results

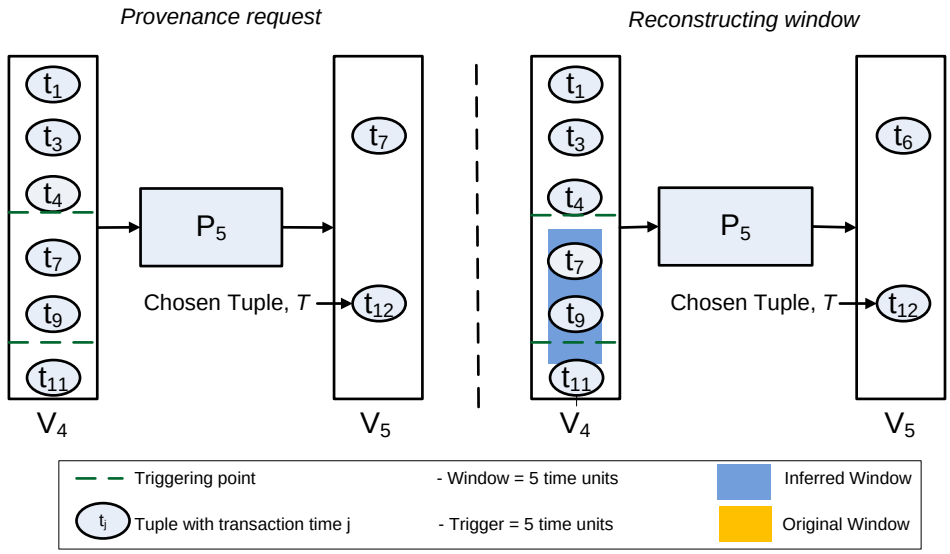


Figure 5.7: Illustration of the backward computation phase

In this example, setting *offset* to 1 returns the maximum value of f(offset) which is the estimated accuracy of applying the probabilistic provenance inference method. Therefore, the *offset* value is set to 1 in Equation 5.3 and 5.4, calculating the *upper bound* and the *lower bound* of the inferred window.

Example

Figure 5.7 depicts the working mechanism of the backward computation phase in the probabilistic provenance inference method. The left-side of Figure 5.7 shows the available data products/tuples in both input and output view, i.e., V_4 and V_5, respectively. The user chooses a tuple T with *transaction time* 12 from the output view V_5, for which he/she initiates the request to infer fine-grained data provenance. The tuple T is also referred to as the *chosen tuple* and the *transaction time* of the *chosen tuple* is referred to as the *reference point*.

The right-side of Figure 5.7 shows the *inferred window* based on Algorithm 5.1 for the workflow shown in Figure 5.4. In this workflow, the window is time-based. Therefore, according to Equation 5.3 and 5.4 and setting the value of *offset* to 1, both the *upper bound* and the *lower bound* of the inferred window is calculated. The inferred window contains tuples from the input view V_4 whose *transaction time* is within the range $[6, 11)$. Therefore, the *inferred window* holds the tuples t_7 and t_9 in view V_4. These tuples are enclosed within the light-blue shaded rectangle over V_4 in Figure 5.7. This set of tuples is considered in the forward computation phase

which associates the exact set of contributing input tuples to the selected output tuple.

5.8 FORWARD COMPUTATION OF PROBABILISTIC PROVENANCE INFERENCE

The *forward computation* phase establishes the data-dependent relationship between the input data products/tuples within the inferred window and the chosen output data product/tuple. This data-dependent relationship is referred to as fine-grained data provenance. The forward computation phase of the probabilistic provenance inference method works exactly in the same way the forward computation phase of the basic provenance inference method does. Therefore, the algorithm of forward computation phase (see Algorithm 4.2) is not repeated in this section. Nevertheless, we explain the mechanism involved with forward computation phase briefly.

First, relevant information about a computing processing element like *number of contributing input tuples*, *number of produced output tuples*, *contributing input views*, *number of input views* is retrieved from the explicated workflow provenance as shown in Figure 5.4.

Next, the forward computation phase infers the exact set of contributing input tuples depending on the type of the computing processing element and retrieved information. Suppose, for computing processing elements implementing operations where only one input tuple contributes to the output tuple such as a *project* or an *union* operation, we have to identify the relevant contributing input tuple from the appropriate input view. To accomplish that, we need to facilitate the assumptions on the order of input views, the contributing input views and order of tuples in the output view, discussed in Section 5.5, to determine the position of the chosen tuple in the output view. Based on the position of the chosen tuple and the assumption on the order of tuples in the output views, forward computation phase can find the exact input tuple which contributed to produce the selected output tuple. In cases where all input tuples contribute to the output tuple, all tuples accessible from the inferred window are selected. Therefore, the set of contributing tuples is the union of the set of input tuples within the inferred window per input view.

Example

Figure 5.8 depicts the forward computation phase. In this example, since the computing processing element P_5, shown in Figure 5.4, performs an *av-*

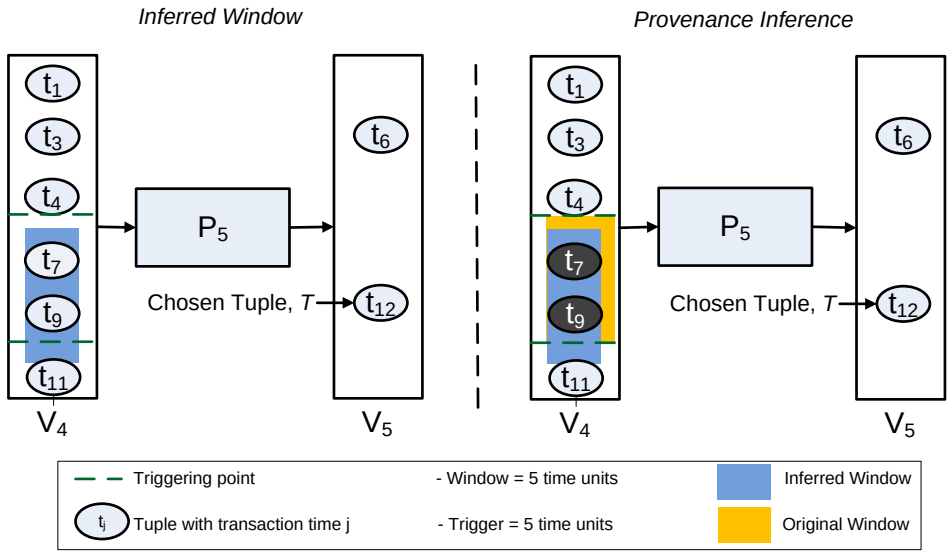


Figure 5.8: Illustration of the forward computation phase

erage operation, the *input-output ratio* is $n : 1$, i.e., all tuples in the inferred window contributed to produce the output tuple. Therefore, forward computation phase concludes that the tuples t_7 and t_9 in the input view V_4 contributed to produce the chosen tuple t_12 in the output view V_5. These contributing input tuples are represented by the shaded tuples within the inferred window in the right side of Figure 5.8. In this example, we can see that the probabilistic provenance inference method can infer accurate fine-grained data provenance.

5.9 EVALUATION

The *probabilistic provenance inference* method is evaluated based on the workflow presented in Section 5.1. The shaded part in the Figure 5.1 is considered for this evaluation. The *probabilistic provenance inference* method infers fine-grained data provenance by facilitating the associated workflow provenance as shown in Figure 5.4. In this figure, the computing processing element P_5 is implementing an *average* operation. The view, V_4, is the input view of P_5 and the view, V_5 is the output view produced by P_5. The collection of tuples in both input and output view is referred to as *sensor data*. In this section, we describe the evaluation criteria, methods, dataset, test cases and also discuss the evaluation results.

5.9.1 Evaluation Criteria and Methods

Criteria

The *probabilistic provenance inference* method is evaluated based on these criteria: i) storage consumption and ii) accuracy. The second research question (*RQ 2*) of this thesis is about the challenge of managing fine-grained data provenance under different system dynamics at reduced cost in terms of storage consumption as discussed in Section 1.4. The probabilistic provenance inference method infers fine-grained data provenance at reduced storage cost under variable processing delay and variable sampling interval. Therefore, the probabilistic provenance inference method addresses *RQ 2* and provides a solution. Since the primary goal of *RQ 2* is to have fine-grained data provenance at reduced storage costs, one of the evaluation criterion is the *storage consumption* by provenance data. Furthermore, the overall goal of the inference-based framework is to provide accurate provenance information under a given system dynamics. Therefore, another evaluation criterion is the *accuracy* of the inferred provenance information.

Methods

As discussed in Section 4.7.1, we developed two implementations of documenting fine-grained data provenance explicitly. These are: i) *explicit provenance* and ii) *improved explicit provenance* method. The storage consumption of the *probabilistic provenance inference* method to maintain provenance data is compared with *explicit provenance*, *improved explicit provenance* and *basic provenance inference* method, discussed in Chapter 4. It may be noted that since both *basic provenance inference* and *probabilistic provenance inference* method are only required to store the *transaction time* for all tuples in *sensor data*, i.e., tuples in both input and output view, to infer fine-grained data provenance, they have the same storage consumption.

The *probabilistic provenance inference* method is also evaluated in terms of accuracy. The accuracy of the inferred provenance information is measured by facilitating the traditional fine-grained provenance information, also known as *explicit provenance*, as a ground truth. We also compare the accuracy between the *basic provenance inference* and the *probabilistic provenance inference* method under variable processing delay and variable sampling interval.

5.9.2 Dataset

A real dataset[3] measuring electrical conductivity of the water, collected by the RECORD project, discussed in Section 5.1, is used to evaluate the performance of different methods. The experiments are performed on a underlying PostgreSQL 8.4[4] database and the Sensor Data Web[5] platform. The input dataset contains 30000 tuples representing a six-month period from July-December 2009 and requires 7200 KB of storage space.

Besides this real dataset, a simulation using artificial data with variable processing delay and variable sampling interval is also performed to compare the achieved accuracy of inference-based methods such as *basic provenance inference* and *probabilistic provenance inference*. Since *probabilistic provenance inference* method can also estimate the accuracy beforehand, the estimated accuracy of the *probabilistic provenance inference* method is also reported in results.

5.9.3 Test cases

The evaluation of the *probabilistic provenance inference* method is performed based on two sets of test cases. The first set of test cases use a real dataset, containing 30000 tuples, as discussed in Section 5.9.2. These test cases are used to compare different methods in terms of storage consumption and accuracy. All these test cases are based on the sliding windows. However, there is a variation in the type and size of the windows as well as the amount of slide. Table 5.4 shows the window size, trigger interval along with some other parameters associated with sampling interval and processing delay of each test case. These test cases with constant processing delay and constant sampling interval are also used to evaluate the *basic provenance inference* method, discussed in Chapter 4. The test cases in Table 5.4, have examples of both tuple-based windows (test case *S1.Tuple.1* and *S1.Tuple.2*) and time-based windows (test case *S1.Time.3* and *S1.Time.4*). Furthermore, some test cases have overlapping windows (test case *S1.Tuple.1* and *S1.Time.3*) and the others have non-overlapping windows (test case *S1.Tuple.2* and *S1.Time.4*). For all test cases shown in Table 5.4, we assume that both sampling interval distribution, i.e., $P(\lambda_4)$ and pro-

Test case set I

3 Available at `http://data.permasense.ch/topology.html#topology`

4 Available at `http://www.postgresql.org/`

5 Available at `http://sourceforge.net/projects/sensordataweb/`

Table 5.4: Test Case Set I : Parameters of Different Test Cases used for the Evaluation using Real Dataset

Test case id	Window size	Trigger interval	mean (λ_4)	max (λ_4)	mean (δ_5)	max (δ_5)
S1.Tuple.1	3	1	2 s	3 s	1 s	2 s
S1.Tuple.2	3	3	2 s	3 s	1 s	2 s
S1.Time.3	6 s	2 s	2 s	3 s	1 s	2 s
S1.Time.4	6 s	6 s	2 s	3 s	1 s	2 s

cessing delay distribution, i.e., $P(\delta_5)$ follow Poisson distribution and some of the parameters of these distributions such as *mean* value and *max* value, are also reported in Table 5.4.

The second set of test cases is introduced to compare the accuracy of inferred provenance information. These test cases are used in a simulation using artificial data. Therefore, these are not used to evaluate the storage consumption. The simulation is performed for 10000 time units. All these test cases have time-based windows so that we can validate the concept of building *tuple-state graph* to calculate the optimal *offset* value which is used by the probabilistic provenance inference method to infer fine-grained data provenance as discussed in Section 5.7. Table 5.5 shows the window size, trigger interval along with some other parameters associated with sampling interval and processing delay of each test case. For all test cases in Set II, we assume that both sampling interval distribution, i.e., $P(\lambda_4)$ and processing delay distribution, i.e., $P(\delta_5)$ follows Poisson distribution and some of the parameters of these distributions such as *mean* value and *max* value, are reported in Table 5.5.

Test case set II

The test cases shown in Table 5.5 are chosen in such a way that each of them has some variety in their parameters compared to the others. Test case *S2.Time.1* is used to explain the probabilistic provenance inference method as shown in Figure 5.4. Test case *S2.Time.2* differs only in the window size from test case *S2.Time.1*. Test case *S2.Time.2* and *S2.Time.3* are almost similar to each other except the value of trigger interval. Test case *S2.Time.4* and *S2.Time.5* are also different, in their corresponding $P(\delta_5)$ dis-

Table 5.5: Test Case Set II : Parameters of Different Test Cases used for the Evaluation using Simulation

Test case id	Window size in time units	Trigger interval in time units	mean (λ_4) in time units	max (λ_4) in time units	mean (δ_5) in time units	max (δ_5) in time units
S2.Time.1	5	5	2	3	1	2
S2.Time.2	10	5	2	3	1	2
S2.Time.3	10	10	2	3	1	2
S2.Time.4	10	10	3	5	1	2
S2.Time.5	10	10	3	5	2	3
S2.Time.6	10	10	4	6	1	2

tribution. Test case *S2.Time.3*, *S2.Time.4* and *S2.Time.6* have the same values for all parameters except the $P(\lambda_4)$ distribution. These test cases are used to explain the influence of different parameters such as window size, trigger interval, sampling interval, processing delay etc. over the accuracy of the inferred provenance information.

5.9.4 Storage Consumption

The storage consumption by different methods to maintain fine-grained data provenance is one of the major evaluation criteria. We compare the storage consumption among *explicit provenance*, *improved explicit provenance*, *basic provenance inference* and *probabilistic provenance inference* method by facilitating the test cases in set I (see Table 5.4), which are defined for the workflow shown in Figure 5.4. In the result, the storage taken by the *sensor data*, i.e., collection of both input and output data products, is also reported for all test cases.

The *explicit provenance* and the *improved explicit provenance* method maintain fine-grained data provenance explicitly in separate relations. The schema diagrams of these two methods is shown in Figure 4.6 and 4.7, respectively.

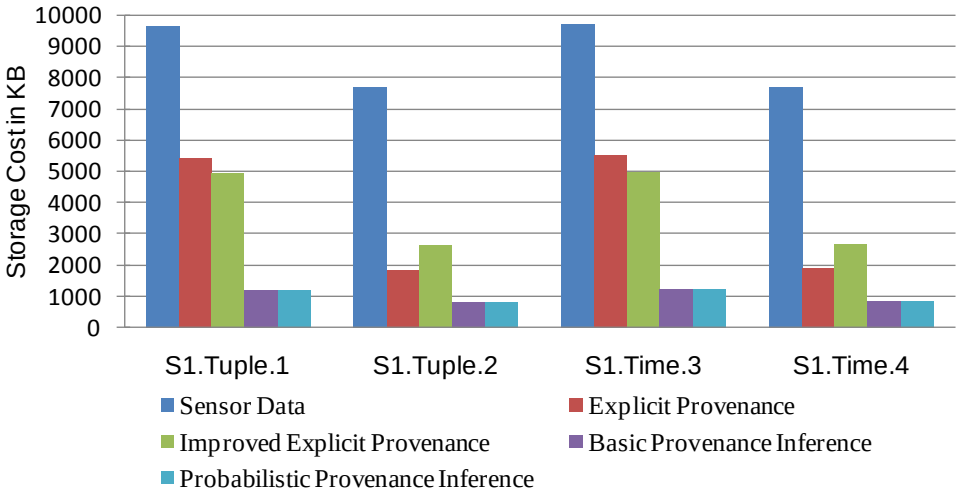


Figure 5.9: Comparison of Storage Consumption among different methods using test case set I

The storage space taken by the relations maintaining provenance data is considered as the storage consumption of the *explicit provenance* and the *improved explicit provenance* method. For the *basic provenance inference* and the *probabilistic provenance inference* method, the storage required by keeping the *transaction time* for all input and output data products is considered as the storage consumption.

Figure 5.9 shows the storage consumption by different methods for all test cases in test case set I, shown in Table 5.4. Since all test cases are defined over the workflow shown in Figure 5.4, they perform an *average* operation. Test case *S1.Tuple.1* has overlapping, tuple-based window with window size of 3 and trigger interval of 1. In this case, the *explicit provenance* method takes around 5400 KB of storage space to maintain fine-grained data provenance. The *improved explicit provenance* method takes less space than the *explicit provenance* method which is 4950 KB of storage space. Both *basic provenance inference* and *probabilistic provenance inference* method take 1200 KB of storage space which is 22% and 24% of storage space consumed by the *explicit provenance* and the *improved explicit provenance* method to maintain fine-grained data provenance.

Overlapping Tuple-based

Test case *S1.Tuple.2* has non-overlapping, tuple-based windows. In this case, the *explicit provenance* method takes around 1850 KB of storage space to maintain fine-grained data provenance. Interestingly, the *improved ex-*

Non-overlapping Tuple-based

plicit provenance method takes more space than the *explicit provenance* method which is 2600 KB of storage space. Since the subsequent windows do not overlap and each input tuple contributes exactly once to produce one output tuple, the *improved explicit provenance* method has more overhead than the *explicit provenance* method. Both *basic provenance inference* and *probabilistic provenance inference* method take 820 KB storage space which is 44% and 32% of storage space consumed by the *explicit provenance* and the *improved explicit provenance* method. In test case *S1.Tuple.2*, the ratio of saving storage space by inference-based methods is less compared to the test case *S1.Tuple.1* because of no overlap between subsequent windows.

Time-based windows

Test case *S1.Time.3* and *S1.Time.4* are the examples of time-based overlapping and non-overlapping windows, respectively. The result of *S1.Time.3* and *S1.Time.4* are similar to tuple-based window test case *S1.Tuple.1* and *S1.Tuple.2*, respectively. Figure 5.9 shows that the inference-based methods also outperform the *explicit provenance* and *improved explicit provenance* in these two test cases as well.

Please note that the reported ratio depends on the chosen window size and trigger specification and if the window size is larger and there is a big overlap between subsequent windows, both *basic provenance inference* and *probabilistic provenance inference* method perform even better.

5.9.5 Accuracy

The accuracy of the inferred provenance information is evaluated and compared between the *probabilistic provenance inference* and the *basic provenance inference*. The accuracy is measured by comparing the inferred fine-grained data provenance of each output data product to the ground truth provided by the explicit provenance method for a particular test case. We facilitate both test case set I and II to evaluate the accuracy of both *probabilistic provenance inference* and *basic provenance inference*. Since, probabilistic provenance inference method can also estimate the accuracy by considering the given distributions only, both *estimated* and *achieved* accuracy of the probabilistic provenance inference method are reported.

Test case set I

Figure 5.10 shows the accuracy of probabilistic provenance inference and basic provenance inference for all test cases in set I, shown in Table 5.4. All these test cases have variable processing delay and variable sampling interval. As already discussed, the basic provenance inference method is prone to infer inaccurate provenance information under variable processing delay

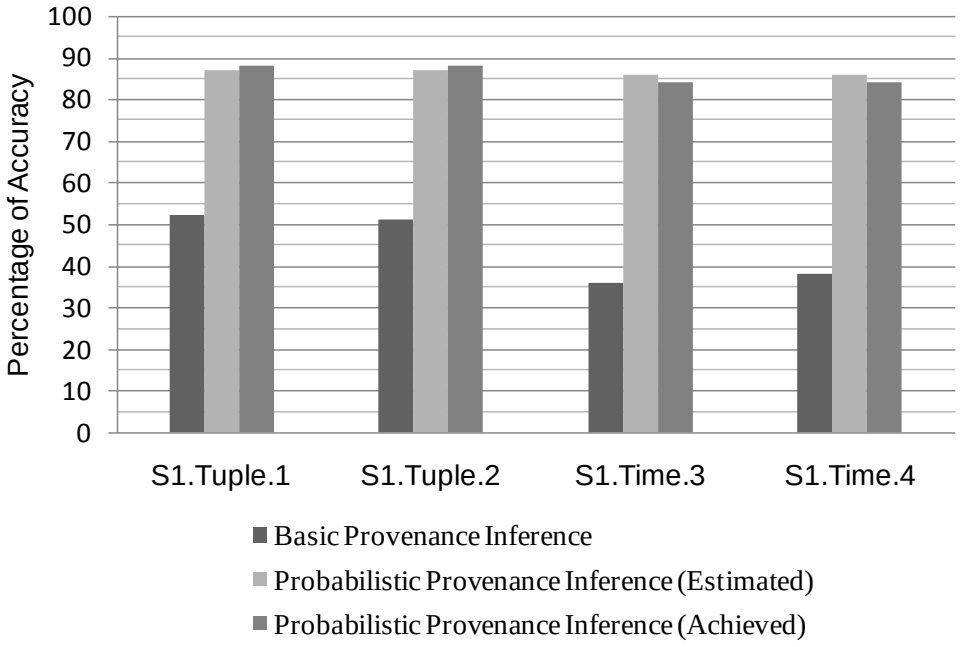


Figure 5.10: Comparison of Accuracy between different inference-based methods using test case set I

and variable sampling interval. This is quite evident in the result reported in Figure 5.10. For test case *S1.Tuple.1* and *S1.Tuple.2*, the basic provenance inference method infers only 52% and 51% accuracy, respectively, while the probabilistic provenance inference method infers 88% accuracy in both cases. In case of time-based windows, i.e., test case *S1.Time.3* and *S1.Time.4*, the basic provenance inference method performs even poorer in terms of accuracy. It only provides around 40% accuracy for these two cases while the probabilistic provenance inference method achieves around 84% accuracy. Therefore, considering all test case in set I, probabilistic provenance inference method achieves higher accuracy than the basic provenance inference method at the same storage consumption.

Test case set II

We also evaluate the accuracy of these two inference-based methods by a simulation facilitating the test cases in set II, shown in Table 5.5. All 6 test cases have time-based windows with different parameters. Figure 5.11 shows the accuracy of probabilistic provenance inference and basic provenance inference method for these test cases. Like the result of test case set I, the probabilistic provenance inference method achieves higher accuracy than the basic provenance inference method for all test cases in

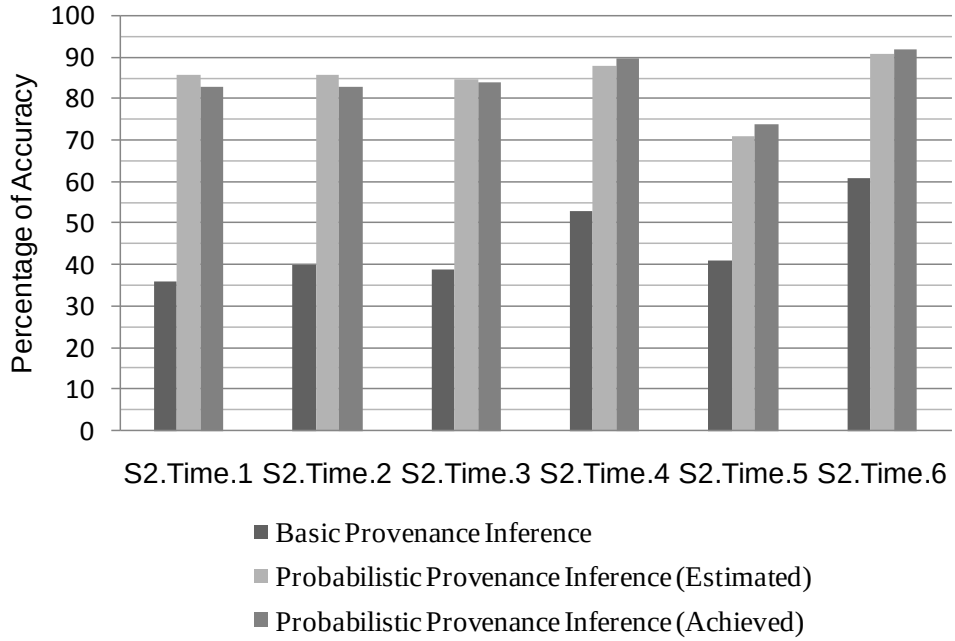


Figure 5.11: Comparison of Accuracy between different inference-based methods using test case set II

set II. Based on this result and *failure conditions* to infer wrong provenance information, we make a few observations which are given below.

Influence over accuracy

In test case *S2.Time.1* and *S2.Time.2*, only the window size is changed while the other parameters remain unchanged. In both cases, we achieve the same level of accuracy by applying the probabilistic provenance inference method. Furthermore, *Failure Condition* 5.2 and 5.3 (see Section 5.3) indicate that window size has no part to play deciding whether accurate or inaccurate provenance can be inferred. Therefore, it seems that *window size does not influence the accuracy*.

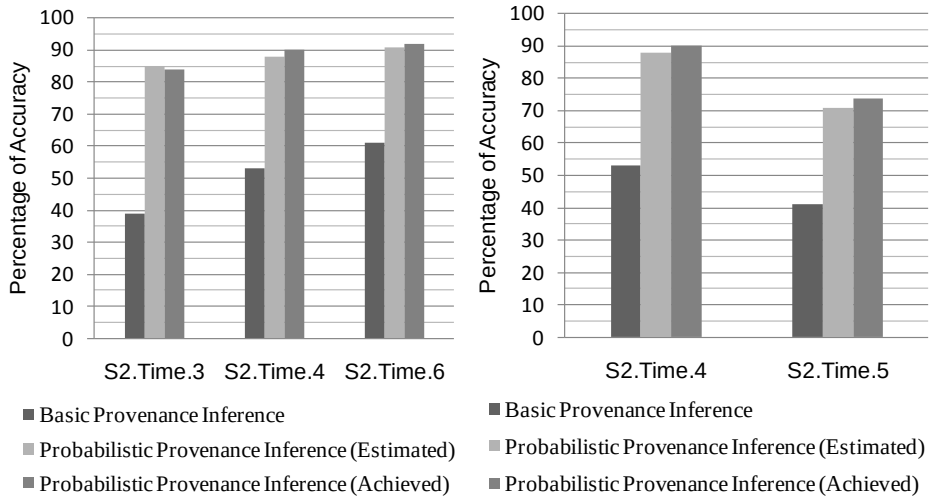
Window size

Trigger interval

Next, we compare the accuracy achieved for test case *S2.Time.2* and *S2.Time.3*. These two cases have the same parameters except the trigger interval. The achieved accuracy is similar for both test cases. Furthermore, the trigger interval is not used to defined *Failure Condition* 5.2 and 5.3. Therefore, it could be possible that *trigger interval has very little influence to the accuracy*.

Sampling interval

Table 5.5 shows that test case *S2.Time.3*, *S2.Time.4* and *S2.Time.6* has the same set of parameters except $mean(\lambda_4)$ and $max(\lambda_4)$. It means that they have different sampling interval distribution $P(\lambda_4)$. Figure 5.12a shows the



(a) Influence of Sampling Interval over Accuracy

(b) Time-based Window executing *Interpolation* operation

Figure 5.12: Influencing Parameters over Accuracy

accuracy of the inference-based methods for these 3 test cases. The highest accuracy is achieved in test case *S2.Time.6* and the lowest one is achieved in test case *S2.Time.3*. Table 5.5 shows that among these 3 test cases, the mean sampling interval, $mean(\lambda_4)$, is the highest in *S2.Time.6* and is the lowest in *S2.Time.3*. In *Failure Condition* 5.2 and 5.3, we can also see that value of λ_i takes a part to decide whether the inference would provide accurate or inaccurate provenance. Based on *Failure Condition* 5.2 and 5.3, keeping δ_k values the same and increasing λ_i values would definitely decrease the chance of a failure. Therefore, this analysis might give a useful indication that *the higher the sampling time, the higher the chance of achieving accurately inferred provenance*.

Processing delay

At last, we compare the accuracy between test case *S2.Time.4* and *S2.Time.5*. These two test cases only differ in $mean(\delta_5)$ and $max(\delta_5)$ values as shown in Table 5.5. Test case *S2.Time.4* has smaller processing delay than the test case *S2.Time.5*. Figure 5.12b shows the achieved accuracy in these test cases. It is quite evident from Figure 5.12b that the higher accuracy is achieved in test case *S2.Time.4*. Furthermore, processing delay takes a part determining whether wrong provenance can be inferred or not based on *Failure Condition* 5.2 and 5.3. Therefore, it seems a reasonable indication that *the*

smaller the processing delay, the higher the chance of achieving accurately inferred provenance.

Furthermore, we can make another observation from Figure 5.10 and 5.11. The estimated accuracy of the *probabilistic provenance inference* method is almost similar to the achieved accuracy of the method. Since the estimated accuracy can be calculated before the actual experiment, it is a useful indicator for the applicability of the probabilistic provenance inference method for a given set of processing delay and sampling interval distributions.

5.10 DISCUSSION

Required information

Like the basic provenance inference method discussed in Chapter 4, the probabilistic provenance inference method has the same set of requirements to be satisfied, discussed briefly in Section 5.5. Some of the requirements such as *explicit system timestamps*, *temporal ordering* are already introduced to process data streams in existing literature. There exist some other assumptions which need to be satisfied by the underlying system to infer accurate provenance information. If these assumptions (see Section 5.5) are not fulfilled by the underlying system, the probabilistic provenance inference method cannot be applied.

Variable ratio operations

The probabilistic provenance inference method is capable of addressing operations with constant *input-output ratio*. In case of an operation with variable *input-output ratio* (e.g. *select* operation in a database), the probabilistic provenance inference method has to transform the input-output ratio of that particular operation from the *variable* ratio to the *constant* ratio, following the same approach taken by the basic provenance inference method as discussed in Section 4.8.

Jumping windows

The probabilistic provenance inference method is applicable for both tuple-based and time-based windows. In case of a time-based window, the proposed method builds a *tuple-state graph* by facilitating Markov chain model, to calculate the optimal offset value for constructing the inferred window. The mechanism of building a *tuple-state graph*, discussed in Section 5.7.2, can handle a non-jumping time-based window, i.e., *window size* $\geqslant$ *trigger interval*. However, the introduced mechanism cannot compute $P(\alpha_i{}^k)$ and $P(\beta_i{}^k)$ distributions correctly in case of a jumping time-based window, i.e., *window size* $<$ *trigger interval*. In this case, there could be a few

input tuples/data products which would not be included in the window and thus, not be processed. However, the current mechanism considers the arrival of these 'not-processed' input data tuples to build a *tuple-state graph*, computing incorrect distributions. However, a little modification in the current mechanism of building a *tuple-state graph* can address a jumping time-based window. In this case, the number of vertices in the *tuple-state graph* is bounded by the value of *trigger interval* instead of *window size*. Later, it is possible to identify the states, i.e., representing a tuple arrival, which fall outside the actual window based on the given *window size*. We can ignore the associated probability of these 'not-processed' states and normalize the distribution to have a correctly computed distribution.

5.11 SUMMARY

In this chapter, we presented the probabilistic provenance inference method that infers fine-grained data provenance accurately at reduced storage costs under variable processing delay and variable sampling interval. The design and development of this method was motivated by the second research question (*RQ 2*) which posed the challenge of managing fine-grained data provenance at reduced storage consumption under different system dynamics. It is an extension of the basic provenance inference method, discussed in Chapter 4, to handle different system dynamics.

At the beginning of this chapter, we defined three *failure conditions* based on the values of processing delay and sampling interval which indicate the possibilities of inferring wrong provenance information in case of both tuple-based and time-based windows. Later, we explained the working principle of the probabilistic provenance inference method based on a scientific workflow that captures sensor measurements on electrical conductivity and facilitates these values in an *average* operation to monitor the change of electrical conductivity. The proposed method has three major phases. First, the workflow provenance is documented, showing the data dependent relationship between processing elements. The next two phases are executed once the user requests provenance for an output data product. In the second phase, the proposed method reconstructs the actual window which had taken part during the execution. The reconstructed window is referred to as the inferred window. The method constructs the inferred window in such a way that the optimal accuracy can be achieved. During

this phase, the method exploits the values of different properties of the particular computing processing element such as window size, processing delay distribution, sampling interval distribution. Furthermore, the proposed method can also estimate the accuracy beforehand based on these distributions and *failure conditions*. It facilitates Markov chain model to construct the inferred window for time-based windows. Finally, the probabilistic provenance inference method associates the selected output data product with the contributing input data products by facilitating the input-output ratio of the particular processing element.

We evaluated the storage consumption and the accuracy of the probabilistic provenance inference method by comparing it to a few other methods such as the explicit provenance collection methods and the basic provenance inference method (see Chapter 4), for different test cases. The evaluation shows that the proposed method takes less space compared to the explicit provenance collection methods for both overlapping and non-overlapping windows. Like the basic provenance inference method, this method reduces storage consumption at higher magnitude if the window size and the overlaps between windows are larger. The accuracy of the probabilistic provenance inference method is calculated by comparing the inferred provenance information to the ground truth provided by the explicit provenance inference method. The probabilistic provenance inference method infers more accurate fine-grained provenance information than the basic provenance inference method at the same storage consumption. The *estimated accuracy* of applying the proposed method is similar to the *achieved accuracy*, and thus, the *estimated accuracy* is a useful indicator to apply the probabilistic provenance inference method for a given test case.

6

MULTI-STEP PROBABILISTIC PROVENANCE INFERENCE

Scientific workflows are widely used by researchers to describe, manage and process scientific computation and analysis. Usually, a scientific workflow represents a data processing chain which has several processing steps and produces the end result. Each of the processing step in a data processing chain is realized by a corresponding computing processing element. A computing processing element processes data products/tuples in input views and produces data products/tuples in an output view. In a data processing chain, there could be some computing processing elements which are producing intermediate results. This intermediate results could be generated because of applying simple operations over the input data products. Therefore, the intermediate result sets might not be useful to scientists. As a result, scientists may not store persistently the intermediate result sets into a view while they only maintain the output view persistently, holding the final result of the scientific computation.

The second research question (*RQ 2*) in this thesis, described in Section 1.4, focuses the challenge of inferring fine-grained data provenance under different situations at reduced storage costs. In Chapter 5, we proposed the probabilistic provenance inference method that can infer fine-grained data provenance under variable processing delay and variable sampling interval. This inference-based method can infer fine-grained data provenance for a scientific workflow with a single processing step, assuming that both input data products and output data products are persistent. In case of a

Challenge

Part of this chapter is based on the work: Fine-Grained Provenance Inference for a Large Processing Chain with Non-materialized Intermediate Views. In *Scientific and Statistical Database Management (SSDBM'12)*, volume 7338 of *LNCS*, pages 397–405, Springer, 2012.

scientific workflow with multiple processing steps, the probabilistic provenance inference method requires all views including the intermediate ones to be persistent to infer fine-grained data provenance, consuming a lot of storage space.

Solution criteria

Therefore, we need an inference-based method that can infer fine-grained data provenance based on a given scientific workflow with multiple processing steps. The envisioned inference-based method should have the capability to infer accurate fine-grained data provenance in presence of non-persistent, intermediate views at a comparatively lower storage costs.

Multi-step Probabilistic Provenance Inference

In this chapter, we present the *multi-step probabilistic provenance inference* method that can infer fine-grained data provenance for an entire processing chain, i.e., a scientific workflow with multiple processing steps, with non-persistent, intermediate views. The *multi-step probabilistic provenance inference* is an extension of the *probabilistic provenance inference* method that also considers the documented workflow provenance information. This method assumes that only the input views and the output view are persistent and other intermediate views are non-persistent. Once scientists request provenance for a particular output data product/tuple, the *multi-step probabilistic provenance inference* method provides a *fine-grained data provenance graph* showing all contributing input tuples to produce that output tuple as vertices and the relationship between tuples as edges. The method assigns a probabilistic value to the inferred *fine-grained data provenance graph* representing the probability of the accuracy of the inferred graph by facilitating the processing delay and sampling interval distribution.

The multi-step probabilistic provenance inference method has further advantages to offer. Like the probabilistic provenance inference method, the *multi-step probabilistic provenance inference* method can estimate the achievable accuracy of the inferred provenance at design time by using Markov chain model and the given processing delay and sampling interval distributions. The working principle of the proposed method is explained based on time-based windows only. It should be possible to extend the multi-step probabilistic provenance inference method to address tuple-based windows too. At the end of this chapter, we briefly discuss the associated challenge posed by a tuple-based window and sketch a possible way of extending the multi-step probabilistic provenance inference method to address tuple-based windows.

This chapter starts with the description of a scenario based on a real project followed by the discussion of the example workflow associated

with the scenario. Next, we define a few basic concepts used to explain the multi-step probabilistic provenance inference method. Afterward, an outline of the multi-step probabilistic provenance inference method is given followed by the list of information to be required by this method to infer fine-grained data provenance. Then we discuss and explain the working principle of the multi-step probabilistic provenance inference method followed by the discussion of the mechanism, estimating the accuracy of the inferred provenance. Finally, we evaluate this method in terms of storage consumption and accuracy followed by the discussion on a few limitations of the proposed inference-based method.

Chapter structure

6.1 SCENARIO AND WORKFLOW DESCRIPTION

We use the scenario introduced in Section 4.1 to explain the multi-step probabilistic provenance inference method. In this section, we provide a brief outline of the scenario and the simplified workflow, defined based on this scenario.

RECORD[1] is one of the projects in the context of the Swiss Experiment[2], which is a platform to enable real-time environmental experiments. In this project, different types of input data products are collected and then processed to monitor river restoration effects. Among these data products, electrical conductivity of the water is also measured which represents the level of salt in water. Scientists use these sensor measurements of electrical conductivity to control the operation of a drinking water well.

Based on the aforesaid scenario, we construct an artificial and simplified workflow which is used to explain the mechanism of the multi-step probabilistic provenance inference method. Figure 6.1 shows the simplified workflow. We assume that there are three sensors measuring electrical conductivity in three different locations. These sensors send data tuples containing the `device id`, the `latitude` and the `longitude` of the location, the measured `electrical conductivity`, the `timestamp` of the measurement, also referred to as *valid time* [79], along with some other attributes. Tuples sent by these sensors are acquired by the source processing elements SP_1, SP_2 and SP_3 and then, are combined by a computing processing element P_4, which stores tuples in view V_4 persistently, as shown in Figure 6.1.

Workflow description

1 Available at `http://www.swiss-experiment.ch/index.php/Record:Home`
2 Available at `http://www.swiss-experiment.ch/`

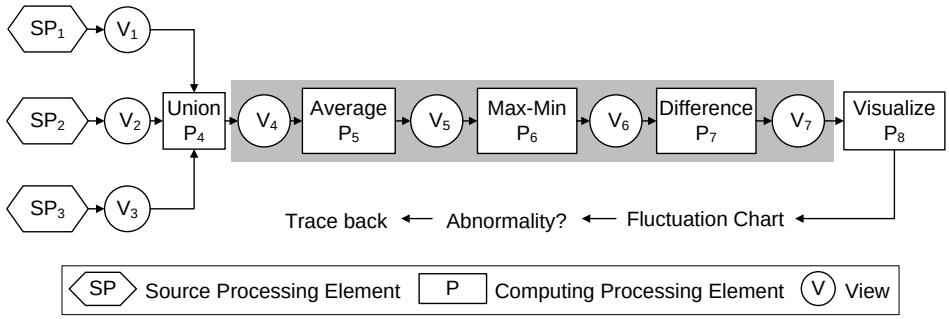


Figure 6.1: The example workflow

Later, the tuples in the input view V_4 are facilitated by a series of operations, represented by P_5, P_6 and P_7 in Figure 6.1, to produce a chart that shows the fluctuation of electrical conductivity over a time period. These computing processing elements are connected along a path (a processing chain) leading towards the generation of the persistent output view V_7 which is facilitated to produce the fluctuation chat. In between the input view V_4 and the output view V_7, there are a couple of non-persistent views, V_5 and V_6, holding intermediate result sets. Scientists could request fine-grained provenance for a particular output data product, shown in the chart, which seems to have an unexpected value.

We consider the shaded part in Figure 6.1 to explain the working principle of the multi-step provenance inference method and to evaluate it. The view V_4 and V_7 hold the input and output tuples, respectively and hence, they are persistent views.

6.2 BASIC TERMINOLOGY

In this section, the definition of the terms which are used to explain the multi-step probabilistic provenance inference method is given. First, we restate the definitions of some basic terms which have already been introduced in Section 4.3.

- *Views:* A view V_i can be defined as a set of tuples ${t_j}^i$ where j is the *transaction time* [79]. The transaction time, j, refers to the system timestamp indicating the point in time when the tuple is inserted into the view V_i.

- *Computing Processing Elements:* A computing processing element, P_k, represents an operation that either computes a value/data product or writes data products into a file, database etc. It takes views as input and produces another view as output.

- *Windows:* A computing processing element, P_k, requires a window to be defined over the input view for its successful execution in the context of data streams. A window $(W_i{}^n)_k$ is a subset of tuples within a view V_i at the n^{th} execution of P_k. A window could be either tuple-based or time-based. A tuple-based window can be defined based on two parameters: i) window size m and ii) a point in time T. A tuple-based window is a finite subset of V_i containing the latest m number of tuples $t_j{}^i$ where $j \leqslant T$. The *window size* is represented as $WS_i{}^k$ where, $WS_i{}^k = m$ (number of tuples). In a time-based window, tuples whose *timestamp* falls into a specific boundary constitutes a window. A time-based window $(W_i{}^n)_k = [start, end)$ is a finite subset of V_i containing all tuples $t_j{}^i$ where $start \leqslant j < end$. In cases of time-based windows, the window size $WS_i{}^k = end - start$ (amount of time units).

- *Trigger Interval:* A trigger interval, TR_k, refers to the predefined interval between two successive executions of a computing processing element, P_k. The trigger interval of a computing processing element could be either tuple-based or time-based.

Like the probabilistic provenance inference method, the multi-step probabilistic provenance inference method can infer fine-grained data provenance under variable processing delay and variable sampling interval. To accomplish that, the multi-step probabilistic provenance inference method takes a few distributions into account. Next, we introduce these distributions and notations to represent them which have already been described in Section 5.2.

Distributions

- *Sampling Interval Distribution:* The amount of time between two successive tuples insertion into a view V_i is referred to as sampling interval, λ_i. λ_i is a discrete random variable which has *integer* values, defined over time domain. The distribution of λ_i is referred to as the sampling interval distribution, denoted as $P(\lambda_i)$.

- *Processing Delay Distribution:* The amount of time to complete the execution of a processing element, P_k, after it is triggered, is referred

to as processing delay δ_k. δ_k is a discrete random variable which has *integer* values, defined over time domain. The distribution of δ_k is referred to as the processing delay distribution, denoted as $P(\delta_k)$.

- *First-tuple appearance Interval:* It refers to the amount of time between a particular window starts which is defined over the view V_i to execute P_k and arrival of the first tuple within that window. It is denoted as ${\alpha_i}^k$ and ${\alpha_i}^k$ is a discrete random variable over time domain. The distribution of the values of ${\alpha_i}^k$ is denoted as $P({\alpha_i}^k)$.

- *Last-tuple disappearance Interval:* It refers to the amount of time between the arrival of the last tuple within a particular window which is defined over the view V_i to execute P_k and the triggering point of that window. it is denoted as ${\beta_i}^k$ and ${\beta_i}^k$ is a discrete random variable over time domain. The distribution of the values of ${\beta_i}^k$ is denoted as $P({\beta_i}^k)$.

Additional terminology

Unlike the probabilistic provenance inference method, the multi-step probabilistic provenance inference method can infer fine-grained data provenance for a processing chain (multiple processing steps) with the presence of non-persistent, intermediate views. The multi-step probabilistic provenance inference method infers a *fine-grained data provenance graph* with a probability indicating how accurate the graph is. To accomplish this, the multi-step probabilistic provenance inference method has to consider the probability of a tuple's existence at a particular point in time. Therefore, in addition to the aforesaid terms, the following term is also used to explain the multi-step probabilistic provenance inference method.

- *Tuple Existence Probability:* The probability of inserting a tuple, ${t_j}^i$, in the view V_i at time j is referred to as tuple existence probability and it is denoted as $P({t_j}^i)$. All tuples in a persistent view, V_p, have the probability $P({t_j}^p) = 1$ while tuples in a non-persistent view V_{np} have probability $P({t_j}^{np}) < 1$.

The aforesaid terms are used to explain the working principle of the multi-step probabilistic provenance inference method presented in this chapter.

6.3 OVERVIEW OF THE MULTI-STEP PROBABILISTIC PROVENANCE INFERENCE

The *multi-step probabilistic provenance inference* method can infer complete fine-grained provenance information for a given processing chain with non-persistent intermediate views. The method has three phases. Like the other inference-based methods discussed in Chapter 4 and 5, first, the workflow provenance information has to be documented which is a one-time action and performed during the setup of the workflow. The next phases are only executed if a user requests provenance information of a particular output data product/tuple.

Documentation of workflow provenance

In the next phase, the proposed method facilitates the processing delay distribution $P(\delta_k)$ of all processing elements P_k which are connected along a path, i.e., a processing chain, producing the output view to calculate the *initial tuple boundary* on the persistent input view. This phase is known as the *backward computation* phase. The input data products/tuples fall within the range of the *initial tuple boundary* might contribute to produce the chosen output tuple.

Backward computation

In the last phase, for each processing element in the chain, original processing windows are reconstructed, i.e., inferred windows, based on the *initial tuple boundary* computed during the backward computation phase. This phase is known as the *forward computation* phase. In this phase, the proposed method also compute the probability of the existence of an intermediate output tuple at a particular timestamp by facilitating the appropriate $P(\delta_k)$ distribution and other windowing constructs such as the window size and the trigger interval etc. Afterward, The proposed method associates the output tuple with the set of contributing input tuples gradually per processing element and this process is continued till we reach the chosen tuple for which provenance information is requested. It provides an inferred *fine-grained data provenance graph* for the chosen tuple.

Forward computation

The *multi-step probabilistic provenance inference* method can estimate the overall accuracy at design time by facilitating the given distributions of processing delay and sampling interval. To achieve this, we use a Markov chain model on the arrival of data tuples within a window to compute specific distributions which is then used to estimate the accuracy of the multi-step probabilistic provenance inference method. A Markov chain is a mathematical system that represents the undergoing transitions from one state to another in a chain-like manner [16].

Estimating accuracy

6.4 REQUIRED INFORMATION

The multi-step probabilistic provenance inference method requires a few information and a set of assumptions to be satisfied by the underlying execution environment like the other inference-based methods as discussed in Chapter 4 and 5. Details on this required information and assumptions have already been discussed in Section 4.5. In this section, we summarize this required information and the set of assumptions.

First, it is required to attach the *transaction time* (system timestamp) to every data products/tuples. Second, the multi-step probabilistic provenance inference method requires the processing elements to process data products/tuples based on their order on *transaction time* in the input view, following *temporal ordering* of tuples during the processing. At last, like any inference-based method, the multi-step probabilistic provenance inference method also facilitates the documented workflow provenance of a scientific model to infer fine-grained data provenance.

Furthermore, there are a few assumptions required to be fulfilled to apply the proposed method. These assumptions, applicable for a computing processing element with multiple input views or producing multiple output data products, have been discussed in Section 4.5.2. One of these assumptions indicates that the inference mechanism must know the *order of input views* which participate in an activity, realized by a computing processing element. Another assumption mentions that the name of the *contributing input view* shall be documented explicitly in cases the activity to be performed has multiple input views. The other assumption also has to be satisfied to ensure that the *order of tuples in the output view* follows the same order found in input views. If these assumptions are not fulfilled by the underlying system, the multi-step probabilistic provenance inference method cannot be applied.

6.5 DOCUMENTATION OF WORKFLOW PROVENANCE

The *documentation of workflow provenance* is the pre-requisite phase which has to be completed before the actual execution of the inference-based method. In this phase, the workflow provenance of the entire data processing is explicated. The multi-step probabilistic provenance inference method requires the same set of properties of a computing processing ele-

ment to be documented as the probabilistic provenance inference method, listed in Section 5.6. We provide a list of these properties below.

- *Window type*: refers to a list of window types; one element for each input view. The value can be either *tuple* or *time*.
- *Window size*: refers to a list of window sizes; one element for each input view. The value represents the size of the window.
- *Trigger type*: specifies how a *computing processing element* will be triggered for execution; The value can be either *tuple* or *time*.
- *Trigger interval*: refers to the interval between successive executions of the same computing processing element.
- *Input-output ratio*: refers to the ratio of the number of input data products contributed to produce output data products over the number of output data products of a particular computing processing element.
- *Number of input views*: refers to the total number of input views.
- *Identifier of input views*: refers to the list of ids (node identifiers) of input views.
- *Contributing input views*: refers to the fact that whether a computing processing element with multiple input views processes data products over *all* input views or a *specific* input view at a time. For computing processing elements with only one input view, it is set to *not applicable*.
- *Processing delay distribution*: refers to the distribution of amount of time required by a computing processing element to complete the execution over the defined window.

Furthermore, the sampling interval distribution of the input view of the workflow has to be documented also. It refers to the distribution of amount of time between two successive tuples insertion into the view V_i, which is an input view of a computing processing element P_k.

Example

Figure 6.2 shows the artificial workflow described in Section 6.1 and its explicated workflow provenance. In Figure 6.2, the workflow consists of three computing processing elements such as P_5, P_6, P_7, and four views such as V_4, V_5, V_6 and V_7. Among the views, V_4 is considered as the input

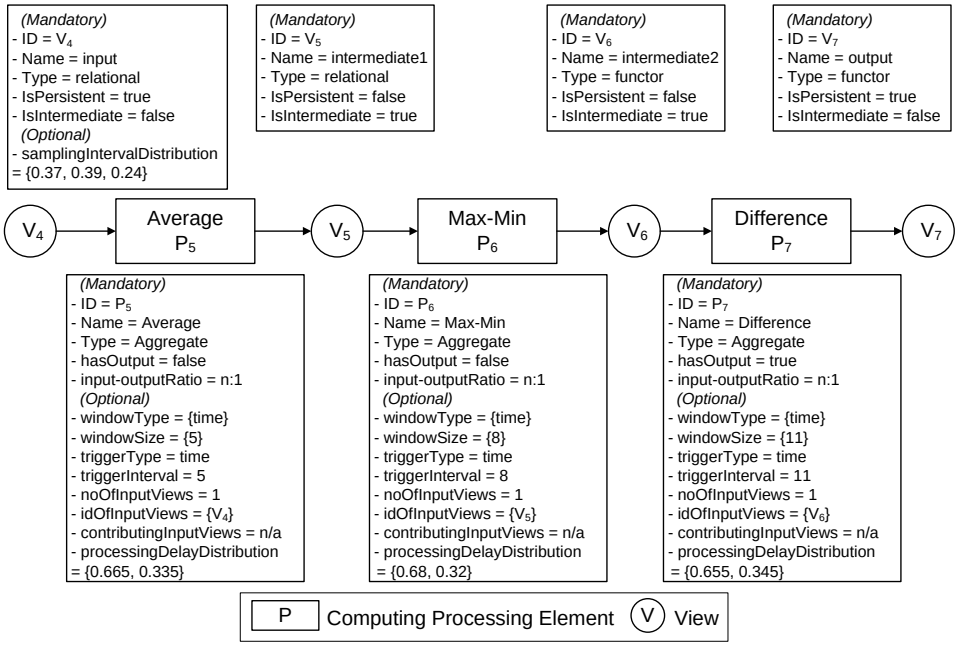


Figure 6.2: Example of the explicated workflow provenance

view and V_7 is the output view and therefore, they are persistent views. The other views, V_5 and V_6 are non-persistent, intermediate views. The multi-step probabilistic provenance inference method facilitates the explicated workflow provenance, shown in Figure 6.2 to infer fine-grained data provenance as well as to estimate the accuracy of the inferred provenance. Figure 6.2 shows the sampling interval distribution of the input view V_4, $P(\lambda_4)$ which is $\{0.37, 0.39, 0.24\}$ for time unit 1, 2 and 3, respectively. It means that 37% tuples arrive after 1 time unit from the previous tuple arrival and so on. Furthermore, it also shows the given processing delay distributions of involved computing processing elements along with other parameters such window size, trigger interval etc. As an example, $P(\delta_5)$ has the values $\{0.665, 0.335\}$ indicating that 66.5% times the processing delay is 1 time unit and so on. The next two phases of the multi-step probabilistic provenance inference method facilitates this documented workflow provenance information as shown in Figure 6.2 to infer the *fine-grained data provenance graph*.

6.6 BACKWARD COMPUTATION OF MULTI-STEP PROBABILISTIC PROVENANCE INFERENCE

The backward computation phase will be only executed if the provenance information is requested for a particular output tuple T produced by the computing processing element P_7 as shown in Figure 6.2. In the multi-step probabilistic provenance inference, the backward computation phase calculates a tuple boundary with *lower bound* and *upper bound* over the input view which possibly holds all input tuples which might have contributed to produce the chosen output tuple T. This tuple boundary is referred to as the *initial tuple boundary*. Since multiple processing steps are involved, the *initial tuple boundary* on the input view might include input tuples from several windows which were defined over the input view during the actual execution. Therefore, the *initial tuple boundary* is not the same as the *inferred window* in the probabilistic provenance inference method representing the reconstructed original window, used for a single processing step.

Initial tuple boundary

The *initial tuple boundary* is calculated by facilitating the explicated workflow provenance as shown in Figure 6.2 and the *transaction time* of the tuple *T* which is referred to as the *reference point*. Calculating both *upper bound* and *lower bound* of the *initial tuple boundary* requires to consider the processing delay distribution, $P(\delta_k)$ of all computing processing elements P_k, which are connected along a path leading towards the persistent output view in the given workflow. Therefore, in this case, $P(\delta_5)$, $P(\delta_6)$ and $P(\delta_7)$ are considered and thus, the value of k ranges from 5 to 7. Since the processing delay of a particular computing processing element vary from one window execution to another, we consider the minimum processing delay, $min(\delta_k)$ of a computing processing element, P_k, extracted from the given distribution $P(\delta_k)$, to calculate the *upper bound* of the *initial tuple boundary*. The processing delay distributions of computing processing elements are shown in Figure 6.2. The *upper bound* of the *initial tuple boundary* can be calculated based on the following equation.

Calculating upper bound

$$upperBound = referencePoint - \sum_{k=5}^{k=7} min(\delta_k) \tag{6.1}$$

where $min(\delta_k)$ refers to the minimum processing delay of a computing processing element $P_k \in \{P_5, P_6, P_7\}$

Calculating lower bound

The *lower bound* of the *initial tuple boundary* can be calculated by considering both processing delay and associated window size. In this case, we consider the maximum processing delay, $max(\delta_k)$ of a computing processing element, P_k, extracted from the given distribution $dist(\delta_k)$, to calculate the *lower bound* of the *initial tuple boundary*. Moreover, the size of the window, ${WS_i}^k$ defined over the input view V_i to execute the computing processing element P_k has to be considered. The *lower bound* of the *initial tuple boundary* can be calculated based on the following equation.

$$\mathtt{lowerBound} = \mathtt{referencePoint} - \sum_{k=5}^{k=7} max(\delta_k) - \sum_{k=5 \wedge i=4}^{k=7 \wedge i=6} {WS_i}^k \quad (6.2)$$

where $max(\delta_k)$ refers to the maximum processing delay of a computing processing element $P_k \in \{P_5, P_6, P_7\}$ and ${WS_i}^k$ refers to the window size defined over view $V_i \in \{V_4, V_5, V_6\}$ to execute $P_k \in \{P_5, P_6, P_7\}$, respectively.

Both *upper bound* and *lower bound* return a point in time and the tuples in the input view V_4 whose *transaction time* is less than the *upper bound* and is greater than or equal to the *lower bound* will be included in the *initial tuple boundary*.

Figure 6.3 shows a snapshot of all associated views based on the explicated workflow provenance depicted in Figure 6.2. In Figure 6.3, the view V_4 is the input view and the view V_7 is the output view. Therefore, these views are persistent. The other two views, V_5 and V_6, hold intermediate result set and therefore, they are non-persistent views. However, the tuples in these intermediate views are shown in Figure 6.3 to calculate the accuracy of the multi-step probabilistic provenance inference method later. The directed edges in Figure 6.3 represent data dependency between input tuples and the corresponding output tuple. For the simplicity, we assume that all computing processing elements in Figure 6.3 start executing at the same point in time which is 0. This is not an assumption that must be satisfied by the underlying execution environment. If a computing processing element starts executing at a later point in time, the difference has to be documented to apply the proposed method.

Example

Scientists request the provenance for the tuple ${t_{46}}^7$ shown in Figure 6.3. Please note that we exclude the index referring to a view when representing a tuple in Figure 6.3 since the tuples are placed within a rectangle, beneath each view. The tuple ${t_{46}}^7$ is referred to as the chosen tuple, *T*. The

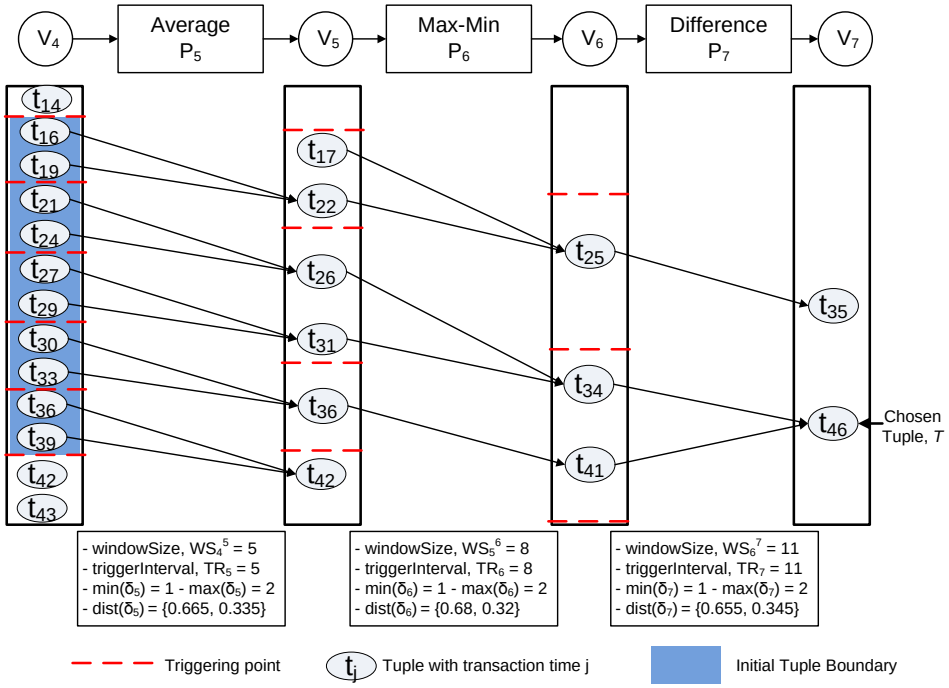


Figure 6.3: A snapshot of the views holding tuples

transaction time of the chosen tuple T is 46 and it is referred to as the *reference point* which is used to calculate the *upper bound* and *lower bound* of the *initial tuple boundary*. Based on Equation 6.1 and 6.2 and the explicated workflow provenance shown in Figure 6.2, the *upper bound* and the *lower bound* of the *initial tuple boundary* is:

$$\begin{aligned}
\mathtt{upperBound} &= \mathtt{referencePoint} - [\min(\delta_7) + \min(\delta_6) + \min(\delta_5)] \\
&= 46 - [1 + 1 + 1] \\
&= 43 \\
\mathtt{lowerBound} &= \mathtt{referencePoint} - [\max(\delta_7) + \max(\delta_6) + \max(\delta_5)] \\
&\qquad - [WS_6{}^7 + WS_5{}^6 + WS_4{}^5] \\
&= 46 - [2 + 2 + 2] - [11 + 8 + 5] \\
&= 16
\end{aligned}$$

Therefore, the *initial tuple boundary* is $[16, 43)$ defined over the input view V_4. The initial tuple boundary can be corrected based on the triggering points of the computing processing element P_5, which has the view V_4

Correcting initial tuple boundary

as an input. The *triggering points* refer to the points in time when the computing processing element will be triggered according to the *trigger interval* parameter. In this case, P_5 has trigger interval of 5 time units and therefore, it will be triggered on time 5, 10, 15 and so on. Both *lower bound* and *upper bound* of the *initial tuple boundary* can be corrected based on these triggering points. The *corrected lower bound* should start just after the highest triggering point which is less than the *lower bound* of the initial tuple boundary. The *corrected upper bound* should not exceed the highest triggering point which is less than the *upper bound* of the initial tuple boundary.

Since the *lower bound* of the initial tuple boundary is 16 and the highest triggering point which is less than 16 is 15, the *corrected lower bound* is $15 + 1 = 16$. The *corrected upper bound* is 40 because it is the highest triggering point which is less than the *upper bound* of the initial tuple boundary, 43. Therefore, the corrected initial tuple boundary is $[16, 40]$ where both *lower bound* and *upper bound* are inclusive. The tuples falling in this range are highlighted by using a shaded rectangle within the view V_4 in Figure 6.3.

The corrected initial tuple boundary may contain some input tuples which have not contributed to produce the chosen output tuple, T. These tuples will be pruned during the next phase of the multi-step probabilistic provenance inference.

6.7 FORWARD COMPUTATION OF MULTI-STEP PROBABILISTIC PROVENANCE INFERENCE

Overview

In this phase, the *multi-step probabilistic provenance inference* method infers the *fine-grained data provenance graph* for the *chosen tuple*. This graph is also referred to as the *inferred provenance graph*. The forward computation phase starts with the first processing step where the corresponding computing processing element, P_k, has the persistent input view, V_i, as an input and produces output tuples, contained possibly in an intermediate view, V_{i+1}. Since intermediate views are non-persistent, the exact *transaction time* of its tuples are not known. Therefore, the forward computation phase establish data dependencies between the input tuples within the window to the *prospective tuples* in the intermediate view by facilitating the given processing delay distribution $P(\delta_k)$ and the triggering points of P_k. The *prospective tuples* refer to a set of tuples with different *transaction time* among which only one of them could be produced based on the actual processing delay.

Therefore, a prospective tuple always has a probability value that refers to the probability of its existence. It is defined as the *tuple existence probability* in Section 6.2. This phase calculates the *tuple existence probability* for tuples produced by all computing processing elements in the given processing chain. The forward computation phase carries out this process till it reaches the chosen tuple, T, for which the provenance information is requested.

Different cases

Depending on the order of a computing processing element, the mechanism to calculate the *tuple existence probability* could differ. There are three cases to consider. First, a tuple is produced by the first computing processing element in the chain, i.e., first processing step (P_5 in Figure 6.2). In this case, the particular computing processing element has persistent input view but non-persistent output view. Second, a tuple is produced by an intermediate computing processing element, i.e., intermediate step (P_6 in Figure 6.2), that has a non-persistent input view and non-persistent output view. Please note that, this case occurs several times if a processing chain has more than three processing steps and does not occur at all if a processing chain has less than three processing steps. Third, a tuple is produced by the last computing processing element, i.e., last processing step (P_7 in Figure 6.2), which has a non-persistent input view but persistent output view. Next, we explain each case and provide a formula to calculate the *tuple existence probability*.

First step

Figure 6.4 shows the *prospective tuples* along with its *tuple existence probability* for the first processing step where the computing processing element P_5 takes the persistent view V_4 as an input and produces intermediate results, hold by the non-persistent view V_5. Since V_4 is a persistent view, all tuples in V_4 with *transaction time* j are assigned with *probability*, $P(t_j{}^4) = 1$. Figure 6.4 shows that P_5 has 5 different triggering points based on its *trigger interval*, within the corrected *initial tuple boundary*, $[16, 40]$. These triggering points are: at time 20, 25, 30, 35 and 40. Based on these triggering points, P_5 produces output tuples, hold by the non-persistent view V_5. Therefore, the *tuple existence probability* of these output tuples has to be calculated based on the processing delay distribution of P_5, denoted as $P(\delta_5)$. For each triggering point at l of P_5, the probability of getting a prospective tuple at time k, where $k > l$, in V_5 can be calculated using the following formula based on the given workflow provenance , shown in Figure 6.2.

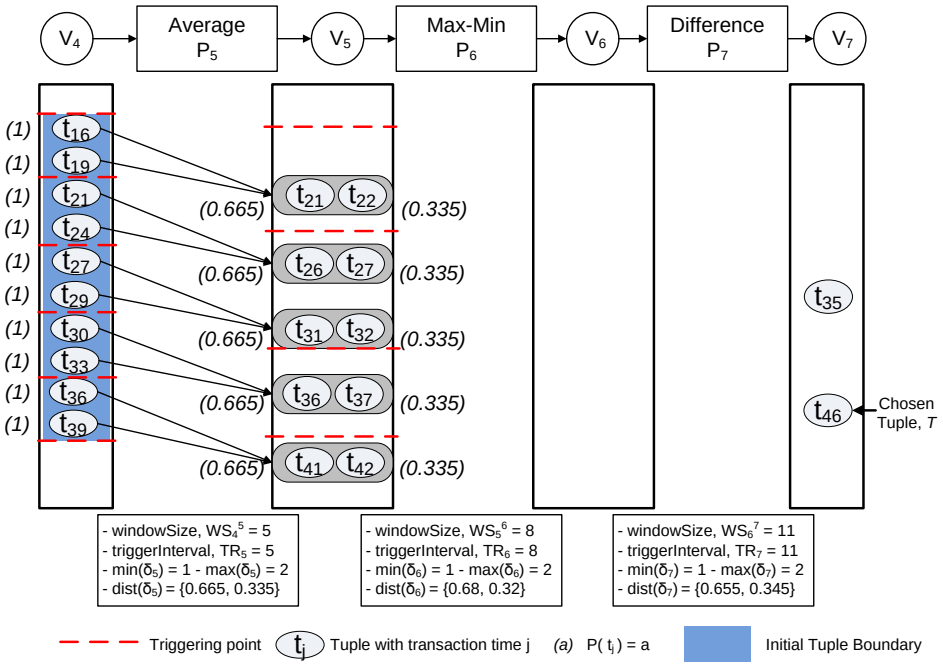


Figure 6.4: Forward Computation for the first processing step

$$P(t_k{}^5) = P(\delta_5 = k - l) \tag{6.3}$$

As mentioned in Figure 6.2, $P(\delta_5 = 1) = 0.665$ and $P(\delta_5 = 2) = 0.335$. Therefore, based on Equation 6.3 the *tuple existence probability* of an output tuple at time 26 (=k) for the triggering at time 25 (=l) is:

$$\begin{aligned} P(t_{26}{}^5) &= P(\delta_5 = 26 - 25) \\ &= P(\delta_5 = 1) = 0.665 \end{aligned}$$

Using the same Equation 6.3, we can also calculate the the *tuple existence probability* of an output tuple at time 27 (=k) for the same triggering point at time 25 (=l).

$$\begin{aligned} P(t_{27}{}^5) &= P(\delta_5 = 27 - 25) \\ &= P(\delta_5 = 2) = 0.335 \end{aligned}$$

Example Figure 6.4 shows the data dependencies between the tuples in the input view V_4 and the prospective tuples in the intermediate view V_5. For

each triggering point of P_5, two prospective tuples are grouped together in view V_5. It indicates that either one of these two tuples is produced from the contributing input tuples in V_4. The *tuple existence probability* values of these tuples are also shown in Figure 6.4.

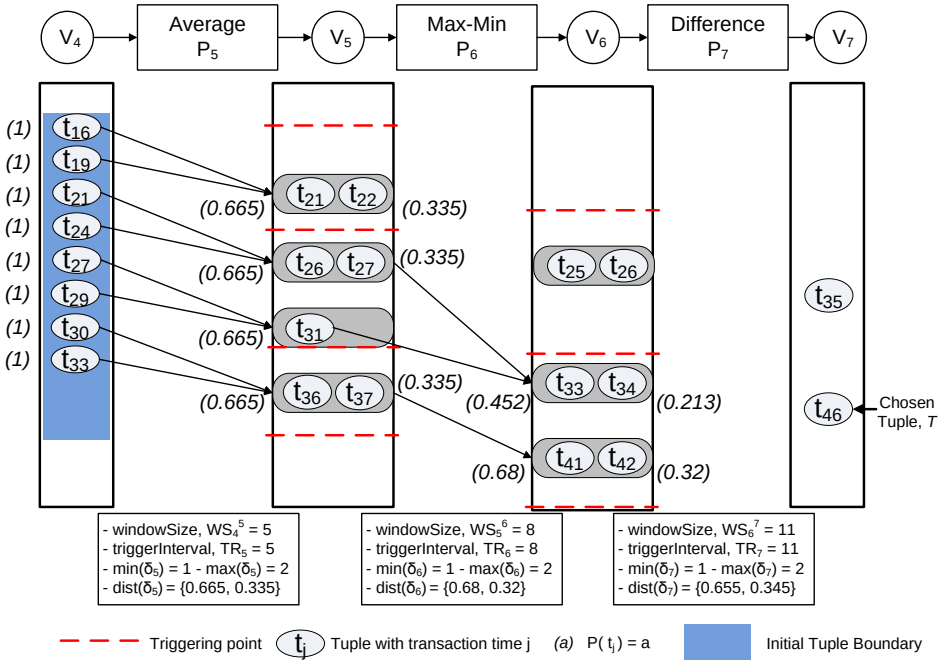
Intermediate step

The forward computation phase now considers the next processing step. This is an intermediate processing step where the computing processing element, P_6, takes a non-persistent, intermediate view V_5 as an input and produces results in another non-persistent, intermediate view V_6. The *trigger interval* of P_5 is 8 time units. The different triggering points of P_6 within the corrected initial tuple boundary $[16, 40]$ are: at time 24, 32 and 40. These triggering points are shown in Figure 6.4.

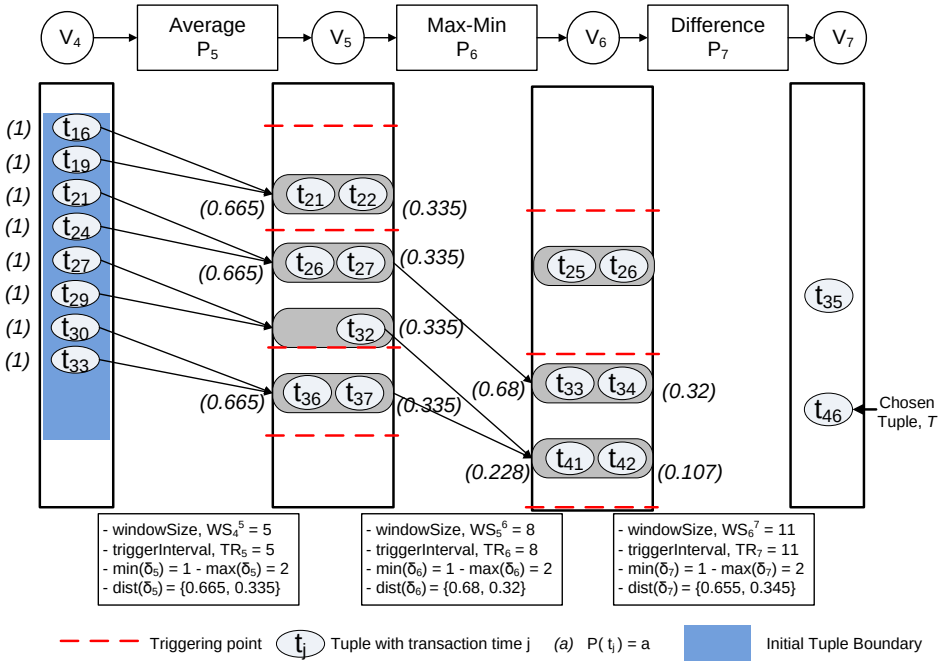
In an intermediate processing step, the tuples within a window defined over the input view could be produced by more than one execution of the previous computing processing element based on its triggering points. As an example, P_6 has a triggering point at 32 and the window contains tuples within the range $[24, 32)$ which were produced by the triggering points at 25 and 30 of P_5. The number of triggering points of the previous processing step which are considered within a window of the current computing processing element is referred to as the *contributing points*, denoted as cp. In the aforesaid example, $cp = 2$. Moreover, the possible timestamps to have a tuple due to a particular triggering point of the previous computing processing element might fall into two different windows of the current processing element. This results into different choice of paths to infer the fine-grained data provenance graph. As an example, for the triggering point at time 32 of P_6, there exist two options: i) ${t_{31}}^5$ is included within the window $[24, 32)$ defined over view V_5 and ii) ${t_{32}}^5$ is included within the window $[32, 40)$ defined over view V_5. Figure 6.5 shows the data dependencies between input and output tuples for both options. In an intermediate step, the *tuple existence probability* at transaction time k produced by a triggering point at time l of P_6 is calculated by using the following formula.

$$P({t_k}^6) = \prod_{x=1}^{cp} (\sum P(\textit{prospective tuples})) \times P(\delta_6 = k - l) \tag{6.4}$$

As depicted in Figure 6.2, $P(\delta_6 = 1) = 0.68$ and $P(\delta_6 = 2) = 0.32$. For option i), where the tuple ${t_{31}}^5$ is included within the window $[24, 32)$



(a) Forward Computation for the intermediate processing step including $t_{31}{}^{5}$



(b) Forward Computation for the intermediate processing step excluding $t_{31}{}^{5}$

Figure 6.5: Forward Computation for the intermediate processing step

defined over V_5, the *tuple existence probability* of an output tuple at time 33 and 34 due to the triggering of P_6 at time 32 based on Equation 6.4 are:

$$\begin{aligned}
P({t_{33}}^6) &= [\{P({t_{26}}^5) + P({t_{27}}^5)\} \times \{P({t_{31}}^5)\}] \times P(\delta_6 = 33 - 32) \\
&= [(0.665 + 0.335) \times 0.665] \times P(\delta_6 = 1) \\
&= 0.665 \times 0.68 \\
&= 0.452 \\
P({t_{34}}^6) &= [\{P({t_{26}}^5) + P({t_{27}}^5)\} \times \{P({t_{31}}^5)\}] \times P(\delta_6 = 34 - 32) \\
&= [(0.665 + 0.335) \times 0.665] \times P(\delta_6 = 2) \\
&= 0.665 \times 0.32 \\
&= 0.213
\end{aligned}$$

According to option ii), the tuple ${t_{32}}^5$ is included within the window $[32, 40)$ defined over V_5 which means that there is no tuple produced at time 31 in view V_5. Therefore, only the tuple produced at either time 26 or time 27 contributes to produce the output tuples ${t_{33}}^6$ and ${t_{34}}^6$. For option ii), based on Equation 6.4, the *tuple existence probability* of an output tuple at time 33 and 34 due to the triggering of P_6 at time 32 are:

$$\begin{aligned}
P({t_{33}}^6) &= [\{P({t_{26}}^5) + P({t_{27}}^5)\}] \times P(\delta_6 = 33 - 32) \\
&= [(0.665 + 0.335)] \times P(\delta_6 = 1) \\
&= 1.000 \times 0.68 \\
&= 0.68 \\
P({t_{34}}^6) &= [\{P({t_{26}}^5) + P({t_{27}}^5)\}] \times P(\delta_6 = 34 - 32) \\
&= [(0.665 + 0.335)] \times P(\delta_6 = 2) \\
&= 1.000 \times 0.32 \\
&= 0.32
\end{aligned}$$

Example

Figure 6.5a and 6.5b show the data dependencies among the tuples in view V_4, V_5 and V_6. Neither of views V_5 and V_6 are persistent. Therefore, the tuples in view V_5 and V_6 are the *prospective tuples* and in most cases, they are grouped together based on their triggering points, indicating that

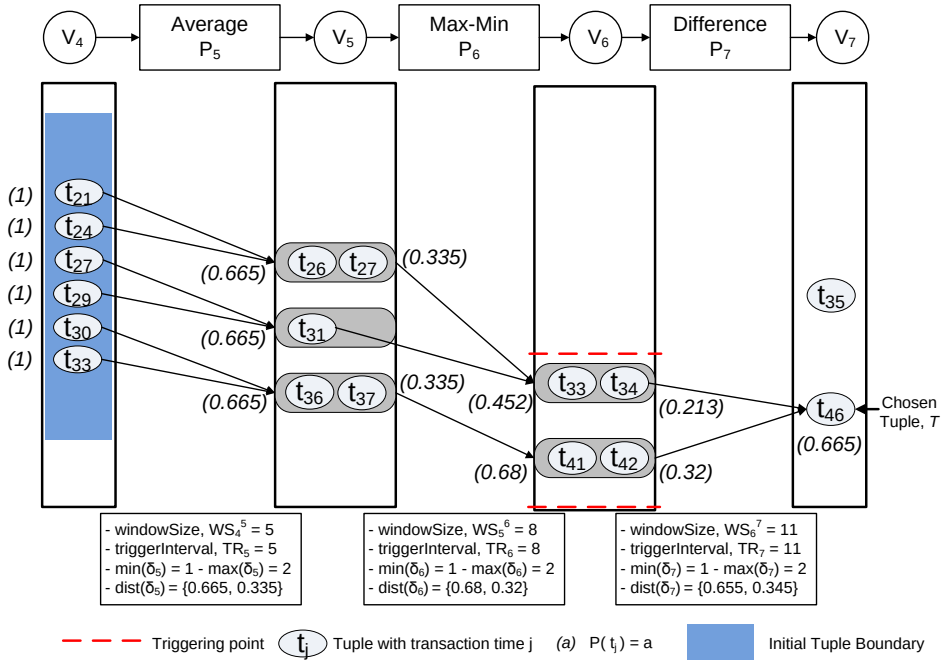
either one of these tuples exists. There are a few exceptions. Figure 6.5a shows the case where we assume that ${t_{31}}^5$ exists and it contributes to produce either one of the tuples ${t_{33}}^6$ or ${t_{34}}^6$. Figure 6.5b shows the opposite case. In this case, ${t_{31}}^5$ does not exist and hence, it does not contribute to produce neither ${t_{33}}^6$ nor ${t_{34}}^6$. In fact, in this case, ${t_{32}}^5$ exists and it contributes to produce either one of the tuples ${t_{41}}^6$ or ${t_{42}}^6$. The data dependencies in both figures are shown by using directed edges from input to output tuples. The *tuple existence probability* values of these tuples are also shown in Figure 6.5a and 6.5b.

Last step

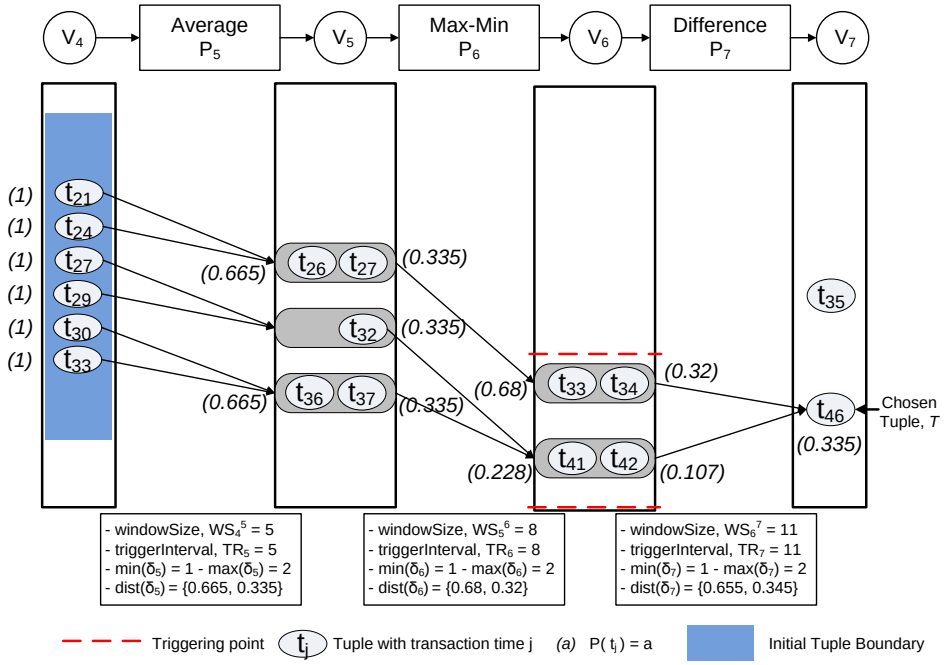
The forward computation phase applies the same process for all intermediate processing steps. At the last processing step, it will establish the data dependencies between the chosen tuple, T, and the contributing input tuples. Based on the given workflow shown in Figure 6.2, the final processing step involves the computing processing element P_7 which takes non-persistent, intermediate view V_6 as an input and produces the output tuples in a persistent view V_7. The computing processing element P_7 has different triggering points at time 22, 33, 44 and so on based on its *trigger interval*, as mentioned in Figure 6.2. Since it is the final processing step, only the highest triggering point of P_7 which is less than the *transaction time* of T is considered. Since the *transaction time* of T is 46, we consider the triggering point at time 44. Therefore, the tuples within the window $[33, 44)$ defined over the input view V_6 participate to produce the chosen tuple ${t_{46}}^7$. The *tuple existence probability* of the chosen tuple can be calculated by a little modification of Equation 6.4. Since this output view V_7 is persistent, the existence of the chosen tuple at its *transaction time* is certain. Therefore, the processing delay distribution, $P(\delta_7)$, is not needed to consider. The formula to calculate the *tuple existence probability* of the chosen tuple, T, in the final processing step is given below.

$$P(T) = \prod_{x=1}^{cp} (\sum P(\textit{prospective tuples})) \tag{6.5}$$

We have to calculate the *tuple existence probability* of the chosen tuple, T, for all aforementioned options. Therefore, for option i), where the tu-



(a) Forward Computation for the last processing step including $t_{31}{}^{5}$



(b) Forward Computation for the last processing step excluding $t_{31}{}^{5}$

Figure 6.6: Forward Computation for the last processing step

ple ${t_{31}}^5$ is included within the window $[24, 32)$ defined over V_5, the *tuple existence probability* of the chosen tuple ${t_{46}}^7$ is:

$$\begin{aligned} P({t_{46}}^7) &= [\{P({t_{33}}^3) + P({t_{34}}^3)\} \times \{P({t_{41}}^3) + P({t_{42}}^3)\}] \\ &= [(0.452 + 0.213) \times (0.665 + 0.335)] \\ &= 0.665 \times 1.000 \\ &= 0.665 \end{aligned}$$

For option ii), where the tuple ${t_{31}}^5$ does not exist and the tuple ${t_{32}}^5$ is included within the window $[32, 40)$ defined over V_5, the *tuple existence probability* of the chosen tuple ${t_{46}}^7$ is:

$$\begin{aligned} P({t_{46}}^7) &= [\{P({t_{33}}^3) + P({t_{34}}^3)\} \times \{P({t_{41}}^3) + P({t_{42}}^3)\}] \\ &= [(0.68 + 0.32) \times (0.228 + 0.107)] \\ &= 1.000 \times 0.335 \\ &= 0.335 \end{aligned}$$

Example

Figure 6.6a and 6.6b show the data dependencies for the complete processing chain, as shown in Figure 6.2. Figure 6.6a shows an *inferred provenance graph* for the chosen tuple T with probability 0.665. Figure 6.6b shows the other *inferred provenance graph* for the chosen tuple T with probability 0.335. As we have already mentioned, this probability value refers to the probability of the accuracy of the *inferred provenance graph*. We choose the *inferred provenance graph* with the highest probability value because of the performance, maximizing the chance of accurately inferred provenance information. Therefore, the *inferred provenance graph* shown in Figure 6.6a is selected as the output of this forward computation phase. Comparing it with the snapshot shown in Figure 6.3, we can conclude that the *inferred provenance graph*, depicted in Figure 6.6a, provides accurate provenance information for the aforesaid example.

6.8 ACCURACY ESTIMATION

The *multi-step probabilistic provenance inference* method can estimate the accuracy of inferred fine-grained provenance information. Estimating the accuracy depends on the *failure conditions*, discussed in Section 5.3 and the given processing delay and sampling interval distributions as shown in Figure 6.2. By facilitating this information, the proposed method can estimate

the optimal accuracy of each processing step. Please note that, currently, we do not involve this accuracy estimation technique during the backward computation and the forward computation phase of the proposed method, inferring fine-grained data provenance graph. In the future, we would like to extend the proposed method by integrating the accuracy estimation technique during the provenance inference phase.

Failure conditions

As already mentioned, the accuracy of the *multi-step probabilistic provenance inference* method is estimated based on two failure conditions, defined in Chapter 5. *Failure Condition* 5.2 identifies that an inaccurate provenance inference can occur *if the processing delay* δ_k *of a computing processing element* P_k *is longer than the amount of time between the original window starts and the arrival of the first tuple in the original window which is defined over the view* V_i*, an input view to* P_k. It indicates that if ${\alpha_i}^k < \delta_k$ holds, there could be a failure by excluding a contributing input tuple. ${\alpha_i}^k$ refers to the first-tuple appearance interval, as defined in Section 6.2. *Failure Condition* 5.3 identifies that an inaccurate provenance inference can occur if *a non-contributing input tuple is inserted into the input view,* V_i*, before completing the execution of a computing processing element,* P_k*, on the current window defined over the input view,* V_i. It indicates that if $\lambda_i - {\beta_i}^k < \delta_k$ holds, there could be a failure by including a non-contributing input tuple. ${\beta_i}^k$ refers to the last-tuple disappearance interval, as defined in Section 6.2.

Required distributions

Furthermore, the *multi-step probabilistic provenance inference* method computes specific distributions such as $P({\alpha_i}^k)$ and $P({\beta_i}^k)$ based on a few given distributions to estimate the accuracy. The method requires the processing delay distribution, $P(\delta_k)$, of all computing processing elements, P_k, connected along a path that produces final output. The sampling interval distribution, $P(\lambda_i)$, of all associated views, V_i, except the output view of the workflow are also required. While all processing delay distributions are observed, only the sampling interval distribution of the input view is observed to minimize the storage overhead incurred by observing sampling interval distributions of other non-persistent views in the processing chain. The sampling interval distributions of non-persistent, intermediate views should be also computed.

Overview

The accuracy estimation process works in the following way. Assuming a processing step that involves a computing processing element P_k, having an input view V_i and producing an output view V_{i+1}. We also assume that the sampling interval distribution of the input view V_i, $P(\lambda_i)$, is given or already computed. First, the sampling interval distribution of

the view V_{i+1}, $P(\lambda_{i+1})$, has to be computed unless V_{i+1} contains the final result and hence, a persistent view. Next, based on the $P(\lambda_i)$ distribution and the given $P(\delta_k)$ distribution, both $P({\alpha_i}^k)$ and $P({\beta_i}^k)$ distribution are computed. Later, the joint probability distribution between the given $P(\delta_k)$, $P(\lambda_i)$ and the computed $P({\alpha_i}^k)$ and $P({\beta_i}^k)$ is calculated to estimate the optimal accuracy of the *multi-step probabilistic provenance inference* method for that particular processing step. This process is repeated for all processing steps. Please note that, the computed $P(\lambda_{i+1})$ distribution will be used in the next processing step, involving P_{k+1}, to compute the corresponding specific distributions such as $P({\alpha_{i+1}}^{k+1})$ and $P({\beta_{i+1}}^{k+1})$.

In the following sections, we discuss the mechanism of computing the aforesaid distributions to estimate the accuracy. Since the mechanism of computing $P({\alpha_i}^k)$ and $P({\beta_i}^k)$ distributions has been discussed in detail in Section 5.7, we explain this mechanism briefly in this chapter. Applying the aforesaid process, we also report the estimated accuracy of the given processing chain as shown in Figure 6.2.

6.8.1 Computing $P(\lambda_{i+1})$ Distribution

As defined in Section 6.2, a sampling interval distribution, $P(\lambda_{i+1})$, of a view V_{i+1} refers to the probability distribution of the amount of time between two successive tuples insertion into the view V_{i+1}. As already mentioned, the sampling interval distribution of the input view of the workflow is given. Therefore, V_{i+1} refers to a non-persistent, intermediate view produced by a computing processing element P_k which takes a view V_i as an input. The sampling interval of the view V_{i+1} depends on the following two parameters.

1. *Processing delay* δ_k*:* The amount of time required to process the tuples within a window defined over the input view V_i by the computing processing element P_k is referred to as the processing delay of P_k, denoted as δ_k. After the processing, P_k produces a new tuple in its output view V_{i+1}. Therefore, the processing delay of two successive executions of P_k influences the sampling interval distribution $P(\lambda_{i+1})$.

2. *Trigger interval* TR_k*:* The amount of time between two successive executions of P_k is referred to as the trigger interval, denoted as TR_k. After each execution of P_k, a new tuple is inserted into its output

view V_i. Therefore, TR_k also has the influence over the sampling interval distribution $P(\lambda_{i+1})$.

The sampling interval, λ_{i+1}, is the distance in time between two successive tuples in view V_{i+1}. We need to consider the trigger interval and the processing delay of P_k in two successive executions to calculate λ_{i+1}. The processing delay at the previous and current execution of P_k are denoted as ${\delta_k}^p$ and ${\delta_k}^c$, respectively. Based on these parameters, the formula to calculate the corresponding λ_{i+1} is given below.

$$\lambda_{i+1} = TR_k - {\delta_k}^p + {\delta_k}^c \tag{6.6}$$

The value of TR_k is explicated in the workflow provenance. The values of both ${\delta_k}^p$ and ${\delta_k}^c$ actually represent the different values of δ_k based on the distribution $P(\delta_k)$. According to Equation 6.6, we can calculate possible values of λ_{i+1} by putting the given value of TR_k and different values of δ_k at respective places based on the given distribution $P(\delta_k)$.

It is also possible to compute the probability of all possible values of λ_{i+1}, i.e., $P(\lambda_{i+1})$, by facilitating the given $P(\delta_k)$ distribution. The probability of $\lambda_{i+1} = z$, $P(\lambda_i = z)$, is the sum of the product of the probabilities of processing delays δ_k resulting in a distance of z for a given trigger interval TR_k. The formula is given below.

$$P(\lambda_{i+1} = z) = \sum_{x=1}^{max(\delta_k)} P(\delta_k = x) \times P(\delta_k = z - TR_k + x) \tag{6.7}$$

Equation 6.7 calculates $P(\lambda_{i+1})$, i.e., the probability of all possible λ_{i+1} values, by facilitating $P(\delta_k)$ distribution and TR_k. Table 6.1 shows different λ_5 values and their corresponding probability based on Equation 6.7 for the workflow, shown in Figure 6.2.

Table 6.2 shows the comparison between the computed $P(\lambda_5)$ distribution based on Equation 6.7 and the corresponding observed distribution which is collected during the execution of the computing processing element P_5. As it can be seen from Table 6.2, the computed $P(\lambda_5)$ distribution is similar to the observed one which shows the soundness of the computation method.

Table 6.1: Probability of different values in $P(\lambda_5)$ Distribution

$\delta_5{}^p = x$	$P(\delta_5{}^p = x)$	$\delta_5{}^c = y$	$P(\delta_5{}^c = y)$	TR_5	$\lambda_5 = z$	$P(\lambda_5 = z)$
1	0.665	1	0.665	5	5	0.442
1	0.665	2	0.335	5	6	0.223
2	0.335	1	0.665	5	4	0.223
2	0.335	2	0.335	5	5	0.112

Table 6.2: Observed vs. Computed $P(\lambda_5)$ Distribution

$\lambda_5 = z$	Observed $P(\lambda_5 = z)$	Computed $P(\lambda_5 = z)$
4	0.216	0.223
5	0.568	0.554
6	0.216	0.223

6.8.2 Computing $P(\alpha_i{}^k)$ and $P(\beta_i{}^k)$ Distributions

The major phase of estimating the accuracy of the multi-step probabilistic provenance inference method is to compute $P(\alpha_i{}^k)$ and $P(\beta_i{}^k)$ distributions. Computing these distributions follow the same approach as discussed in Section 5.7. Therefore, in this section, we discuss the computation process briefly.

First, we compute $P(\alpha_i{}^k)$ distribution which refers to the probability of all possible values indicating the distance in time between start of a window defined over view V_i and the arrival of the first tuple within that window. Computation of $P(\alpha_i{}^k)$ distribution is accomplished by constructing a *tuple-state graph*, G_α, by facilitating Markov chain model [16]. Each vertex in G_α represents a state, which identifies the position of a tuple within a window w.r.t. the start of the window, defined over the input view V_i of the computing processing element P_k. There are two different types of states in the *tuple-state graph*, G_α. These are:

1. *First states*: These states represent that the current tuple is the first tuple of a particular window. These are denoted as the arrival timestamp of the tuple in the window w.r.t the start of the window, followed by a letter 'F' (e.g. 0F, 1F, 2F). In this case, the arrival timestamps indicate the *first-tuple appearance interval* as discussed in Section 6.2.

2. *Intermediate states*: These states represent the arrival of tuples within a window without being the first tuple. The states are represented by the arrival timestamp of the new tuple in the window w.r.t the start of the window, followed by a letter 'I' (e.g. 1I, 2I, 3I, 4I).

The *tuple-state graph* G_α has a set of vertices and directed edges, i.e., $G_\alpha = (V_\alpha, E_\alpha)$. First, a set of *first* and *intermediate* states are added as vertices/nodes based on Equation 5.5. Next, directed edges are created between these vertices, indicating transitions from one state to another based on the sampling interval of the input view V_i. These transitions are classified into two types: i) transitions within the window boundary (e.g. from '0F' to '1I', from '1I' to '3I' etc.), ii) transitions crossing the window boundary (e.g. from '4I' to '2F'). Furthermore, each directed edge, representing a transition, has a *weight* value refers to the probability of this transition based on $P(\lambda_i)$ distribution. The formulas to create transitions both within the window boundary and crossing the window boundary are given in Equation 5.6 and 5.7, respectively.

The initial state probabilities are uniformly distributed when the *tuple-state graph* G_α, is constructed based on the window size defined over the input view of the given workflow, i.e., view V_4 in Figure 6.2. However, the initial state probabilities of the Markov model for the *first states* are not uniformly distributed when the *tuple-state graph* is constructed based on the window size defined over a non-persistent view V_{np}, produced by a computing processing element P_{int1}. In this case, the initial probability of being in a first state 'xF' can be 0 if there is no possibility of getting a specific distance measured between the start of a window and the first arrival of a tuple in the window. In particular, if the distance x can not be constructed between the *transaction time* of a tuple in V_{np}, i.e., the sum of c_1th triggering point of P_{int1} and δ_{int1}, and the start of a window defined over V_i for the c_2th trigger of the next computing processing element,

P_{int2}, in the chain, the probability of being in state 'xF' becomes 0. The formula of the distance x is given below.

$$x = (c_1 \times TR_{int1} + \delta_{int1}) - (c_2 \times TR_{int2} - {WS_i}^{int2})$$

The long-term behavior of a Markov chain enters a steady state, i.e., the probability of being in a state will not change with time [47]. In the steady state, the vector s_α represents the average probability of being in a particular state based on the *tuple-state graph* G_α. To optimize the steady state calculation, vertices with no incoming edges are discarded.

Following the aforesaid mechanism, we construct the *tuple-state graph* G_α by facilitating the $P(\lambda_4)$ distribution, the window size ${WS_4}^5$ and the trigger interval TR_5, explicated in Figure 6.2. The resulting graph is the same as the *tuple-state graph* shown in Figure 5.5 because of using the same set of parameters. Based on G_α, we can calculate the corresponding steady-state vector s_α. To compute $P({\alpha_4}^5)$ distribution, we only consider the probabilities of states with suffix 'F' from the vector s_α. The resulting $P({\alpha_4}^5)$ distribution is same as it has been shown in Table 5.2.

Along the lines of computing $P({\alpha_4}^5)$ distribution discussed above, $P({\beta_4}^5)$ distribution indicating the probability distribution on the distance between the last tuple in a window and the end of the window can be calculated. In this case, a *tuple-state graph* G_β has to be constructed which has the following two states.

1. *Intermediate states*: These states represent the arrival of tuples within a window without being the last tuple of that window. The states are represented by the arrival timestamp of the new tuple in the window w.r.t the start of the window, followed by a letter 'I' (e.g. 0I, 1I, 2I, 3I, 4I).

2. *Last states*: These states represent that the current tuple is the last tuple of a particular window. These are denoted as the arrival timestamp of the tuple in the window w.r.t the start of the window, followed by a letter 'L' (e.g. 2L, 3L, 4L). In this case, the arrival timestamps indicate the *last-tuple disappearance interval* as discussed in Section 5.2.

Like G_α, G_β has a set of vertices and directed edges, i.e., $G_\beta = (V_\beta, E_\beta)$. The vertices are added based on Equation 5.8. Afterward, directed edges, representing transitions between one vertex to another, are added. There

could be three different sets of directed edges in G_β. The first set includes directed edges connecting two points (states) within the same window. The second set includes directed edges representing that the current tuple is the last tuple in the window. The third set includes directed edges representing the transitions crossing the window boundary. The formulas to add directed edges in G_β is given in Equation 5.9, 5.10 and 5.11.

After constructing G_β, we calculate the steady-state vector s_β. Next, we only consider the probabilities of states with suffix 'L' from the vector s_β to compute the $P({\beta_4}^5)$ distribution. The resulting distribution is same as it has been shown in Table 5.3. The detailed mechanism of computing these specific distributions has been discussed in Section 5.7.2.

6.8.3 Estimating Accuracy

After computing $P({\alpha_i}^k)$ and $P({\beta_i}^k)$ distributions, the multi-step probabilistic provenance inference method calculates a joint probability distribution between the given $P(\lambda_i)$, $P(\delta_k)$ and the computed $P({\alpha_i}^k)$ and $P({\beta_i}^k)$ distributions based on *Failure Condition* 5.2 and 5.3 indicating situations where wrong provenance can be inferred. *Failure Condition* 5.2 indicates that a contributing input tuple could be excluded if the following condition holds: ${\alpha_i}^k < \delta_k$. *Failure Condition* 5.3 indicates that a non-contributing input tuple could be included if the following condition holds: $\lambda_i - {\beta_i}^k < \delta_k$.

Furthermore, the proposed method also facilitates an *offset* value while estimating the accuracy. The *offset* value refers to the distance in time between the *upper bound* of a reconstructed window and the *transaction time* of the output tuple produced due to that window execution. Therefore, $0 \leqslant \mathit{offset} \leqslant max(\delta_k)$, where $max(\delta_k)$ refers to the maximum value of the random processing delay of P_k. For a given *offset* value, we can calculate the estimated accuracy by using the joint probability distribution as mentioned above. Therefore, we will choose a *offset* value such that the resulting estimated accuracy would be the maximum one. The formula to calculate the estimated accuracy is given in Equation 5.12.

According to Equation 5.12, setting the *offset* to 0, returns the estimated accuracy of 30% by facilitating the given $P(\lambda_4)$, $P(\delta_5)$ (see Figure 6.2) and the computed $P({\alpha_4}^5)$ and $P({\beta_4}^5)$ distributions. If we set the *offset* to 1, it returns 84% estimated accuracy based on Equation 5.12. Therefore, we conclude that for the first processing step involving P_5, the estimated accuracy of the multi-step probabilistic provenance inference method is 84%.

Following the same approach, the *multi-step probabilistic provenance inference* method can estimate the accuracy for other processing steps shown in Figure 6.2. It estimates about 95% accuracy for the processing step involving P_6 while it provides an estimation of about 97% accuracy for the processing step involving P_7. Considering the complete processing chain, the maximum achievable accuracy could be 84% since this is the minimum estimated accuracy among all three step-by-step estimated accuracy.

6.9 EVALUATION

The *multi-step probabilistic provenance inference* method is evaluated based on the workflow presented in Section 6.1. The shaded part in the Figure 6.1 is considered for this evaluation. In the given workflow, there are 3 computing processing elements, as shown in Figure 6.1. Each of these computing processing elements takes a view as an input and produces another view as an output. Since P_5 is the first computing processing element in the given processing chain, the input view of P_5, V_4 is the input view of the given processing chain. P_7 is the last computing processing element in the given processing chain. Therefore, the output view of P_7, V_7 is the output view of the given processing chain. As already mentioned in Section 6.1, both input and output views of the given processing chain are persistent. The collection of tuples in these views (V_4 and V_7) is referred to as *sensor data*. The other two views contain intermediate results and hence are non-persistent views. In this section, we describe the evaluation criteria, methods, dataset, test cases and also report the evaluation results.

6.9.1 Evaluation Criteria and Methods

Criteria

The *multi-step probabilistic provenance inference* method is evaluated based on these criteria: i) storage consumption, ii) accuracy and iii) precision and recall. The second research question (*RQ 2*) of this thesis is about the challenge of managing fine-grained data provenance under different system dynamics at reduced cost in terms of storage consumption as discussed in Section 1.4. The multi-step probabilistic provenance inference method infers fine-grained data provenance at reduced storage cost under variable processing delay and sampling interval for a given workflow. Therefore, the multi-step probabilistic provenance inference method ad-

dresses *RQ 2* and provides a solution. Since the primary goal of *RQ 2* is to have fine-grained data provenance at reduced storage costs, one of the evaluation criterion is the *storage consumption* by provenance data. Furthermore, the overall goal of the inference-based framework is to provide accurate provenance information under variable system dynamics. Therefore, another evaluation criterion is the *accuracy* of the inferred provenance information. Unlike the other two methods discussed in Chapter 4 and 5, the *multi-step probabilistic provenance inference* method provides an *inferred provenance graph*, showing data dependencies between contributing input tuples and the output tuple. Therefore, it is important to assess the quality of an inferred provenance graph compared to the original one. To do so, we calculate the *precision* and the *recall* of an inferred provenance graph.

Methods

As discussed in Section 4.7.1, we developed two implementations of documenting fine-grained data provenance explicitly. These are: i) *explicit provenance* and ii) *improved explicit provenance* method. The *explicit provenance* and the *improved explicit provenance* method maintain fine-grained data provenance based on the schema diagram shown in Figure 4.6 and 4.7, respectively. Since there are multiple computing processing elements in the given workflow, shown by the shaded part in Figure 6.1, there might be several relations maintaining provenance data for all processing steps. The storage cost of these relations maintaining provenance data is considered as the storage consumption of the *explicit provenance* and the *improved explicit provenance* method. The storage consumption of the *multi-step probabilistic provenance inference* method to maintain provenance data is compared with the *explicit provenance*, the *improved explicit provenance*, the *basic provenance inference* and the *probabilistic provenance inference* method. It may be noted here that both *basic provenance inference* and *probabilistic provenance inference* method require to store the *transaction time* of all tuples in each view including the non-persistent ones to infer fine-grained data provenance. Therefore, these two methods have the same storage consumption as reported in Section 5.9. The *multi-step probabilistic provenance inference* method requires to store the *transaction time* of all tuples in *sensor data*, i.e., tuples in input and output view only, to infer fine-grained data provenance.

The *multi-step probabilistic provenance inference* method is also evaluated in terms of accuracy. The accuracy of an *inferred provenance graph* is measured by facilitating the traditional fine-grained provenance information, also known as *explicit provenance*, as a ground truth. Furthermore, the pre-

cision and the recall of an *inferred provenance graph* is also measured by comparing it to the original provenance graph based on the ground truth.

6.9.2 Dataset

A real dataset[3] measuring electrical conductivity of the water, collected by the RECORD project, discussed in Section 5.1, is used to evaluate the performance of different methods. The experiments are performed on a underlying PostgreSQL 8.4[4] database and the Sensor Data Web[5] platform. The input dataset contains 30000 tuples representing the six-month period from July-December 2009 and requires 7200 KB of storage space.

Besides this real dataset, a simulation using artificial data with variable processing delay and sampling interval is also performed to evaluate the accuracy of the *multi-step probabilistic provenance inference* method. Since this method method can also estimate the accuracy beforehand, the estimated accuracy calculated is also reported in results.

6.9.3 Test cases

The evaluation of the *multi-step probabilistic provenance inference* method is performed based on two sets of test cases. The first set of test cases uses a real dataset, containing 30000 tuples, as discussed in Section 6.9.2. These test cases are used to compare different methods in terms of storage consumption and accuracy. All these test cases are based on the sliding windows. One of these test cases has non-overlapping sliding windows (*S1.Time.1*) while the other has overlapping sliding windows (*S1.Time.2*).

The second set of test cases is introduced to compare the accuracy of the inferred provenance information. These test cases are used in a simulation using artificial data. Therefore, these are not used to evaluate the storage consumption. The simulation is performed for 10000 time units.

Table 6.3 and 6.4 show the window size, trigger interval, processing delay of each computing processing element in the given workflow as shown by the shaded part in Figure 6.1 for every test case in both sets. Moreover, $P(\lambda_4)$, the sampling interval of the input view V_4, is also given for every

3 Available at `http://data.permasense.ch/topology.html#topology`
4 Available at `http://www.postgresql.org/`
5 Available at `http://sourceforge.net/projects/sensordataweb/`

Table 6.3: Test Case Set I : Parameters of Different Test Cases used for the Evaluation using Real Dataset

Test case id	Processing Element P_k	Window size in time units	Trigger interval in time units	$mean/max$ (δ_k) in time units	$mean/max$ (λ_4) in time units
S1.Time.1	P_5	5	5	1/2	2/3
	P_6	8	8	1/2	
	P_7	11	11	1/2	
S1.Time.2	P_5	5	2	1/2	2/3
	P_6	8	2	1/2	
	P_7	11	2	1/2	

test case. For these two test cases shown in Table 6.3 and 6.4, we assume that given sampling interval distribution, $P(\lambda_4)$, and all processing delay distributions- $P(\delta_5)$, $P(\delta_6)$, $P(\delta_7)$, follow Poisson distribution. The *mean* value and *max* value of these distributions are also reported in Table 6.3 and 6.4.

The 4 test cases shown in Table 6.4 are chosen in such a way that each of them has some variety in their parameters compared to the others. The difference between test case *S2.Time.1* and test case *S2.Time.2* is that they have different values for *mean* and *maximum* processing delay for all computing processing elements. Apparently, test case *S2.Time.1* has smaller processing delays than test case *S2.Time.2*. Test case *S2.Time.3* and *S2.Time.4* are almost similar to each other except the parameters of sampling interval of the input view V_4, λ_4. In test case *S2.Time.3*, $mean(\lambda_4)$ and $max(\lambda_4)$ are 2 and 3 time units, respectively where as in test case *S2.Time.4*, $mean(\lambda_4)$ and $max(\lambda_4)$ are 3 and 5 time units, respectively. Therefore, the input tuples arrive faster in test case *S2.Time.3* than test case *S2.Time.4*.

Table 6.4: Test Case Set II : Parameters of Different Test Cases used for the Evaluation using Simulation

Test case id	Processing Element P_k	Window size in time units	Trigger interval in time units	mean/max (δ_k) in time units	mean/max (λ_4) in time units
S2.Time.1	P_5	6	6	1/2	2/3
	P_6	10	10	1/2	
	P_7	14	14	1/2	
S2.Time.2	P_5	6	6	2/3	2/3
	P_6	10	10	2/3	
	P_7	14	14	2/3	
S2.Time.3	P_5	7	5	1/2	2/3
	P_6	13	11	1/2	
	P_7	23	17	1/2	
S2.Time.4	P_5	7	5	1/2	3/5
	P_6	13	11	1/2	
	P_7	23	17	1/2	

6.9.4 Storage Consumption

The storage consumption by different methods to maintain fine-grained data provenance is one of the major evaluation criteria. We compare the storage consumption among *explicit provenance*, *improved explicit provenance*, *basic provenance inference*, *probabilistic provenance inference* and *multi-step probabilistic provenance inference* method by facilitating the test case set I, described in Table 6.3. In the result, the storage taken by the *sensor data*, collection of both input and output data products, is also reported for all test cases.

Figure 6.7 shows the storage consumption by different methods for test cases in test case set I, shown in Table 6.3. Test case *S1.Time.1* has non-

overlapping, time-based window for all computing processing elements- P_5, P_6 and P_7. Please note that, all computing processing elements participating in the workflow, have *input-output ratio* $= n : 1$ ('many to one'), i.e., a computing processing element takes n tuples in the window as input and produces 1 tuple as an output. View V_4 is the input view of the workflow which is processed by P_5 and it produces around 12000 tuples. View V_5 holds these tuples and these are processed further by P_6. P_6 produces around 7500 tuples which are held by view V_6. Eventually, P_7 process these tuples and produces around 5500 output tuples, which are inserted into view V_7. All these computing processing elements start executing at the same time.

Non-overlapping windows

The *explicit provenance* method documents fine-grained data provenance for every tuples produced by computing processing elements. In the non-overlapping case, this method has to store $30000 + 12000 + 7500 = 49500$ tuples maintaining provenance data. It takes around 3000 KB of storage space. The *improved explicit provenance* method takes a little more space than the *explicit provenance* method which is around 3200 KB of storage space. In non-overlapping case, a particular input tuple does not contribute several times. Therefore, *improved explicit provenance* method cannot compress the storage required by provenance data. Both *basic provenance inference* and *probabilistic provenance inference method* have to keep the *transaction time* of all tuples in the input, intermediate and output views. The total number of tuples are $30000 + 12000 + 7500 + 5500 = 55000$. Both methods take around 1100 KB of storage. *Multi-step probabilistic provenance inference* method has to only keep the *transaction time* of tuples only in input and output views. The number of input and output tuples is $30000 + 5500 = 35500$. It takes around 700 KB of storage space. In the non-overlapping case, the *multi-step probabilistic provenance inference* method consumes around 23%, 22% and 63% of the storage space required by the *explicit provenance*, the *improved explicit provenance* and the other inference-based methods to maintain provenance information, respectively.

The other test case, *S1.Time.2*, shown in Table 6.3, has overlapping, time-based windows for all computing processing elements- P_5, P_6 and P_7. The *input-output ratio* of all computing processing elements is $= n : 1$. View V_4 is the input view of the workflow which is processed by P_5. Since the average sampling interval of V_4 is 2 time units and trigger interval of P_5 is also 2 time units, it can be said that upon the insertion of every tuple in V_4, P_5 is executed. Therefore, P_5 produces around $30000 \div 1 = 30000$ tuples.

Overlapping windows

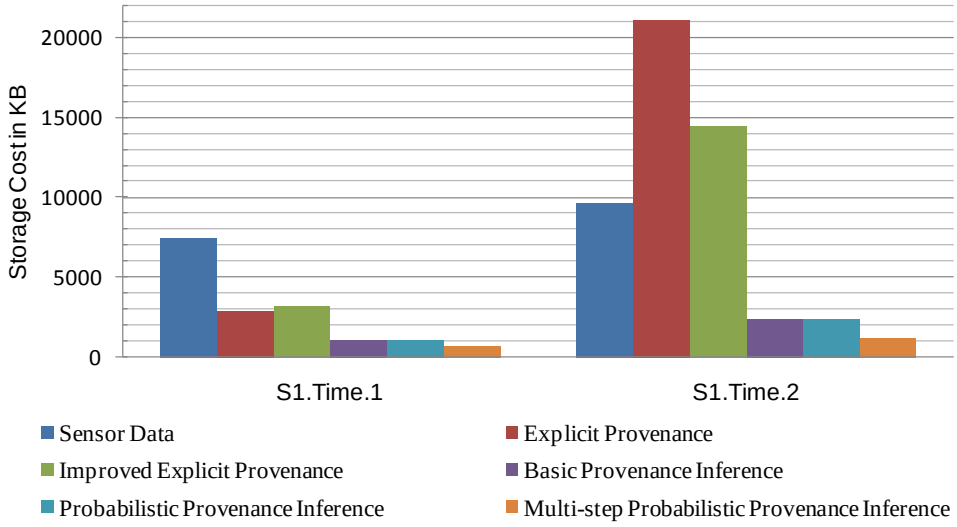


Figure 6.7: Comparison of Storage Consumption among different methods using test case set I

View V_5 holds these tuples and these are processed further by P_6. P_6 also produces around 30000 tuples which are held by view V_6. Eventually, P_7 process these tuples and because of the trigger interval of 2 time units, it also produces around 30000 output tuples, which are inserted into view V_7.

In the test case *S1.Time.2*, the *explicit provenance* method has to store around 360000 tuples maintaining provenance data. It takes more than 21000 KB of storage space. The *improved explicit provenance* method takes less space than the *explicit provenance* method which is around 15000 KB of storage space. In the overlapping case, a particular input tuple contributes several times to produce output tuples. Therefore, the *improved explicit provenance* method can compress the storage required by provenance data. Both *basic provenance inference* and *probabilistic provenance inference method* have to keep the *transaction time* of all tuples in input, intermediate and output views. The total number of tuples is around 120000. Both methods take around 2400 KB of storage space. *Multi-step probabilistic provenance inference* method has to only keep the *transaction time* of only input and output tuples. The number of input and output tuples is around $30000 + 30000 = 60000$ tuples which requires around 1200 KB of storage space. In this test case, the *multi-step probabilistic provenance inference* method consumes around 6%, 8% and 50% of the storage space

required by the *explicit provenance*, the *improved explicit provenance* and the other inference-based methods to maintain provenance information, respectively.

Please note that the reported ratio depends on the window size, trigger specification and other parameters as shown in Table 6.3. If the window size is larger and there is a big overlap between subsequent windows, the *multi-step probabilistic provenance inference* method performs even better.

6.9.5 Accuracy

The accuracy of the *multi-step probabilistic provenance inference* method is measured by comparing an *inferred provenance graph* with the original provenance graph constructed from explicitly documented provenance information. For a particular output tuple, if these two graphs match exactly with each other then the accuracy of the inferred provenance information for that output tuple is 1 otherwise, it is 0. We calculate the average of the accuracy for all output tuples produced by a given test case, known as *average accuracy*. If there are n number of output tuples for a given test case and $\mathtt{accuracy}_i$ represents the accuracy (either 0 or 1) of i^{th} output tuple then *average accuracy* is expressed as:

$$\textit{Average accuracy} = (\frac{\sum_{i=1}^{n} \mathtt{accuracy}_i}{n} \times 100)\%$$

We facilitate both test case set I and II to evaluate the average accuracy of the *multi-step probabilistic provenance inference* method. Furthermore, we also report the estimated accuracy of *multi-step probabilistic provenance inference* method. The estimated accuracy is measured based on the mechanism discussed in Section 6.8. Table 6.5 shows estimated and average accuracy for each test case. Moreover, Table 6.5 also shows the achieved accuracy of the *probabilistic provenance inference* and the *basic provenance inference*. In all test cases, the multi-step probabilistic provenance inference method provides the same level of accuracy as the probabilistic provenance inference method at lower storage costs. Next, we discuss some of the observations based on the result presented in Table 6.5.

Test case *S1.Time.1* and *S1.Time.2* have the same parameters except the trigger interval of computing processing elements. The multi-step probabilistic provenance inference method achieves 83% and 84% accuracy for test case *S1.Time.1* and *S1.Time.2*, respectively. Next, we compare test case

Table 6.5: Comparison of Accuracy between Different Inference-based Methods

Test case id	Multi-step Probabilistic (Estimated)	Multi-step Probabilistic (Average)	Probabilistic	Basic
S1.Time.1	84%	83%	83%	38%
S1.Time.2	86%	84%	84%	36%
S2.Time.1	83%	84%	84%	39%
S2.Time.2	61%	65%	67%	28%
S2.Time.3	86%	84%	84%	41%
S2.Time.4	92%	93%	93%	61%

S1.Time.1 and *S2.Time.1*. Both test cases have non-overlapping windows of different size. Though window size defined over views for different computing processing elements do not match with each other, the level of accuracy of inferred provenance is almost similar in these two test cases. The multi-step probabilistic provenance inference method achieves 83% and 84% accuracy for test case *S1.Time.1* and *S2.Time.1*, respectively. From these comparisons, it seems that both trigger interval and window size have almost no influence over the accuracy. However, experiments with the same window size and trigger interval for all computing processing elements show that the multi-step probabilistic provenance inference achieves 100% accuracy irrespective of other parameters such as the processing delay and the sampling interval. It also happens when the window size and the trigger interval of all computing processing elements has a common factor between them. Since the triggering points of different processing elements will be aligned with each other in these cases, there is a very low chance to have different provenance graph for a selected output tuple and this could increase the level of accuracy.

Influence: window size and trigger interval

Based on Table 6.5, the multi-step probabilistic provenance inference method achieves only 65% accuracy for test case *S2.Time.2*. This test case has the same parameters like test case *S2.Time.1* except the processing delay of participating computing processing elements. Computing processing elements in test case *S2.Time.2* take more time to process input tuples than test case *S2.Time.1*. The multi-step probabilistic provenance infer-

Processing delay

ence method achieves 65% and 84% accuracy for test case *S2.Time.2* and *S2.Time.1*, respectively. According to *Failure Condition* 5.2 and 5.3, the processing delay takes a part determining whether wrong provenance can be inferred or not and if it increases the chance of inaccuracy also increases. Therefore, it seems reasonable to conclude that *the higher the processing delay, the lower the chance of achieving accurately inferred provenance*.

Sampling interval

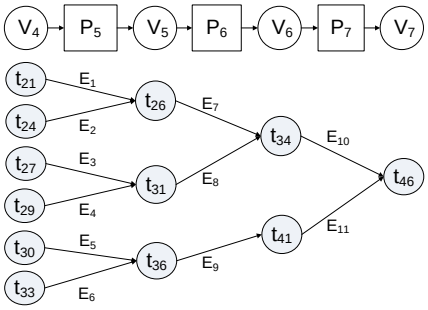
At last, we compare the accuracy of multi-step probabilistic provenance inference method between test case *S2.Time.3* and *S2.Time.4*. These two test cases have exactly the same parameters except the sampling interval of the input view V_4. In test case *S2.Time.3*, input tuples are inserted more frequently than in test case *S2.Time.4*. Table 6.5 shows that the accuracy of the method increased by almost 10% in test case *S2.Time.4*, compared to the accuracy achieved in test case *S2.Time.3*. In *Failure Condition* 5.2 and 5.3, we can also see that value of λ_i takes a part to decide whether the inference would provide accurate or inaccurate provenance. Based on *Failure Condition* 5.2 and 5.3, keeping δ_k values the same and increasing λ_i values would definitely decrease the chance of a failure. Therefore, this analysis might give a useful indication that *the higher the sampling time, the higher the chance of achieving accurately inferred provenance*.

Furthermore, we can make another observation from the result reported in Table 6.5. The estimated accuracy of the multi-step probabilistic provenance inference method is almost similar to the achieved average accuracy of the method. Since the estimated accuracy can be calculated before the actual experiment, it is a useful indicator for the applicability of multi-step probabilistic provenance inference method for a given test case.

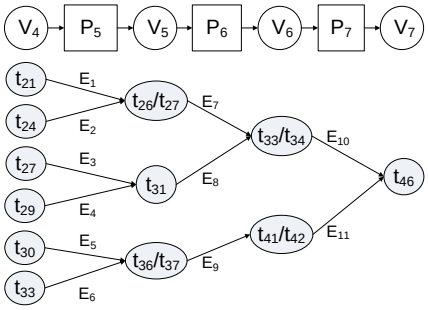
6.9.6 Precision and Recall

The *multi-step probabilistic provenance inference* method infers a fine-grained data provenance graph. If one of the edges in the inferred provenance graph does not match with the original provenance graph, the accuracy of that particular inferred provenance graph becomes 0. Therefore, we introduce precision and recall of the inferred provenance graph to have a finer criterion than the accuracy.

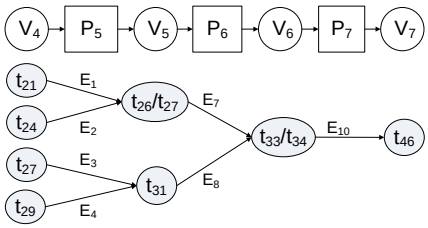
Precision and recall are widely chosen measures for evaluating the performance of information retrieval systems [9]. Precision is the fraction of the retrieved documents that are relevant to the user's information need. Recall is the fraction of the documents relevant to a query that are success-



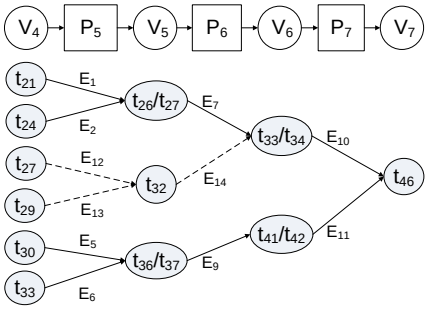
(a) Original Provenance Graph



(b) Example of an Inferred Provenance Graph with $\mathtt{precision} = 100\%$ and $\mathtt{recall} = 100\%$



(c) Example of an Inferred Provenance Graph with $\mathtt{precision} = 100\%$ and $\mathtt{recall} = 64\%$



(d) Example of an Inferred Provenance Graph with $\mathtt{precision} = 73\%$ and $\mathtt{recall} = 73\%$

Figure 6.8: Example of Inferred Provenance Graphs with $\mathtt{precision}$ and $\mathtt{recall}$ values

fully retrieved. Both precision and recall are used to quantify the quality of the result of a query.

We adopt the aforesaid definition of precision and recall to assess the quality of an *inferred provenance graph*, comparing it to the corresponding *original provenance graph*. In this case, we consider the edges between vertices which represent data dependences between two tuples. Assume that, I be the set of edges in an *inferred provenance graph* and O be the set of edges in the corresponding *original provenance graph*. Precision and recall can be calculated using the following formula.

$$\mathtt{precision} = (\frac{|I \cap O|}{|I|} \times 100)\% \qquad \mathtt{recall} = (\frac{|I \cap O|}{|O|} \times 100)\%$$

Table 6.6: Average Precision and Average Recall of Multi-step Probabilistic Provenance Inference

Test case id	Average Precision	Average Recall
S1.Time.1	87%	98%
S1.Time.2	85%	97%
S2.Time.1	88%	96%
S2.Time.2	86%	93%
S2.Time.3	90%	97%
S2.Time.4	93%	96%

Example

Figure 6.8 shows examples of *inferred provenance graph* with *precision* and *recall* values and the corresponding *original provenance graph*. Figure 6.8a depicts the *original provenance graph* for an output tuple t_{46} in view V_7. The original provenance graph is drawn based on the documented explicit fine-grained data provenance. Each edge in the original provenance graph is labeled such as E_1, E_2 etc. No two edges can have the same label unless their *start* and *end* vertex are the same. In the *original provenance graph*, there are 11 edges. Therefore, $|O| = 11$. Figure 6.8b shows an example of an *inferred provenance graph* for the same output tuple t_{46} in view V_7. Since view V_5 and V_6 in the given workflow are non-persistent views, the *multi-step probabilistic provenance inference* method can infer only a set of possible *transaction times* for a tuple in a non-persistent view based on the appropriate processing delay distribution. As an example, in view V_5, a tuple could exist either at time 26 or at time 27. Therefore, there would be two versions of the corresponding edge E_1. When comparing it with the original provenance graph, we keep the edge which exactly matches with that in the original provenance graph and remove the other edge. The edge is labeled with the same value which is found in the original provenance graph. After finding appropriate versions of edges, we get the *inferred provenance graph* shown in Figure 6.8b. In this case, $|I| = 11$ and $|I \cap O| = 11$. Therefore, this inferred provenance graph has $\texttt{precision} = 100\%$ and $\texttt{recall} = 100\%$. It is the accurate inferred provenance graph for the selected output tuple.

Figure 6.8c shows another example of an inferred provenance graph for the same output tuple t_{46} in view V_7. This inferred provenance graph does not have a few edges compared to the original provenance graph. However, existing edges in this graph match exactly with corresponding edges in the original provenance graph, shown in Figure 6.8a. It has $\mathtt{precision} = 100\%$ and $\mathtt{recall} = 64\%$. The last example of an inferred provenance graph shown in Figure 6.8d has $\mathtt{precision} = 73\%$ and $\mathtt{recall} = 73\%$. The dotted edges in this graph, shown in Figure 6.8d, are incorrect ones based on the original provenance graph.

In the light of the aforesaid example, we calculate *precision* and *recall* for each output tuple and then compute the *average precision* and *average recall* per test case. Table 6.6 shows average precision and average recall for different test cases. In all test cases, *average recall* is higher than *average precision*. It means that the inferred provenance graph may contain some extra edges which are not present in the original one. However, values of both *precision* and *recall* in all test cases suggest that the quality of an inferred provenance graph is high and it could be very useful to scientists.

6.10 DISCUSSION

Like the other inference-based methods discussed in Chapter 4 and 5, the multi-step probabilistic provenance inference method has the same set of requirements to be satisfied, discussed briefly in Section 6.4. If these requirements are not fulfilled by the underlying system, the multi-step probabilistic provenance inference method cannot be applied. Moreover, to address activities with variable input-output ratio, the proposed method follows the same approach taken by the other inference-based methods as discussed in Section 4.8.

Long processing chain

The multi-step probabilistic provenance inference method is capable of inferring fine-grained data provenance for a given processing chain with several processing steps. In case of a long processing chain with tens or hundreds of steps, the proposed method might perform poorly in terms of accuracy. In this case, the method has to handle the higher magnitude of dynamism introduced in the system by having many processing steps and variable processing delays for each of these steps. Furthermore, a long processing chain requires more information to be explicated in the workflow provenance. In this case, aggregating different steps into a single process-

ing step could minimize the storage costs as well as maximize the accuracy. Therefore, it is recommended to apply the multi-step probabilistic provenance inference method on a reasonably smaller processing chain.

Tuple-based windows

The multi-step probabilistic provenance inference method has been explained based on a processing chain where associated views have time-based windows only. It is possible to extend the proposed method for tuple-based windows too. In this case, the key element of the inference mechanism is the sequence number of the tuples in views rather the timestamps of tuples in case of time-based windows. Since sequence number of the tuples are local to each view, not a global variable like timestamps, the inference mechanism has to also transform the local sequence number into a global one before applying the method. As an example, we assume that a computing processing element P_k has the input view V_i and it produces the output view V_{i+1}. The input-output ratio of P_k is $n : 1$ where n be the window size defined over V_i. A tuple produced by P_k has the sequence number s in view V_{i+1}. This local sequence number s could be transformed into a global sequence number S based on the following formula: $S = (s \times TR_k) + (WS_i{}^k - TR_k) + 1$. In this case, we assume that P_k starts executing once there are $WS_i{}^k$ tuples in view V_i and the window is a non-jumping window. Using the principle of backward computation phase, discussed in Section 6.6, it could be possible to estimate global sequence number of tuples in all views including the intermediate ones. However, a combination of tuple-based and time-based windows in a processing chain could be more tricky to address. In future, we would like to extend the multi-step probabilistic provenance inference method in this direction.

Finally, we would like to integrate the accuracy estimation mechanism of the multi-step probabilistic provenance inference method into the actual inference mechanism to achieve more accurate provenance information.

6.11 SUMMARY

The multi-step probabilistic provenance inference method that can infer fine-grained data provenance at reduced storage costs for a given workflow with multiple processing steps and non-persistent intermediate views. The design and development of this method was motivated by the second research question (*RQ 2*) which mentioned the challenge of managing fine-grained data provenance at reduced storage consumption under different

system dynamics. The proposed method is an extension of the probabilistic provenance inference method, discussed in Chapter 5, to handle multiple processing steps in a workflow with non-persistent, intermediate views.

Like the other inference-based methods discussed in Chapter 4 and 5, the multi-step probabilistic provenance inference method has three phases. First, the workflow provenance is documented. The next two phases are executed only once the user requests provenance for an output data product. In the second phase, the method calculates an *initial tuple boundary*, defined over the input view of the workflow, which includes potential input tuples those might have contributed to produce the selected output data product/tuple. During this phase, the method exploits the values of different properties of the particular computing processing element such as window size, processing delay distribution to construct the *initial tuple boundary*. Finally, the proposed method establishes data dependencies between input and output tuples per processing step till it reaches the selected output tuple. The outcome of the final phase is a *fine-grained data provenance graph*.

The proposed method can also estimate the accuracy of inferred provenance information by facilitating a few given distributions on processing delay and sampling interval. At the time of estimating the accuracy, it involves Markov chain model to compute several specific distributions which are necessary to estimate the level of accuracy.

We evaluated storage consumption and accuracy of the multi-step probabilistic provenance inference method for different test cases. The evaluation shows that the multi-step probabilistic provenance inference method takes less space compared to the explicit provenance collection methods and the other inference-based methods, discussed in Chapter 4 and 5. If the window size and the overlaps between windows is larger, the proposed method performs even better. Furthermore, the method achieves almost similar level of accuracy as the probabilistic provenance inference method (Chapter 5). Therefore, applying the multi-step probabilistic provenance inference method, we can achieve similar accuracy at even reduced storage costs. To quantify the quality of inferred provenance, we also calculated precision and recall of inferred provenance graphs. The average precision and average recall achieved by this method shows us that the inferred provenance graph resembles the original provenance graph very well and hence, the inferred provenance graph could be useful to scientists to trace an unexpected value back to its source.

7

SELF-ADAPTABLE FRAMEWORK

The inference-based framework, discussed in this thesis, should be capable of handling different situations. A particular situation can be often defined as a set of values of related parameters such as processing delay, sampling interval etc. in this particular setting. Depending on values of the aforesaid parameters, situations can change quickly and the inference-based framework has to accommodate these changes to infer accurate provenance at a comparatively lower storage costs.

We presented different components of the framework that can infer data provenance at reduced costs in terms of storage consumption and time. In Chapter 3, we described the *workflow provenance inference* method which can infer workflow provenance, i.e., data dependencies between operations/activities, from the source code of a given scientific model. Inference of workflow provenance allows scientists to capture workflow provenance of their scientific model automatically which in turn saves significant amount of time. Later, the workflow provenance of a scientific model is facilitated by inference-based methods which can infer fine-grained data provenance. These inference-based methods were described in Chapter 4, 5 and 6. As explained in Chapter 4, the *basic provenance inference* method can infer accurate fine-grained data provenance for offline (non-stream) data. It also infers accurate provenance for data streams if the amount of time required to process input data products/tuples, also referred to as processing delay, is always constant. However, processing delay could vary depending on the system workload as well as types of operations/activities. In Chap-

This chapter is based on part of the work: An Inference-based Framework to Manage Data Provenance in Geoscience Applications. Accepted in *IEEE Transactions on Geoscience and Remote Sensing*, IEEE Geoscience and Remote Sensing Society, 2013. (Impact Factor: 2.895)

ter 5, we presented the *probabilistic provenance inference* method which infers fine-grained data provenance under variable system dynamics such as processing delay and tuple arrival pattern, also referred to as sampling interval. It facilitates appropriate processing delay and sampling interval distributions to infer optimally accurate fine-grained data provenance. However, this method is suitable for a processing step with persistent input. The *multi-step probabilistic provenance inference* method extends the previous approach and can infer fine-grained data provenance for an entire workflow, consisting of several processing steps with non-persistent, intermediate results under variable system dynamics, as discussed in Chapter 6.

Challenge

We have developed these inference-based methods to handle different situations, i.e., offline data/data streams and constant/variable system dynamics. However, the framework cannot decide autonomously to choose the appropriate inference-based method based on the characteristics of a given scientific model and the underlying execution environment. This challenge is identified and mentioned in the third research question (*RQ 3*) of this thesis, as discussed in Section 1.4. This research question focuses on how to incorporate the self-adaptability into the provenance inference framework, discussed in this thesis.

Self-adaptable framework

Since there could be variations in the underlying system dynamics and model characteristics, self-adaptability of the framework becomes a major concern. Self-adaptability allows the proposed inference-based framework to select the most appropriate inference-based method based on some key characteristics. Furthermore, a self-adaptable framework is also capable of monitoring the execution environment continuously so that changes in any key characteristic can be accommodated promptly. A framework inferring data provenance without self-adaptability cannot address any changes in system dynamics such as processing delay, sampling interval etc. and in turn, it could infer inaccurate provenance information. The self-adaptable framework can infer highly accurate provenance information with minimal guidance and intervention from developers side.

In this chapter, we describe the key characteristics of a scientific model which needs to be considered to incorporate self-adaptability into the framework. Based on these key characteristics, we present a decision tree which is facilitated to take the decision of the most appropriate inference-based method based on current system dynamics and model characteristics.

Chapter Structure

The structure of this chapter is as follows. First, we discuss the key characteristics of a scientific model which are considered to achieve a self-

adaptable system. Next, we describe and explain the decision making process to select the most suited inference-based method. Finally, we discuss the potential of assessing applicability of inference-based methods in terms of storage costs and accuracy during a decision making process.

7.1 KEY CHARACTERISTICS OF A SCIENTIFIC MODEL

In Section 1.3, we discussed the characteristics of different entities associated with a scientific model at both design and execution phase. Some of these characteristics are considered and explored to achieve a self-adaptable, inference-based provenance management framework. In this section, we describe these key characteristics briefly and also point out the reasons which make them important. The key characteristics are given in the following.

- *Model developing platform:* A scientific model could be developed either in a platform which collects provenance automatically or in a platform that has no provenance support. The former is referred to as *provenance-aware* platform, while the later is referred to as *provenance-unaware* platform. Workflow engines such as Kepler [84], Karma2 [116], Taverna [102], VisTrails [24] are examples of provenance-aware platforms. General purpose programming languages like Python[1], general purpose data manipulation tools such as Microsoft Excel[2], R[3] etc. are examples of provenance-unaware platforms. Depending on the model developing platform, the envisioned self-adaptable framework can decide whether to apply the *workflow provenance inference* method or not. Since scientific models developed in a provenance-unaware platform has no corresponding workflow provenance, the workflow provenance inference method, discussed in Chapter 3, is applied over the scientific model.

- *Type of activities:* A scientific model is comprised of several activities/operations. Workflow provenance of a scientific model not only documents data dependencies between activities but also annotates a number of properties for each activity. Based on these properties,

1 Available at `http://www.python.org/`
2 Available at `http://office.microsoft.com/en-us/excel/`
3 Available at `http://www.r-project.org/`

we can define different categories of activities. One of these properties is *hasOutput* which defines whether an activity, when executed, produces persistent output views or not. The persistence of a view is indicated by the property *IsPersistent*, as discussed in Section 1.3.1. Since scientist can only request provenance information for a data product that is persistent, inference-based methods which infer fine-grained data provenance, discussed in Chapter 4, 5 and 6, are only applied over activities which produce persistent view (*IsPersistent=true*).

The other property which needs to be considered is *input-output ratio*. It refers to the ratio between the number of input tuples contributed to produce output tuples to the number of produced output tuples. Activities which maintain the same *input-output ratio* through out the execution phase are referred to as *constant ratio* activities and activities which does not maintain the same *input-output ratio* is referred to as *variable ratio* activities. Since inference-based methods, inferring fine-grained data provenance, are directly applicable to *constant ratio* activities, this is another important criterion that should be considered to design a self-adaptable framework.

- *Type of input data:* A scientific model computes over input data products. Input data might arrive continuously (e.g. data streams) during the model execution or it can be collected before the execution begins (e.g. offline data). All inference-based methods inferring fine-grained data provenance are applicable to both data streams and offline data. However, a particular inference-based method could be more suited based on the type of input data products. Therefore, it is another important characteristic which should be considered at the time of deciding the most appropriate inference-based methods.

- *System dynamics:* System dynamics refers to a set of parameters that control the nature of how the data products are arriving into the system for processing and when the input data products are processed. To be more specific, system dynamics depends on two parameters: i) processing delay and ii) sampling interval. Processing delay refers to the amount of time required to complete the execution of a computing processing element or an activity over a set of input data products/tuples. Sampling interval refers to the amount of time between the arrival of two successive input tuples for processing. A computing processing element might be executed several times if data ar-

rives continuously (data streams). In this case, processing delay, i.e., the time required to process input data tuples, could vary from one execution to another because of the current system workload or nature of the processing. This is also true for the sampling interval as well. In case of a data stream, arrival of tuples could be either regular or irregular because of the network delays, broken sensors etc. The parameters showing variation in their values over a time period require special attention to infer fine-grained data provenance. Therefore, both processing delay and sampling interval have to be considered to select the most appropriate inference-based methods.

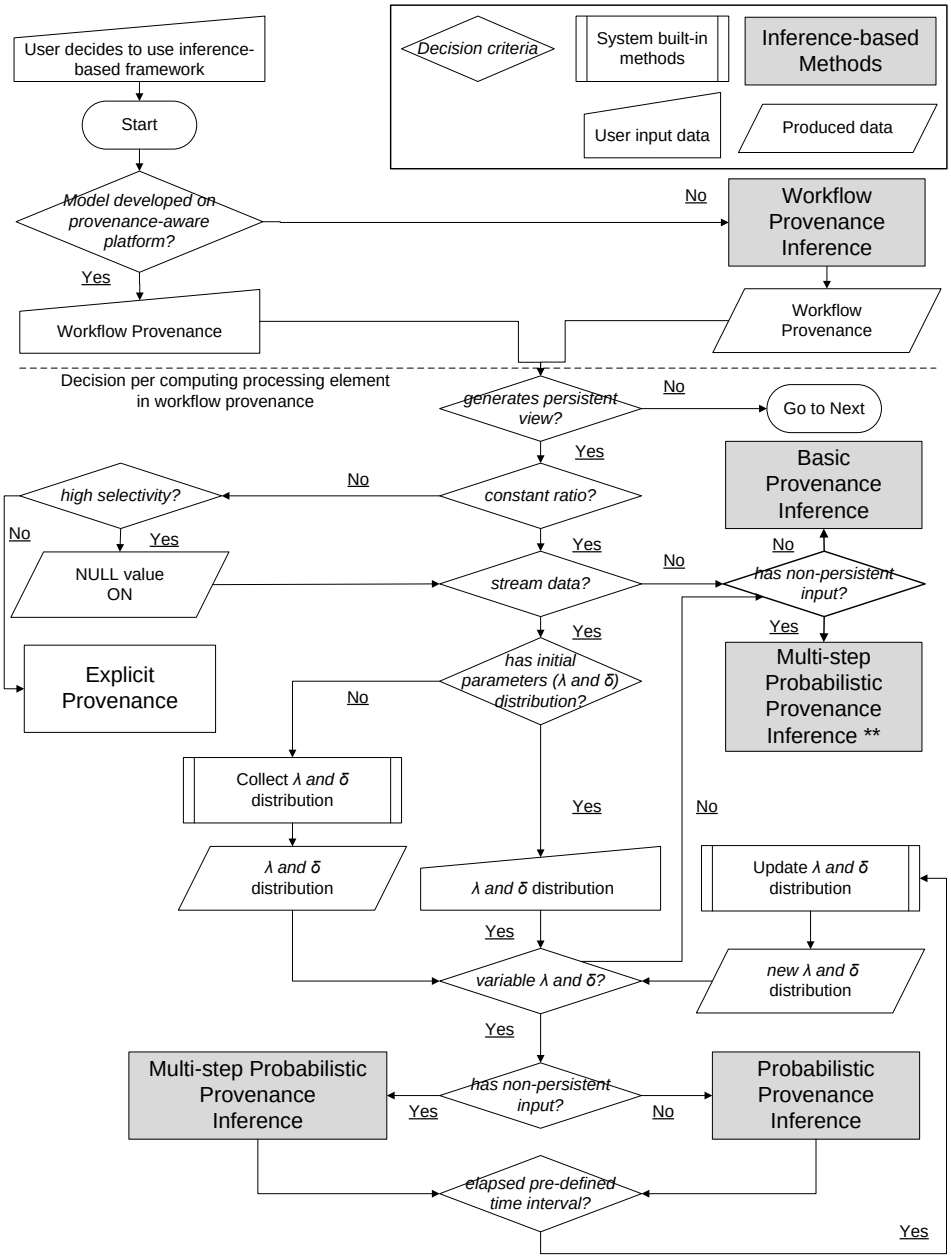
As discussed in Section 1.3, the aforesaid characteristics define a particular scientific model. Since the goal of the self-adaptability is to select the most appropriate inference-based methods based on a given scientific model, these characteristics need to be considered. In a nutshell, developing an inference-based framework to manage both workflow and fine-grained data provenance requires attention to the underlying platform along with the system dynamics including characteristics of processing element/activity and data products. The inference mechanisms should take variation in the used environment, processing delay and data arrival pattern into consideration to infer highly accurate provenance information. To accomplish that, a self-adaptable framework is required which can decide when and how to execute the most appropriate inference-based methods based on a given scientific model and it's associated data products.

7.2 DECISION TREE OF SELF-ADAPTABLE FRAMEWORK

We propose to incorporate self-adaptability into the framework inferring fine-grained data provenance by facilitating a decision tree. The decision tree shown in Figure 7.1 considers the key characteristics discussed in Section 7.1 to select the most appropriate inference-based method for a given scientific model.

Checking model developing platform

The decision making process starts when a scientist decides to use the framework. First, the development platform of the model is considered. If the model is developed using a provenance-aware platform, the workflow provenance graph is readily available. Otherwise, the framework decides to apply the *workflow provenance inference* method to infer the workflow provenance. The documented workflow provenance of the scientific model



** for offline data, with multiple processing steps, a variant of *Multi-step Probabilistic Provenance Inference* method is applied that has constant parameters and hence, no probability distributions.

Figure 7.1: Decision Tree selecting the appropriate inference-based method enabling a self-adaptable framework

is represented as a graph, referred to as *workflow provenance graph*. The annotated characteristics of processing elements/activities in the workflow provenance graph is used in the decision process.

The next phase of the decision making process is executed per computing processing element/activity in the workflow provenance graph. First, the framework considers whether the computing processing element/activity generates a *persistent view* or not. If the particular processing element does not produce a persistent view, the decision making process stops and considers the next processing element since scientists cannot request provenance for an output tuple in a non-persistent view. Otherwise, the framework considers the *input-output ratio* of the given processing element/activity in the next step. If the input-output ratio of the given processing element is variable like *selection* operations in a database, the decision tree then considers the selectivity rate, i.e., the percentage of input data products to be selected for processing within a processing window if the given condition is met. If the processing element has high selectivity rate, then it switches *NULL* value mode *ON* referring to the inclusion of *NULL* data products in the output view if the corresponding input data product is not selected for processing. The inclusion of *NULL* data products in the output view transforms the *input-output ratio* of the activity from *variable* ratio to *constant* ratio ('one to one'). Furthermore, it also ensures that the output data product is created in the same order as the appearance of the contributing input data product satisfying the assumption on *order of tuples in the output view*, as discussed in Section 4.5. Therefore, the framework can apply inference-based methods now. If the computing processing element has low selectivity rate, inclusion of *NULL* data products in the output view will incur more storage overhead and one of the major advantages of using inference-based methods is canceled out. Therefore, the decision making process decides to use *explicit provenance* method for the given model.

Checking type of activities

After checking the selectivity rate, if the framework decides that inference-based methods can be applied on the given model (e.g. high selectivity rate) or if it finds that the given computing processing element has constant input-output ratio, it executes the next step. In this step, the decision process checks the type of available input data, i.e., data streams or offline data. If the model uses offline data for calculation, the decision process further checks whether the computing processing element/activity has a non-persistent view as an input or not. If an activity has a non-persistent

Checking type of input data

input view, it indicates that the workflow has multiple processing steps with intermediate results and therefore, the framework decides to apply a variant of the *multi-step probabilistic provenance inference* method which is appropriate for offline data with multiple processing steps. This version of the multi-step probabilistic provenance inference method follows the same principle as the basic provenance inference method, i.e., constant processing delay and constant sampling interval. On the contrary, if an activity has a persistent input view, the framework decides to apply the *basic provenance inference* method which is suitable to offline data with persistent input data products.

Checking system dynamics

Otherwise, in cases of data streams, the framework considers the set of parameters defining system dynamics. It checks whether there exists distributions of the parameters processing delay and sampling interval as these are required by the *probabilistic provenance inference* and the *multi-step probabilistic provenance inference* method. If these distributions do not exist, the framework has to collect this information for a pre-defined time interval during the actual execution of the model. After collecting the required distributions at run-time if necessary, the decision making process now checks the nature of these distributions. If it finds that both processing delay and sampling interval remain constant, the decision process checks for a non-persistent input view and takes the decision accordingly as discussed in the last paragraph. On the other hand, if there is a variation in processing delay and/or sampling interval, the framework again checks whether the computing processing element/activity has a non-persistent view as an input or not. If the computing processing element has a non-persistent view as an input, it means that the given workflow has multiple processing steps with non-persistent intermediate results. Therefore, in this case, the most suitable method to infer fine-grained data provenance is the *multi-step probabilistic provenance inference* method. Otherwise, the framework selects the *probabilistic provenance inference* method.

Figure 7.1 shows the complete decision tree which is facilitated by the framework to take the decision of selecting the best suited inference-based method. By selecting the best suited method, a self-adaptable framework always infers optimally accurate provenance information. Figure 7.1 also shows a few built-in methods which monitor the variation in the processing delay and sampling interval distribution and keep track of these values so that the framework can adapt the decision based on the current context during model execution.

7.3 DISCUSSION

The inference-based framework managing data provenance, has the capability of being a self-adaptable framework by using the decision tree discussed in Section 7.2. The decision tree takes several key characteristics of a scientific model into account to facilitate the decision making process. The decision making process of the self-adaptable, inference-based framework could be further enhanced by estimating the performance of these inference-based methods in terms of storage consumption and accuracy. Performance estimation is possible by introducing a cost metric based on appropriate parameters. Some of these parameters are already documented in workflow provenance information such as window size, trigger interval, processing delay distribution, sampling interval distribution etc. Different distributions could help us to estimate the accuracy of an inference-based method which could be applied over a given scientific model. Chapter 5 and 6 discuss accuracy estimation technique for the probabilistic provenance inference and the multi-step probabilistic provenance inference method, respectively. However, more parameters are required to estimate the storage consumption of inference-based methods and compare them with explicit provenance collection techniques. These parameters include size of input data tuple, size of output data tuple, size of provenance data tuple etc. A study on this cost metrics to estimate performance of inference-based methods would also help scientists to assess the applicability of a particular inference-based method. In this thesis, the complete assessment of applicability of inference-based methods based on a cost metric is not discussed due to the lack of time. However, one can get a preliminary insight of this assessment based on the evaluation results reported in Chapter 4, 5 and 6. A complete assessment of applicability of inference-based methods in terms of storage consumption and accuracy based on a cost metric is a potential direction that could be investigated in future.

7.4 SUMMARY

In this chapter, we explained the process of achieving a self-adaptable, inference-based provenance management framework. Self-adaptability is an important feature which allows the framework to adapt with the current

situation. Necessity of a self-adaptable framework was pointed out by the third research question (*RQ 3*) in this thesis, as discussed in Section 1.4.

The characteristics of a scientific model should be understood first to achieve a self-adaptable framework. We discussed a few characteristics of a scientific model in Section 1.3. Based on this discussion, in this chapter, we described the key characteristics of a scientific model which needs to be considered to incorporate the self-adaptability into the framework. Next, we presented a decision tree and explained the decision making process based on these key characteristics. The framework takes the decision of the most appropriate inference-based method based on current system dynamics and characteristics of the given scientific model by facilitating the decision tree. Finally, we discussed the possibility of an enhancement of the decision making process by including the assessment of applicability of inference-based methods based on a cost metric. We would like to thoroughly investigate this topic in future.

8

CASE STUDY I: ESTIMATING GLOBAL WATER DEMAND

The increasing data volume and highly complex scientific models used in different domains make it difficult to debug scientific models in cases of anomalies. Mostly scientists would like to trace the origin of an unexpected/abnormal output data product back to it's source data products. This way of debugging is referred to as *instance-driven debugging*. Most high-level programming languages such as Java[1] or Python[2] have their own debuggers that enable developers/scientists to monitor the execution of a program/model. These debuggers allow developers to trace back the value of a particular variable only at execution time. Furthermore, it often becomes too difficult to trace back until a desired point within the source code because of the complexity of the program and the time and manual effort it takes.

In this thesis, an inference-based framework managing data provenance is proposed. The framework is capable of extracting a data-driven workflow based on a given program and can annotate it with values of data products/tuples, reducing the effort of *instance-driven debugging*. In Chapter 3, we discussed *workflow provenance inference* method which automatically extracts the data-driven relationship among operations/processing elements, called workflow provenance graph, based on a given Python

This chapter is based on the following work: An Inference-based Framework to Manage Data Provenance in Geoscience Applications. Accepted in *IEEE Transactions on Geoscience and Remote Sensing*, IEEE Geoscience and Remote Sensing Society, 2013. (Impact Factor: 2.895) & From scripts towards provenance inference. In *Proceedings of the IEEE International Conference on E-Science (e-Science'12)*, pages 118–127, IEEE Computer Society, 2012.

1 Available at `http://www.java.com/en/`

2 Available at `http://www.python.org/`

program. Later, we presented three inference-based methods to infer fine-grained data provenance under different system dynamics in Chapter 4, 5 and 6. Upon the availability of actual input and output data products, these inference-based methods provide a fine-grained data provenance graph by annotating values of contributing input data products and produced output data products based on the given workflow provenance graph. Therefore, the inference-based framework allows scientists to trace back an output data product without even executing the actual program, realizing a scientific model.

In this chapter, we present a case study which describes the process of applying the proposed inference-based framework over a scientific model that estimates the global water demand [127]. The scientific model is developed in Python. Since Python has no built-in support to extract workflow provenance automatically, scientists have to rely on their own expertise to identify data dependencies manually between different operations within the program in case of an unexpected result. This debugging becomes more difficult and time consuming since the model processes a massive amount of offline data, stored in around 4000 flat files. Therefore, once scientists encounter any output with an unexpected value, it becomes a tedious job not only to find the contributing input values producing that unexpected output but also to visualize the data-driven relationship between different operations. Since the proposed inference-based framework can address such complexity of instance-driven debugging, we apply the framework over this scientific model. Evaluation of the inference-based framework based on this case demonstrates relevance and applicability of the proposed framework to scientific experiments.

Chapter structure

This chapter is organized in the following way. First, we describe the use case of estimating global water demand followed by the description of a few key characteristics of this scientific model. Next, we provide an overview of how the inference-based framework is applied over this model followed by the discussion on extracting data-driven workflow provenance of the scientific model and inferring fine-grained data provenance. Finally, we evaluate the performance of the inference-based framework based on a couple of interviews with scientists who develop this scientific model followed by the discussion on a few limitations of the framework within the context of this case study.

8.1 USE CASE: ESTIMATING GLOBAL WATER DEMAND

Freshwater is one of the most important resources for various human activities and food production. During the past decades, use of water has been increased rapidly, yet available freshwater resources are finite. Therefore, estimating water demand and availability on a global level is necessary to assess the current situation as well as to make policies for the future. In this use case, we focus on a scientific model that estimates the total water demand from the year 1960 to 2000 at a monthly resolution.

8.1.1 Model Inputs

Source data are collected from different existing datasets to estimate the amount of water that is required globally. The scientific model combines irrigated areas, crop-related data and simulated potential and actual evaporation and transpiration, which are all processed at a 0.5° grid spatial resolution, i.e., 50 km by 50 km, and at a monthly temporal resolution. Irrigated areas are prescribed by the MIRCA2000 dataset [104] and the FAOSTAT database[3]. Crop factors, growing season lengths, and rooting depth are obtained from GCWM [112]. The irrigated areas are representative for the period 1960-2000 at a yearly temporal resolution, i.e., remains constant over each year, while the crop-related data sets are representative for the year 2000 at a monthly temporal resolution. A map of country-specific irrigation efficiency factors is also obtained from [107]. In addition, daily potential and actual bare soil evaporation and transpiration are prescribed from the simulation results from the global hydrological and water resources model PCR-GLOBWB [120]. Various units are used during the calculation, but water demand is subsequently expressed as volume per time such as million m^3 $month^{-1}$.

Figure 8.1 shows input and output datasets and data dependencies between them. The rectangles represent input datasets collected from various sources and the shaded ones represent output datasets. A directed edge represents the data dependency of the target dataset on the source dataset. All datasets are PCRaster[4] maps, containing 360×720 cells. Moreover, ellipses in Figure 8.1 represent simulation models.

3 Available at `http://faostat.fao.org/`
4 Available at `http://pcraster.geo.uu.nl/`

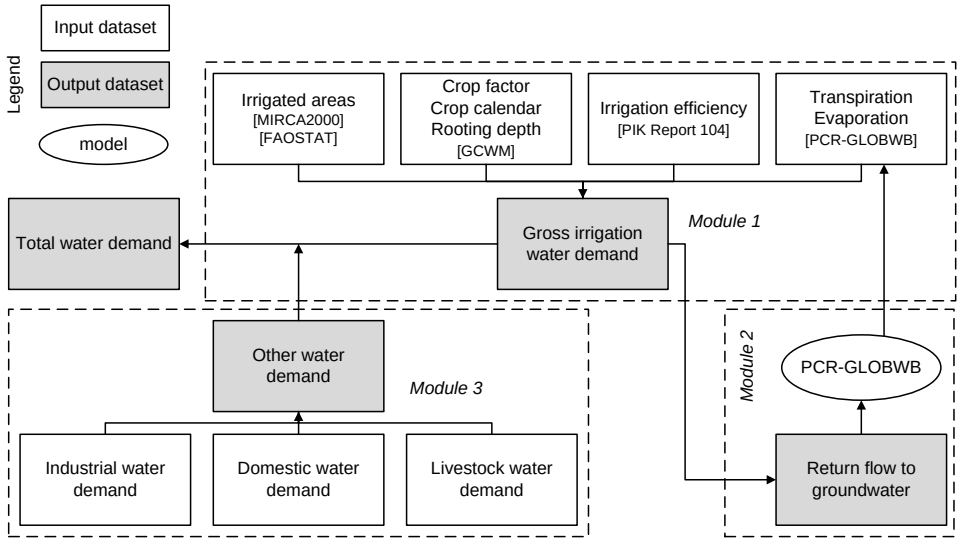


Figure 8.1: Different types of datasets used to estimate global water demand

8.1.2 Model Computation

The process begins with reading the annual and monthly input maps described in Section 8.1.1. First, using irrigated areas, crop factors, growing season lengths and potential transpiration, the model calculates potential crop transpiration per layer; the first and second layer represents for shallow and deep soil, respectively. Transpiration is a part of the water cycle and the loss of water vapor from parts of plants, i.e., in this case, irrigated crops. Due to the limited availability of soil water, transpiration at a potential rate is not often achieved, and the difference between potential and actual crop transpiration is required to be supplied by irrigation water to ensure the optimal crop growth for irrigated crops. Then we calculate actual crop transpiration by using the fraction of actual to potential transpiration, and with rooting depth to estimate the amount of soil water available to root zones. In addition, we compute the difference between potential and actual bare soil evaporation for the top soil layer. This water also needs to be applied over the irrigated areas to prevent soil salinization. Net irrigation water demand thus equals the sum of the differences between the potential and actual crop transpiration, and between the potential and actual bare soil evaporation which ensures maximum crop growth over the irrigated areas [127]. However, much of this water is lost to evaporation

and percolation during the transport and application. So, water in excess of the net demand has to be applied. Therefore, we use country-specific irrigation efficiency factors and multiply these with the net irrigation water demand to yield gross irrigation water demand. The top part in Figure 8.1 shows related input and output datasets to calculate gross irrigation water demand. These datasets are enclosed within a module which is referred to as *module 1*.

After computing the gross irrigation water demand, PCR-GLOBWB model is used to calculate the return flow to groundwater or groundwater recharge from irrigation. This water is vital when estimating renewable groundwater resources, which is used in subsequent studies [126]. Return flow to groundwater is calculated by taking the minimum of irrigation losses, i.e., the difference between the gross and net irrigation water demand, and the unsaturated hydraulic conductivity of the bottom soil layer [128]. The bottom-right part in Figure 8.1 shows the module that calculates return flow to groundwater, also referred to as *module 2*.

At last, the estimated gross irrigation water demand is added to other sectoral water demands such as industrial, domestic and livestock water demand, to calculate the total water demand. These values are directly read from the corresponding input maps. This process is shown by *module 3* in Figure 8.1, placed at the bottom-left corner.

8.1.3 Model Outputs

The model reports the resulted total water demand, gross irrigation water demand, irrigation return flow and other water demands as PCRaster maps (shaded rectangles in Figure 8.1) for each year from 1960 to 2000 at a monthly temporal resolution.

In this use case, we mainly focus on the process that calculates total water demand and gross irrigation water demand, carried out by *module 1* and *module 3* as shown in Figure 8.1.

8.2 MODEL CHARACTERISTICS

The inference-based framework is applied over the aforesaid scientific model based on the decision tree, discussed in Section 7.2. The framework facilitates the decision tree to select the best suited inference-based method that

could be applied over the scientific model. To accomplish that, the decision making process takes a few characteristics of the model into consideration. In this section, we briefly discuss the key characteristics of the aforesaid scientific model.

Model developing platform

The scientific model, estimating global water demand, is developed in Python. As described in Section 1.3, Python is a general purpose programming language and has no built-in support to collect data provenance. Therefore, based on the definition given in Section 1.3, we can classify the model as the one which has been developed in a *provenance-unaware* platform.

Type of activities

The aforesaid model is comprised of many operations which are realized by a Python program using PCRaster package. These operations are referred to as activities. Some of these activities are arithmetic operators, performing addition, subtraction, multiplication etc. between two input maps. Other activities such as *cover* replaces a missing value with an appropriate one depending on its parameters which are input maps. All these activities perform their intended operation by considering each cell of given parameters, i.e., input maps, and produce corresponding output. Therefore, these activities are executed on the cell level. One important characteristic of these activities is their *input-output ratio*. Since these activities are executed on cell level, all these activities have *constant input-output ratio*.

Type of input data

The input dataset used in this scientific model has been collected before. Therefore, the model estimates global water demand based on offline or non-stream data products. The input dataset has around 4000 PCRaster maps which produce water demand map in a global level from year 1960-2000 at a monthly resolution.

System dynamics

The system dynamics of the scientific model is another important characteristic to consider. System dynamics include a set of parameters that describes the time required to process input data products, referred to as *processing delay*, as well as the time between the arrival of two successive input data products, referred to as *sampling interval*. Since the model facilitates offline input data products for calculation, these two parameters have no effect over actual execution of the model.

In a nutshell, the aforesaid scientific model is developed in a *provenance-unaware* platform. The model consists of several *constant ratio* activities which facilitate *offline* data to produce output. Parameters like *processing delay* and *sampling interval* have no effect during execution because of the offline processing nature of the model.

8.3 OVERVIEW: APPLYING INFERENCE-BASED FRAMEWORK

In previous section, we described a few key characteristics of the scientific model that estimates global water demand. As explained in Section 8.2, the model has been developed in a *provenance-unaware* platform, using Python language. Therefore, we can apply the *workflow provenance inference* method, discussed in Chapter 3, to extract the workflow provenance of the model automatically. Workflow provenance information is represented as a graph, referred to as *workflow provenance graph*, showing the data-driven relationship among all activities on a cell level. Next, we can infer fine-grained data provenance by facilitating the workflow provenance graph and available data products. Fine-grained data provenance inference phase is only executed when scientists request provenance for a particular output data product which seems to have an unexpected value. In this case, since input and output data products are already available, we can initiate the fine-grained provenance inference phase without even executing the scientific model. This phase provides an inferred fine-grained data provenance graph which is annotated by the actual values of input and output data products by facilitating the data-driven workflow provenance graph. Next, we describe workflow provenance inference method and fine-grained provenance inference method which are applied over the aforesaid scientific model.

8.4 WORKFLOW PROVENANCE INFERENCE

As mentioned in Section 8.3, workflow provenance inference method, discussed in Chapter 3, is applied over the Python program, realizing the computation of the scientific model. The Python program is comprised of several activities, realizing different kinds of operations which include assignment, looping, PCRaster operations etc. Operations such as assignment, PCRaster operations are purely based on data-flow coordination where availability of data triggers the next activity. However, operations like looping is implemented by using control-flow based coordination and results into control dependencies between pertinent activities. Since data provenance identifies the data dependencies between activities, control dependencies must be transformed into data dependencies to infer workflow provenance.

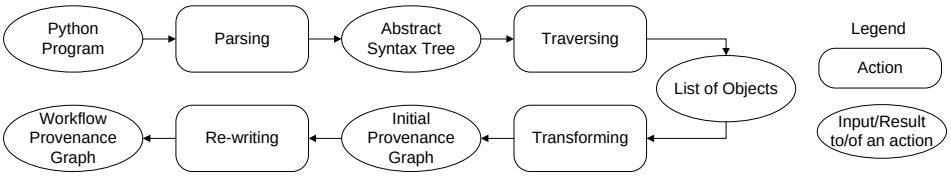


Figure 8.2: Steps during workflow provenance inference

Mechanism

Figure 8.2 depicts the steps associated with the workflow provenance inference method which is applied over the aforesaid model. In Figure 8.2, ellipses represent either input or output of a particular action/step which is represented by a corresponding round-shaped box. First, the workflow provenance inference method takes the Python program as an input. Then, the method parses the given Python program based on a combined grammar, containing parser and lexer rules. After parsing the script, it returns an abstract syntax tree (AST) for the given Python program. Then, the method traverses through this AST by facilitating a tree grammar and for each node in the AST, an object of the appropriate class based on the object model of Python is created. The outcome of the traversing step is a list of objects. Then, the method transforms this list of objects into an initial workflow provenance graph based on the workflow provenance model, discussed in Section 3.1. The initial workflow provenance graph has four types of nodes such as source processing elements, computing processing elements, constants and views. This graph preserves control-flow based coordination between nodes and therefore, it has to be transformed into a form where the initial workflow provenance graph exhibits data dependencies only. To do this, the workflow provenance inference method applies a set of functions, consisting of graph re-write rules, discussed in Section 3.6, 3.7 and 3.8. A re-write rule has two parts: left-hand side (LHS) and right-hand side (RHS). Once a rule is defined and is executed, it searches for the isomorphic sub-graph equivalent to the sub-graph pattern mentioned in the LHS of the rule. If the isomorphic sub-graph is found, it is replaced by the sub-graph pattern mentioned in the RHS of the rule. This process continues till no other isomorphic sub-graph equivalent to the LHS sub-graph pattern exists.

At the end of the re-writing step, the workflow provenance graph of the given Python program is extracted, showing data-driven relationship among activities. The Python program, realizing the computation of the aforesaid scientific model, has 116 lines of code. Table 8.1 shows the num-

Table 8.1: Comparison of Number of Nodes in provenance graphs

Node Type	Initial Workflow Provenance Graph	Final Workflow Provenance Graph
Source processing elements	33	33
Computing processing elements	147	93
Constants	58	0
Views	180	0
Total nodes	418	126

ber of nodes in both initial workflow provenance graph and workflow provenance graph. The initial workflow provenance graph has 418 nodes. After applying different re-write rules on this graph, total number of nodes in the final workflow provenance graph becomes 126 which is around one-third of the initial workflow provenance graph in size. In Table 8.1, we can see that the number of both constant nodes and view nodes are 0 in the workflow provenance graph. This is because of applying the graph compression re-write rules, discussed in Section 3.8, which encode constant and view nodes properties into corresponding source processing elements or computing processing elements and then delete that particular constant/view nodes.

A workflow provenance graph depicts the data-driven relationship among different processing elements within the model. Furthermore, a workflow provenance graph also documents the values of properties of different nodes as discussed in Section 3.1. Next, a workflow provenance graph is facilitated during fine-grained provenance inference phase.

8.5 FINE-GRAINED DATA PROVENANCE INFERENCE

The fine-grained provenance inference phase is executed once scientists request provenance information of an output data product which seems to

have an unexpected value. To infer fine-grained data provenance, first, we need to import input and output data products into a database. We create a SQLite[5] database for this purpose. In this use case, there are around 4000 PCRaster[6] maps. Each map contains $720 \times 360 = 259200$ cells. We create a data tuple/row for each cell. Further, we attach a timestamp to every data tuple, representing the point in time a particular tuple is valid. As an example, if a particular tuple is valid during June, 1990, the timestamp field is set to $1990 - 06$. The total number of data tuples in the database is more than 1 billion which require around 50 GB of storage space.

Mechanism

After importing data products into a database, the appropriate fine-grained provenance inference method is executed. The self-adaptability nature of the framework selects the appropriate inference-based method by facilitating the decision tree, shown in Figure 7.1. As already mentioned in Section 8.2, the scientific model computes over offline data. These offline data participates in a series of activities/operations, i.e., a processing chain, based on the captured workflow provenance. Therefore, we apply a variant of the *multi-step probabilistic provenance inference* method based on the decision making process, discussed in Section 7.2. This version of the multi-step probabilistic provenance inference method has a few difference with the original one, discussed in Chapter 6. This inference-based method has no dependency over any specific distributions that describe system dynamics because of the offline nature of the processing and data products. Therefore, the variant of the multi-step probabilistic provenance inference method follows the same principle of the basic provenance inference method, discussed in Chapter 4, but can be applied over multiple processing steps with non-persistent, intermediate result.

The outcome of this inference phase is a *fine-grained data provenance graph* which annotates the workflow provenance graph of the scientific model with the exact value of contributing input data products. Scientists can facilitate a fine-grained data provenance graph to trace an output data product having unexpected value back to it's source values.

5 Available at `http://www.sqlite.org/`

6 Available at `http://pcraster.geo.uu.nl/`

8.6 EVALUATION

The inference-based framework, discussed in this thesis, is evaluated by facilitating the use case based on the scientific model estimating global water demand which is discussed in Section 8.1. The proposed inference-based framework achieves 100% accurate fine-grained data provenance without consuming any extra storage space to store provenance data.

Furthermore, we had a couple of evaluation sessions with two scientists who are working on this use case to discuss potential applications of fine-grained data provenance graphs. The transcript of these meetings is documented in Appendix A.2. We arranged two evaluation meetings with the scientists. During the first evaluation meeting, we presented the preliminary results, i.e., fine-grained data provenance graphs, and demonstrated the initial version of the prototype, realizing the proposed framework. We also asked them a few questions about the applicability of fine-grained data provenance graphs. Scientists also provided their feedback on different aspects of the developed tool.

Based on the feedback of the scientists, we extended the prototype and tested it with different types of Python programs. After finalizing the prototype, we arranged another evaluation session with the scientists. In this session, we asked them a few open-ended questions on several features of the proposed framework. These features are: i) ease of debugging, ii) extensibility and iii) reproducibility. Next, we present the key points discussed during these evaluation sessions. Transcripts of these sessions are documented in Section A.2.4 and A.2.4.

8.6.1 Ease of Debugging

Debugging is an emerging application of data provenance [74]. Both workflow debugging (code-level) and instance-driven debugging (value-level) are possible based on the granularity level of provenance information. In the context of this inference-based framework, a workflow provenance graph, which automatically captures provenance from a Python program, shows the data-flow of the program and hence, can be used for code-level debugging. A fine-grained data provenance graph shows the contributing values and hence, can be facilitated to satisfy a request of value-level de-

bugging. In this connection, we would like to highlight a point raised by one of the scientists during the first evaluation session.

> *When I (the scientist) encounter a value which seems to be unexpected, I would like to trace back to source values which contribute to produce the abnormal one. To do that, I (the scientist) must find all contributing points out of hundreds of input maps which is quite time consuming and frustrating. The built-in debugging option of Python does not help much because of the complexity of the operations as well as the length of the processing chain, i.e., large number of operations. I (the scientist) would be happy to have a tool that can show me all contributing values automatically and within a short time period.*

The aforesaid point motivates the potential use of data provenance for debugging purpose well. In this connection, we asked the following question to the scientists.

Question

To what extent do you think that the provenance graphs are useful for debugging? Do provenance graphs make debugging easier than built-in debuggers?

Feedback

Scientists appreciate the idea of debugging their model using provenance graphs. Workflow provenance graphs enable code-level debugging which is useful to determine the efficiency of the code, i.e., finding out code repetition within a program. Workflow provenance graphs are also useful to compare two different versions of the code, expected to produce the same value.

Fine-grained data provenance graphs enable value-level debugging. A fine-grained provenance graph generated for a particular instance/output data product shows all contributing input data products with their values. Therefore, it provides easy access to actual data within short period of time. Fine-grained provenance graphs also prove beneficial while tracing back to identify the missing values in the file which is another aspect of value-level debugging.

Usually, scientists use the debugging option that comes with the development environment. However, the classical debugging technique requires

scientists to understand related operations completely and it also requires scientists to interpret the debugging results. Therefore, it is a time consuming process and requires necessary expertise.

8.6.2 Extensibility

Another feature of the proposed inference-based framework is the extensibility of the framework. Extensibility refers to the ability to handle different Python programs and extracting workflow provenance graphs out of them. Our prototype of the framework can handle varieties of Python programs using different libraries. However, a user has to provide a few parameters for each method signature at the very first occurrence of the method while executing the prototype. These parameters include whether the method reads persistent data or not (e.g. true/false) and whether the method writes persistent data or not (e.g. true/false). We asked scientists the following question regarding extensibility.

Question

To what extent do you think that the extensibility of the proposed framework is justified?

Feedback

The proposed approach is generic in the sense that it can handle a variety of Python programs and can extract workflow provenance graphs out of those. However, during the first time execution of the prototype, the user has to enter method-specific information which might be time consuming and also requires some training on how to provide this information to the framework.

8.6.3 Reproducibility

Reproducibility refers to the ability to produce the same result using the same set of input data products, irrespective of the time of the execution of the involved operations. Data provenance can help scientists to achieve reproducible results. Since the proposed inference-based framework infers

fine-grained data provenance, we asked scientists the following question regarding reproducibility.

Question

To what extent do you think that a fine-grained provenance graph is useful to achieve reproducibility? How do you use your reproducible results?

Feedback

A fine-grained data provenance graph shows source data values contributed to produce a result which explains the derivation history of that particular output and also provides a replication recipe of that output. In practice, reproducible results might be useful to explain the mechanism of the scientific model to one of the other scientists from the same group. However, the exchange of reproducible results from one research group to another is not very common.

8.6.4 Remarks

Based on the interview with scientists, we have found that scientists like the idea of tracing back to the input datasets by facilitating fine-grained data provenance information. Furthermore, workflow provenance graph is also useful to them since it explains the complete mechanism of the scientific model visually which is easy to understand and learn. In future, we would like to conduct a usability study on using data provenance as a debugging tool among a large group of scientists. Extensibility of the framework ensures that the developed prototype can handle different Python programs. However, the method-specific information must be entered once during the first time execution of the prototype. Since it requires a few parameters, potential users could be trained with a reasonable effort. Scientists also recognize that fine-grained data provenance graphs help to achieve reproducible results which could be used to validate the scientific model. Overall, the inference-based framework managing data provenance could possibly satisfy scientists with its simplicity and ease to use.

8.7 DISCUSSION

The inference-based framework, discussed in this thesis, is applied over the scientific model estimating global water demand. The evaluation shows that scientists who develop the model are satisfied with the performance and features of the framework. The framework infers 100% accurate fine-grained data provenance. The reasons to achieve maximum accuracy is that the available data products are offline in nature and the processing is also executed offline. Therefore, there is no influence of variable processing delay and sampling interval over the inference method.

However, there is one particular situation where applicability and accuracy of the inference-based framework could be compromised. Next, we discuss this limitation of the framework in the context of this use case, discussed in this chapter.

8.7.1 Recursive Operations

The scientific model estimating global water demand uses Python programs with PCRaster[7] libraries. PCRaster is a programming package that defines and supports many spatio-temporal operations. The Python program realizing the scientific model has different classes of operations such as arithmetic, boolean, missing value creation etc. All these operations are executed on a cell level. The proposed framework can handle these operations and extract a workflow provenance graph automatically from the given program.

However, there is a class of operations in PCRaster package, named as neighborhood operations. Operations falling in this category are considering the neighborhood cells while calculating some value for a particular cell. It means that the value of a cell (x, y) depends on the neighboring cells which construct an imaginary square having $(x + 1, y + 1)$ as top-right corner and $(x - 1, y - 1)$ as bottom-left corner. Furthermore, these operations are recursive in nature.

One of the limitations of the inference-based framework is its inability to handle recursive operations since it is quite difficult to determine the *input-output ratio* of a recursive operation in advance. *Input-output ratio* refers to the ratio between number of contributing input tuples producing

7 Available at `http://pcraster.geo.uu.nl/`

output tuples and number of produced output tuples and it is an important parameter during the inference phase. Even if we can figure out the input-output ratio of a recursive operation, representing all contributing tuples in a recursive operation would significantly increase the size of a fine-grained data provenance graph. Due to these limitations, the proposed inference-based framework is not suitable for recursive operations.

8.8 SUMMARY

In this chapter, we presented a case study, applying the proposed inference-based framework over a scientific model that estimates the global water demand. First, we introduced the use case based on the scientific model. The scientific model computes over different types of input data products, collected from different sources beforehand. Therefore, the data products are offline data (non-stream) and the model is also executed offline. Next, we identified the key characteristics of the scientific model. These characteristics are considered by the self-adaptability nature of the framework to apply the appropriate inference-based methods over the scientific model. This decision making process facilitates the decision tree, discussed in Chapter 7. Since the model is developed in Python which is a provenance-unaware platform, the decision making process applies the workflow provenance inference method, discussed in Chapter 3, to extract a workflow provenance graph automatically based on the given Python program. Later, the workflow provenance graph is used during the fine-grained provenance inference phase. During this phase, the self-adaptability nature of the framework, discussed in Chapter 7, decides to apply a variant of the multi-step probabilistic provenance inference method which is tailored for offline data and offline processing. The inference-based framework achieves 100% accurate fine-grained data provenance without having any extra storage overhead to maintain provenance data. Furthermore, we conducted a couple of evaluation sessions with scientists who developed this model. Scientists accepted the importance of data provenance especially for instance-driven debugging and fine-grained provenance graphs serve that purpose well. In sum, the proposed inference-based framework has features relevant and suitable for scientists who would like to keep track of provenance data at minimal effort and time for their scientific experiments.

9

CASE STUDY II: ACCESSIBILITY OF ROAD SEGMENTS

Data provenance is often used for auditing and debugging of data intensive applications [114, 74]. It allows scientists working in different domains, to trace the origin of a data product and to validate their models. In case of a highly complex model and massive amount of data to deal with, data provenance provides an easy access to source values which contribute to produce a particular output value.

An inference-based framework inferring data provenance of a scientific model is proposed in this thesis. The framework is capable of extracting the workflow provenance based on a given program of a model and can annotate the workflow provenance with values of participating data products/tuples, i.e., fine-grained data provenance. In Chapter 8, we demonstrated the applicability of the inference-based framework on a scientific model estimating global water demand which is developed in Python. The framework can extract workflow provenance automatically based on a given Python program and can also infer fine-grained data provenance of a selected output data product/tuple which explains the derivation history of that particular tuple. Therefore, fine-grained data provenance can be used as a potential debugging tool to know the origin of a particular data product/tuple, also referred to as *instance-driven debugging*.

To broaden the scope of the framework, we extend the framework beyond procedural languages, to be used for purely declarative languages such as logic programming under the stable model semantics. A class of

This chapter is based on the internal report: Data Provenance Inference in Logic Programming: Reducing Effort of Instance-driven Debugging. Technical Report TR-CTIT-13-11, Centre for Telematics and Information Technology, University of Twente, 2013.

these languages based on the stable model semantics, namely Answer Set Programming (ASP), has been intensively applied in the last decade to solve computationally hard problems [52]. The ability to deal with incomplete knowledge, conflicting and noisy input and common sense reasoning made ASP applicable to domains such as civil engineering, home healthcare, sensor networks, planning, bio-informatics, phylogenesis, system biology, industrial applications and more [101, 88, 89, 39, 49]. Despite this wide applicability, explaining unexpected outcome in an instance-driven way, by detecting errors in the knowledge model or in the inference rules related to a particular output data product, is still under investigated. Therefore, in this chapter, we demonstrate the viability of the proposed framework as an additional tool for debugging ASP programs.

We apply the inference-based framework over a scientific model, based on an ASP program, that provides an indication on the accessibility rating of a particular road segment by facilitating streaming data from different sources such as twitter, rss and weather update etc. The accessibility rating of a road segment is based on the traffic delay, i.e., whether it is expected that traveling along this road will cause delays or not. For experiments and validation, we use the Answer Set Programming solver oClingo [50], which makes it possible to formulate and solve stream reasoning problems in a purely declarative fashion. We demonstrate how the benefits of the inference-based framework, discussed in this thesis, over the explicit provenance collection method still holds in a declarative setting. Furthermore, we briefly discuss the potential impact of the framework over declarative programming, in particular for instance-driven debugging of the model in declarative problem solving.

Chapter structure

This chapter starts with an overview on Answer Set Programming. Then, we describe the use case of determining accessibility of road segments in a particular region followed by the description of the key characteristics of the model. Next, we provide an overview of how the inference-based framework is applied over this model followed by the discussion on extracting workflow provenance of the scientific model and inferring fine-grained data provenance. Finally, we evaluate performance of the inference-based framework by providing both quantitative and qualitative analysis followed by discussion on a few limitations of the current approach in the context of this case study.

9.1 BACKGROUND ON ANSWER SET PROGRAMMING

Answer Set Programming (ASP) [10, 83, 52] is a purely declarative and non-monotonic logic programming paradigm. It applies "generate and test" approach, based on the stable model semantics [54] where solutions are represented by sets of atoms (*answer sets*) for which all rules in the program are satisfied. ASP enables programmers to model solutions to a particular problem by defining *what* valid outputs are, rather than *how* those outputs should be derived.

One of the distinguishing features of ASP is its ability to derive multiple answer sets. This is primarily achieved through *non-monotonicity* and the use of Negation As Failure (NAF). In general, this allows ASP to perform default reasoning as well as the ability to non-deterministically derive solutions even when reasoning under incomplete knowledge. Additionally, ASP's declarativeness means that the ordering of statements is irrelevant.

Stream reasoning

In this chapter, we focus on scientific models with complex stream reasoning capabilities based on ASP, which can offer these reasoning capabilities along with pure declarativity in the problem specification [51]. Stream processing and reasoning [36, 35, 117] are a relatively new area of research individually concerned with processing large amounts of dynamically generated data. While stream processing is under active research for several years already, stream reasoning is a relatively new field which focuses on manipulating streaming data as knowledge through abstraction and logic inference.

While some attempts have been made to re-purpose ASP reasoners for streaming data [37], The current work by Gebser et al. [50] provides an all encompassing software solution to ASP-based stream reasoning. *oClingo* is an Answer Set Programming reasoner designed to work similarly with the normal syntax and semantics of the language, but with the addition of new functionality to allow for the streaming input of data. It is primarily based on their previous work into incremental based ASP reasoning [48] in which external data was sequentially passed as input into the reasoner to provide continuous results.

oClingo has the ability to reason upon different frames of reference in addition to its powerful ASP-based reasoning. This allows programmers to define reasoning tasks in which temporal information can be deliberated upon and solutions to frame-based problems (e.g. involving *sliding windows* of knowledge) could be encoded. The oClingo implementation

Table 9.1: Relevant oClingo ASP Syntax

Syntax	Description
$h \leftarrow a_1, ..., a_m[, not\ a_{m+1}, ..., not\ a_n].$	Logical rule
$a \leftarrow .$	Fact
$\leftarrow a_1, ..., a_m[, not\ a_{m+1}, ..., not\ a_n].$	Constraint
$l\{h_1, .., h_n\}u \leftarrow$ $[a_1, ..., a_m[, not\ a_{m+1}, ..., not\ a_n]].$	Choice [rule] ($l, u \in \mathbb{N}$)
$\#external\ a.$	Indicates streaming input
$\#volatile\ t : \mathbb{N}$	Indicates time-decay rules

achieves this by extending the ASP language to deal with emerging and expiring knowledge, and to reason with time-decaying logic programs. In Table 9.1, the part of oClingo's ASP syntax for stream reasoning relevant to the scientific model, discussed in Section 9.2, is shown.

9.2 USE CASE: ACCESSIBILITY OF ROAD SEGMENTS

ASP provides stream reasoning capabilities in a purely declarative fashion. In this section, we present a use case that can illustrate the power of ASP well in case of stream reasoning and processing.

Forecasting traffic information over a network of road segments makes drivers and travelers aware of potential problems and hazards and eventually can reduce accidents. In this section, we consider the problem of determining the accessibility of a particular road segment/location in a specific area (e.g London, UK). We consider available data from various sources, namely i) the traffic conditions on major arterial roads, ii) the weather conditions being received from various stations within the area, and iii) twitter status updates from users, specific to the geo-location and using some particular keywords or hashtags for filtering.

These sources provide data streams, i.e., data tuples sent to the destination continuously. To build an application that can constantly update the accessibility status of different road segments, we need to be able to process and reason data streams produced by the sources, based on some

intelligent aggregation and reasoning over the knowledge being received. As an example, in the worst case scenario, the program could infer that the location is *very inaccessible* if all three streams report that it is difficult to enter or exit the area (e.g. slow traffic due to an accident; a snow blizzard hindering travel). When information from data streams is lacking or contradictory, the program would need to combine available knowledge in a qualitative ranking, in order to decide a suitable *accessibility rating* to inform potential travelers. Contextual information or user preferences can also be used to guide the final results. The possibility of having multiple possible solutions supported by subsets of the available input streams can be also desirable when the qualitative metrics do not lead to any actual winning outcome.

As already mentioned, possible errors in modeling the domain or formulating constraints and qualitative criteria for ranking, might result into an unexpected output. In this case, domain experts or scientists might be interested to trace that unexpected outcome back to the details of contributing data which is not a trivial task in ASP due to its declarative nature and to the lack of tool for instance-driven model debugging.

9.3 REPRESENTING USE CASE IN A LOGIC PROGRAM

To deal with these different aspects of the use case, described in Section 9.2, we declaratively encoded it in ASP. The logic program consists of two primary processes: a) generating the search space of plausible answer set solutions based on the input data streams, and b) infer answer sets relevant only to the data streams by navigating and pruning the search space. Grounding techniques for ASP generate a complete and defined search space domain and the ASP solver explores and prunes the search space (via heuristics) to find the desired solutions with some optimization and simplification that can reduce the complexity of this phase.

Search space generation

The search space and the domain of interest have to be defined in a closed-world setting. As an example, Listing 9.1 shows a few facts and a logical rule. Line 2 in Listing 9.1 indicates that there could be three types of streams: weather updates, rss feeds and twitter status updates. Furthermore, the facts shown in line 3-4 report that the value of a weather status is bounded to the keyword 'snow' and the value of a rss feed is bounded to the keyword 'accident'. This is how we can restrict the status keywords

into a finite set, i.e., a closed-world. Line 6 shows a logical rule that considers all valid weather and rss feed status types to produce a twitter status update.

Listing 9.1: Predicates showing facts and a logical rule

```
1 % FACTS
2 streamtype(weather; rss; twitter).
3 status(weather, snow).
4 status(rss, accident).
5 % Copies all valid weather and rss status types into tweets
6 status(tweet,X) :- status(S,X), streamtype(S).
```

Based on this given facts and the logical rule, the grounding of ASP generates all possible valid twitter status updates. Listing 9.2 shows the grounding of Listing 9.1. This is how we can limit the search space and then, all possible combinations of valid predicates are generated.

Listing 9.2: Grounding of Listing 9.1

```
1 status(tweet,accident).
2 status(tweet,snow).
```

Limiting search space

Up until this point, we have not referred to any data streams yet. So far we have defined permutations of all possible solutions that the program can output. In most cases, this space can be very large and most likely contain many solutions those are not relevant. To limit answer sets to relevant solutions, logical constraints are applied which help pruning portions of the search space. With input data streams, we use constraints in this way to ensure that data streams elements correspond to solutions within the search space. A *binding* is performed by encoding rules that essentially state that valid solutions from the program should only be those that relate to data that has been recently streamed.

Through the combination of search space generation and elimination via constraint based reasoning, the logic program based on ASP can essentially aggregate the information from three streams: twitter, rss and weather update, as discussed in Section 9.2, to determine the accessibility of a particular road segment in a given settings.

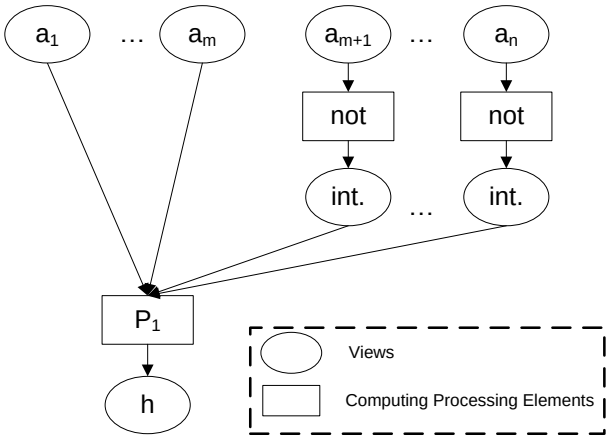


Figure 9.1: Representation of a logical rule based on Workflow Provenance Model

9.4 MODEL CHARACTERISTICS

The proposed inference-based framework is applied over the aforesaid scientific model based on the decision tree, discussed in Section 7.2. The framework facilitates the decision tree to select the best suited inference-based method that could be applied over the scientific model. To accomplish that, the decision making process takes a few characteristics of the model into consideration and then, decide accordingly. In this section, we briefly discuss the key characteristics of the aforesaid scientific model and also compare these characteristics with the other scientific model, discussed in Chapter 8.

In Section 9.3, we describe the basic principle of encoding the use case into a logic program. The logic program, written in ASP, constitutes the scientific model that determines accessibility of road segments in a particular area. Answer Set Programming has no native support to collect data provenance either at design or at run time of the model. Therefore, based on the discussion in Section 1.3, we can classify the model as the one which is developed in a *provenance-unaware* platform.

Model developing platform

The aforesaid scientific model is comprised of many logical rules. A logical rule, R, has two parts: head and body, as shown in Table 9.1. If the predicates in the body are satisfied then the predicate in the head of the rule can be inferred. Each logical rule in an ASP is represented as an activity in the data-driven workflow of the logic program. As described in Section 1.3.2, activities are also referred to as computing processing ele-

ments during the execution of the model. According to the definition in Section 3.1, a computing processing element has views/constants as input and produces another view as an output. In case of a logic program, predicates in both head and body of the rule are represented as views in the workflow of the logic program. Therefore, each predicate is represented by a single view unlike the scientific model, discussed in Chapter 8, where a view consists of multiple data tuples/products. Furthermore, the way activities are executed also differs in this case compared to the model, used in case study I. In case study I (Chapter 8), the model is comprised of many activities that are triggered independently during the model execution. However, in this case, the entire search space is generated at once and at each logical clock tick, the search space is pruned via constraints to derive answer sets. Therefore, rules/activities in the model involved in case study II are triggered at once at each logical clock tick.

Semantics of Activities & Views

Figure 9.1 shows the resulting workflow provenance of a logical rule, R, defined in Table 9.1. In Figure 9.1, the rule is represented as a computing processing element P_1, which has a number of body predicates (views) such as $a_1, ..., a_m$ as input and produces a head predicate (view) h as the output. Moreover, some of the body predicates $a_{m+1}, ..., a_n$ are connected to their corresponding processing elements *not* which produce intermediate results, represented as views. These intermediate views are also connected to the computing processing element P_1 as input.

Example

As already discussed in Section 3.1, nodes in a workflow have different properties and the proposed inference-based framework can infer fine-grained data provenance by facilitating these properties. One important property of these activities/processing elements is the *input-output ratio* which refers to the ratio of number of contributing input tuples/predicates producing output tuples/predicates to number of produced output tuples/predicates. The computing processing element P_1, representing the logical rule R, has *constant input-output ratio* of 'many to one', since there exist multiple input predicates (multiple views) which derive the output predicate (a view). In a logic program, there could be processing elements with *variable input-output ratio* as well. In Appendix A.3.4, an example with *variable* ratio processing elements is described in detail.

Type of activities

These activities of the scientific model compute over data streams which come from three different sources such as twitter, rss and weather update. Based on the available data at a particular point in time the model determines the accessibility rating of road segments. Therefore, this model is

Type of input data

Table 9.2: Differences between Case study I and Case study II

Characteristic	Case Study I	Case Study II
Language	Procedural (Python)	Declarative (Answer Set Programming)
Structure	Flow-based	Predicates-based
Input Data	Offline	Streaming
Activities	Independent trigger interval	All activities trigger at a time
Results	Deterministic	Non-deterministic

executed based on data streams. The scientific model, used in case study I (see Chapter 8), takes offline (non-stream) data as input data products.

The ASP solver oClingo is used to reason over input data streams. During the reasoning process, oClingo assigns a logical timestamp to each data product. Then, it executes all computing processing elements at once, as already mentioned before. This is a major difference with the model developed using a flow-oriented structure like the one discussed in Chapter 8. Output data products produced by the model is also assigned with the same logical timestamp indicating that these are valid outputs at that time point. Therefore, in this case, there is no processing delay which refers to the amount of time required to process input data products, due to the nature of oClingo execution. Moreover, sampling interval, referring to the time between the arrival of two successive input data products, also has no influence over the execution since the complete workflow is executed at a time. Therefore, this model differs from the one described in Chapter 8, which is influenced by system dynamics during its execution.

System dynamics

Another major distinguishing feature between these two models is their ability to derive multiple answer sets/results. The scientific model involved in case study I always provides exactly one answer for each operation. However, the scientific model, discussed in this case study, is developed using ASP and can derive multiple results at a single point in time which is achieved through non-monotonicity and the use of Negation As Failure (NAF).

Non-monotonic reasoning

Table 9.2 summarizes the aforesaid differences between two case studies. In case study I (Chapter 8), we have discussed mechanisms of applying the inference-based framework over a model, developed in a procedural language, to infer fine-grained data provenance. In this chapter, we extend the scope of inference-based framework by applying it over a model developed in a declarative fashion which shows the wide applicability of the inference-based framework.

9.5 OVERVIEW: APPLYING INFERENCE-BASED FRAMEWORK

The characteristics of the scientific model determining accessibility rating of road segments are discussed in Section 9.4. As explained in Section 9.4, the scientific model has been developed in a *provenance-unaware* platform, using Answer Set Programming (ASP). Therefore, we apply the workflow provenance inference method (Chapter 3), to extract workflow provenance of the model. While the basic principle and the workflow provenance model remain the same as discussed in Chapter 3, the mechanism of extracting workflow provenance has to be adjusted and extended to deal with an ASP program. In this case, workflow provenance is extracted based on the complete search space generated by grounding techniques of ASP. Grounding binds the given facts to the appropriate parameters of a predicate and generates all possible combinations of valid predicates.

The workflow provenance of the given logic program represents data dependencies between predicates through logical rules/activities of the complete search space. At a particular point in time, depending on recently received elements of data streams (input data products), only a few predicates satisfy logical rules which infer answer sets (output data products). The sub-graph consisting of these satisfying predicates and their corresponding rules can explain the derivation history of a particular answer set which is referred to as fine-grained data provenance. Therefore, fine-grained data provenance is inferred by facilitating the workflow provenance graph and available data products. Fine-grained data provenance inference phase is only executed when scientists request provenance for a particular output data product.

Next, we describe both workflow provenance inference method and fine-grained data provenance inference method which are applied over the aforesaid scientific model developed in ASP. To have a quantitative analysis

of the framework in terms of accuracy, execution time and storage costs, we also collect fine-grained data provenance at run time of the model, which is referred to as explicit provenance method. We realize explicit provenance method by extending the logic program to encode the provenance information explicitly as predicates. The details of explicit provenance collection method in the context of a logic program is given in Appendix A.3.

9.6 WORKFLOW PROVENANCE INFERENCE

The inference of the workflow provenance is based on static analysis of a logic program. The oClingo tool explicates the search space where all possible instances of logical rules and constraints are explicated. The workflow provenance inference method analyzes the grounding of the logic program and clusters the rules based on their types. Based on these clusters a workflow provenance graph can be inferred consisting of different types of nodes as discussed in Section 3.1.

A cluster of rules can be derived from the information provided by the grounding of the logic program. As already mentioned, grounding of the logic program is the explication of the search space, which cannot be limited due to the usage of volatile predicates, i.e., elements of data streams. In the grounding, logical rules and constraints containing non-volatile predicates are not represented but only the result of the logical rule since it is static. However, the volatile part of logical rules and constraints remain as rules in the grounding. In the following, we give an example of clustering rules and constraints contained in the grounding and discuss the method of extracting the workflow provenance graph.

Listing 9.3: Example of logical rules

```
1 riskvalue(rss, high, LOCATION) :- rss(STATUSTYPE, LOCATION, SEVERITY
      , TIME), negative(STATUSTYPE), SEVERITY > 1.
2 riskvalue(rss, low, LOCATION) :- roadsegments(LOCATION), not
      riskvalue(rss, high, LOCATION).
```

Example

Listing 9.3 shows two logical rules mentioned in the ASP program of the model. These rules infer the `riskvalue` associated with a road segment of a particular `LOCATION` based on a rss data product/tuple. The risk values are categorized as *high* and *low*. The rule mentioned in line 1 indicates that if the `SEVERITY` of a *negative* `STATUSTYPE` such as *accident*, *brokencar*, *roadwork*

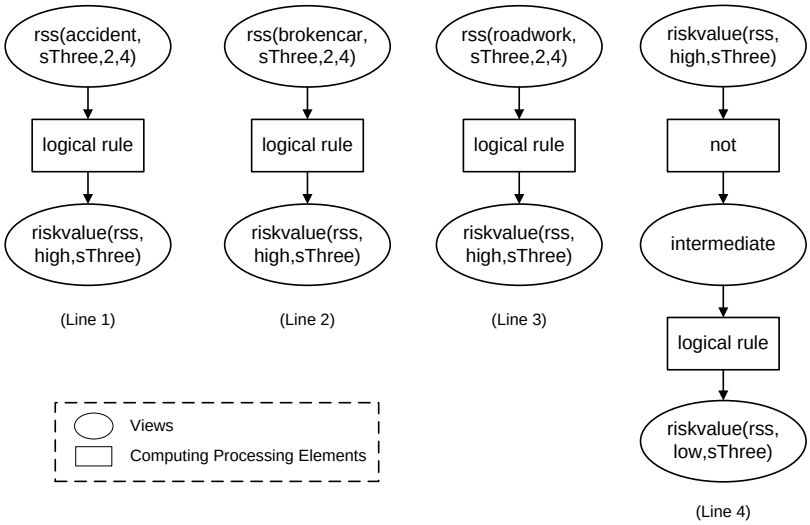


Figure 9.2: Initial workflow provenance graphs before clustering based on Listing 9.4

is greater than 1 then the rule infers that the `riskvalue` of that particular `LOCATION` is *high*. Please note that the keywords like *accident* of negative status types are defined as facts within the logic program. The other rule mentioned in line 2 indicates that if the model cannot infer the `riskvalue` of a particular `LOCATION` as high, the model infers the `riskvalue` as *low* by default.

Listing 9.4: Examples of the grounding of logical rules shown in Listing 9.3

```
1 riskvalue(rss,high,sThree):-rss(accident,sThree,2,4).
2 riskvalue(rss,high,sThree):-rss(brokencar,sThree,2,4).
3 riskvalue(rss,high,sThree):-rss(roadwork,sThree,2,4).
4 riskvalue(rss,low,sThree):-not riskvalue(rss,high,sThree).
```

The grounding of a logic program having the aforesaid rules produces logical rules shown in Listing 9.4. The first three rules infer the `riskvalue` as *high* of a road segment in `LOCATION` *sThree* at logical timestamp 4 due to the appropriate rss predicates, valid at the same logical time point (see Line 1-3 in Listing 9.4). The rule mentioned in line 4 infers that the `riskvalue` of *sThree* is *low* by default if the model fails to infer the `riskvalue` as *high* for the same location. As already mentioned, the grounding does not contain any non volatile predicates.

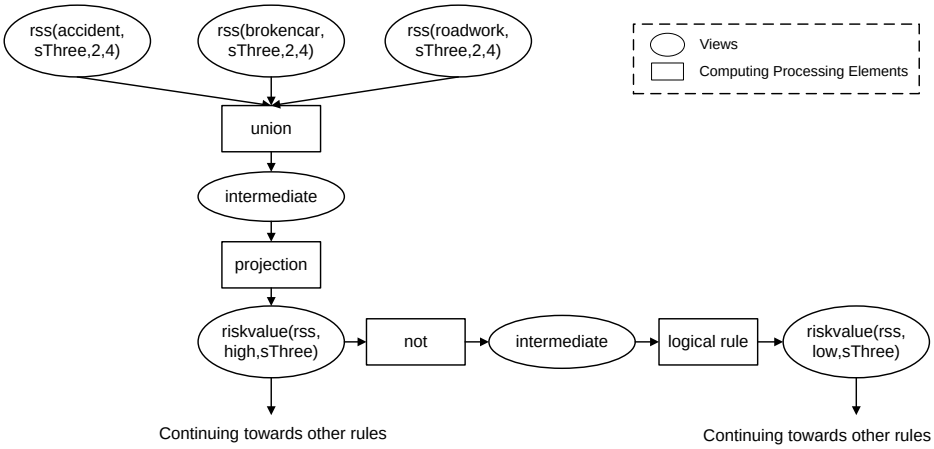


Figure 9.3: Workflow provenance graph after clustering based on Listing 9.4

Each of these rules contained in Listing 9.4 is transformed into a separate workflow provenance graph that exhibits causality between head and body predicates of a rule. Figure 9.2 shows four different initial workflow provenance graphs before clustering them. As already discussed, these provenance graphs do not contain any views representing a non-volatile predicate like `negative(accident)`.

Next, we apply the clustering mechanism based on types of rules. Figure 9.2 shows that provenance graphs for line 1, 2 and 3 in Listing 9.4 has the same predicate as output. As we have mentioned in Section 9.4, in case of a logic program, each predicate is represented by a single view. Since provenance graphs in line 1, 2 and 3 have the same output predicate, they must be grouped together into a single cluster to represent the output predicate by a single view. However, this grouping requires to split the corresponding logical rule into two activities/processing elements. We introduce an *union* activity to realize the aforesaid grouping operation. The other activity then projects/propagates appropriate predicates based on available input data products at a particular point in time. The left part of Figure 9.3 shows this clustering process. We extend this single cluster by unifying the provenance graph for line 4, shown in Figure 9.2. After this clustering, we get the intended workflow provenance graph, shown in Figure 9.3. From this figure, one can observe that a negated predicate is represented by introducing a *not* processing element in the workflow provenance graph. This *not* processing element has a variable input-output ratio and therefore it

requires special attention during fine-grained provenance inference phase which is discussed in the next section.

9.7 FINE-GRAINED DATA PROVENANCE INFERENCE

Fine-grained provenance inference facilitates the generated workflow provenance of a given model, i.e., a logic program, and the available input and output data products, i.e., input and output predicates in the answer set. In previous chapters, we discussed several inference-based methods to infer fine-grained data provenance (see Chapter 4, 5 and 6). Among this inference-based methods, the *multi-step probabilistic provenance inference*, presented in Chapter 6, can infer fine-grained data provenance for multiple processing steps, i.e., a complete workflow. There are multiple rules in the logic program which is involved in this case and these logical rules are transformed into a series of processing steps. Therefore, we apply the *multi-step probabilistic provenance inference* method, with a little adjustment because of the model characteristics, to infer fine-grained data provenance.

Fine-grained provenance inference has three phases: i) documentation of workflow provenance, ii) backward computation and iii) forward computation. In Section 9.6, we discuss the principle of extracting the workflow provenance of a given logic program. The workflow provenance graph represents the entire search space of the model and it is facilitated by the the next two phases of fine-grained provenance inference method. Please note that a database, consisting of input and output predicates, has to be created before executing the fine-grained provenance inference method. We create a SQLite[1] database that contains input and output data predicates as data tuples in respective views/tables. The size of the database may vary depending on the number of iterations the model is executed.

9.7.1 Backward Computation

This phase is executed once a scientist requests provenance of a predicate from the available answer sets. This predicate is known as chosen predicate/tuple. The logical timestamp associated with this predicate is referred to as *reference point*. Based on this *reference point*, the backward computation phase calculates the tuple boundary which refers to a time period with *up-*

1 Available at `http://www.sqlite.org/`

per bound and *lower bound*. Predicates falling in this range are considered as potential contributing input predicates producing the chosen output predicate. Since the model has multiple processing steps, the framework decides to apply the *multi-step probabilistic provenance inference* method to calculate the tuple boundary based on the decision tree as shown in Figure 7.1. However, we have to also consider the model characteristics discussed in Section 9.4 to calculate this boundary,. Based on this discussion, we can see that the entire workflow of the scientific model, developed in ASP, is executed at a time. Furthermore, the model is executed based on a logical clock and has no processing delay. Therefore, unlike the multi-step probabilistic provenance inference method, we need not consider accumulated window size and processing delay in this case. This approach is a variant of the multi-step probabilistic provenance inference method, that calculates the tuple boundary using the equations given below.

$$\mathtt{upperBound} = \mathtt{referencePoint}$$
$$\mathtt{lowerBound} = \mathtt{referencePoint} - \mathit{windowSize}$$

The variant of the multi-step probabilistic provenance inference method follows the same principle of the basic provenance inference method, discussed in Chapter 4, but can be applied over multiple processing steps with non-persistent, intermediate result.

9.7.2 Forward Computation

After calculating the tuple boundary, the forward computation phase reconstructs windows consisting of input data products/predicates that fall in that boundary. Then, it starts traversing the workflow provenance graph from those views representing input predicates and continues establishing data dependent relationship between predicates/views for each logical rule/processing elements until it reaches the chosen output predicate. It also retrieves appropriate data tuples which correspond to the input predicates from the database.

9.7.2.1 *Post-processing*

The forward computation phase removes the branches of the workflow provenance graph which are not directing towards the chosen predicate as

a post-processing of the inferred fine-grained data provenance. It also handles branches created for a negated body predicate as the one shown in line 4 in Listing 9.4. Figure 9.3 shows a *not* processing element that has been created to handle a logical rule with a negated body predicate. During this post-processing phase, if we reach at the view representing the predicate `riskvalue(rss, high, sThree)` (see Figure 9.3) by traversing the workflow provenance graph from one of the top nodes (an input predicate), it is confirmed that the model infers the existence of that particular `riskvalue` predicate. Then, we do not traverse towards the branch with *not* processing element rather we continue traversing to the next logical rule. The 'not-traversed' branch staring with a *not* processing element (see Figure 9.3) is removed from the graph. On the other hand, if the inference-based method infers that there exists no such predicate `riskvalue(rss, high, sThree)`, it always includes the branch staring with the *not* processing element.

Furthermore, post-processing phase also handles variable ratio processing elements. One of the variable ratio processing elements is a *selection* processing element (see Appendix A.3.4). Depending on a particular condition, a *selection* processing element could either infer an output predicate or not. Therefore, in the post-processing phase, the fine-grained provenance inference method determines whether a *selection* processing element satisfies the condition and thus, produces the output predicate. The other branches having non-satisfiable *selection* processing elements are pruned from the resulting fine-grained provenance graph.

Executing all these phases provides a fine-grained data provenance graph that explains the derivation history of the chosen output predicate. The fine-grained data provenance graph is a subset of the workflow provenance graph which represents the complete search space of the model.

9.8 EVALUATION

The purpose of evaluating the inference-based framework over a scientific model, developed using logic programming, is twofold. First, we would like to evaluate the efficiency of the inference-based framework over the explicit method of collecting provenance in a logic program, discussed in Appendix A.3. Second, we would like to see whether the resulting fine-grained data provenance graph is useful for debugging logic programs. We present a comparative analysis between the explicit provenance collection

method and the proposed inference-based method in terms of accuracy, execution time and storage consumption to measure the efficiency of the inference-based framework. Next, we present opinions from two experts in the logic programming domain from Digital Enterprise Research Institute (DERI), about the usefulness of the inferred fine-grained data provenance graph for debugging logic programs.

To compare accuracy, execution time and storage costs between the explicit and the inference-based method, we execute the logic program which is developed in ASP for 100, 200, 300 and 500 iterations with randomly generated synthesized data representing twitter status, rss message and weather updates.

9.8.1 Accuracy

The accuracy of the inferred fine-grained data provenance graph is measured against the explicit provenance graph which represents the ground truth. The explicit provenance graph contains base facts/non-volatile predicates, which are static and not considered in the inferred fine-grained data provenance graph since the grounding mechanism does not explicate any base facts/non-volatile predicates. Non-volatile predicates could be added to the inferred provenance by extending the technique of static analysis of the logic program. Due to lack of time, we have not done this step. Therefore, we remove nodes representing non-volatile predicates/base facts from the explicit provenance graph while comparing it to the corresponding inferred fine-grained data provenance graph.

An inferred fine-grained data provenance graph for a particular output predicate is considered to be accurate if the set of input predicates/input data products of both explicit and inferred provenance graph are equivalent. Mathematically, it can be referred to as $accuracy_i$ where i indicates the i-th output predicate. If an inferred provenance graph is accurate, the value of $accuracy_i$ is set to 1, otherwise 0. The average accuracy of the inference-based method can be measured by computing the accuracy of the inferred provenance graphs for all output predicates. The average accuracy is calculated using the following equation assuming there are n number of output predicates.

$$\textit{Average accuracy} = (\frac{\sum_{i=1}^{n} accuracy_i}{n} \times 100)\%.$$

Table 9.3: Comparison of Execution Time (in seconds)

No. of iterations	Inference-based Method	Explicit Method	Ratio
100	18.2	58.1	1:3.2
200	57.4	166.9	1:2.9
300	115.1	329.6	1:2.9
500	280.3	out of memory	-

In all test cases, the average accuracy of the inference-based method is 100% which indicates that all inferred provenance graphs match exactly the corresponding explicit provenance graph. In this case, 100% accuracy is expected. Since parameters like processing delay and sampling interval do not influence the execution because of the model characteristics as discussed in Section 9.4, there are no possibilities to occur any error during the inference phase and hence, we achieve 100% accuracy.

9.8.2 Execution Time

The execution time of a logic program depends on the number and complexity of logical rules. The inference-based method infers fine-grained data provenance by facilitating the workflow provenance graph which is generated by the grounding of the logic program. However, the explicit provenance collection method has to add extra logical rules, encoding provenance information of the logic program. Therefore, the version of the logic program executed by the explicit provenance method has a higher number of logical rules then the original version of the logic program. Therefore, the explicit provenance method incurs overhead in terms of execution time due to the inclusion of extra logical rules. Table 9.3 shows a comparison between these two methods. One can observe that the original version of logic program executes around 3 times faster than the extended version documenting provenance information explicitly. For the last test case with 500 iterations, the extended logic program with extra logical rules, encoding provenance information, cannot even finish the execution because of a memory allocation error in oClingo.

Table 9.4: Comparison of Storage Consumption (in KB)

No. of iterations	Inference-based Method	Explicit Method	Ratio
100	14	181	1:13
200	23	347	1:15
300	29	533	1:18
500	57	out of memory	-

Therefore, the execution of the logic program with extra logical rules as used by the explicit provenance collection method is much slower and in case of a higher number of iterations of the program, it is simply not possible to document explicit provenance information.

9.8.3 Storage Space Consumption

The storage consumption is measured by comparing the size of the SQLite databases that hold both input and output predicates of the logic program in form of relational data tuples. Both the explicit provenance collection method and the inference-based method, require to materialize all input and output predicates into a database. Furthermore, the explicit provenance method materializes all predicates, encoded with provenance information, as tuples into the database. As a consequence, the storage overhead of the explicit provenance method is dependent on the number of logical rules for which provenance has to be encoded. Table 9.4 shows the disk space consumed by these two methods.

From Table 9.4, it is evident that the explicit provenance method has several magnitudes of storage overhead compared to the inference-based method. The explicit provenance method takes at least 13 times more storage than the inference-based method. In case of the inference-based method, the storage consumption only depends on the number of input and output predicates. It is independent on the window size and size of intermediate results produced by processing elements within the logic program. Therefore, the larger the window size and the higher the number of processing elements, the more storage space can be saved by applying the inference-based method.

9.8.4 Ease of Debugging

In case of a logic program, the workflow provenance graph represents the entire search space of the program. The workflow provenance graph identifies all possible data dependent relationship between predicates. The fine-grained provenance graph is a subset of workflow provenance graph that explains the derivation of a particular output predicate. Both provenance graphs could be used for debugging an ASP program. In this regard, we would like to highlight a point raised by one of the ASP domain experts during our discussion with them.

> *It would be much easier to debug an ASP program if I (domain expert) have the opportunity to look at the connections between predicates in the search space. In this case, I (domain expert) could easily understand how constraints influence the outcome of the program and this knowledge could be facilitated to obtain correct answer sets.*

Based on the raised point, we asked two ASP domain experts about how provenance graphs could be useful for debugging an ASP program. Their feedback is presented below. Please note that we are aware that this is not a representative usability evaluation, but it gives an indication on the applicability of provenance graphs for debugging. Conducting an extensive usability study could be a potential future work.

Feedback of ASP domain experts

The workflow and fine-grained provenance graphs provide useful insights to both the logic programmer and the domain expert. The workflow provenance graph captures the way data is related in the search space and facilitates the understanding of these connections. It also helps identifying how constraints should be added or removed to reduce or to expand the set of correct solutions. The fine-grained provenance graph is a subset of the workflow provenance graph that explains the complete derivation history of a chosen fact in an answer set. It allows to verify the correctness of the rules for modeling a particular domain. Fine-grained data provenance provides the complete derivation history of an out-coming fact. Therefore, this graph can be also useful to achieve reproducible results at the instance-level. All these insights facilitate early-stage *instance-driven* debugging of an ASP program.

9.9 DISCUSSION

The inference-based framework, discussed in this thesis, is applied over the scientific model which determines accessibility of road segments, developed in ASP. The evaluation of inference-based framework in the logic programming domain shows that the framework can infer accurate provenance information without any storage overhead faster than the method of documenting provenance information explicitly. Furthermore, a brief discussion with two ASP domain experts recommends that provenance graphs should be considered as an additional tool for debugging ASP programs in the initial modeling phase, when domain experts and ASP programmers sit together to formulate a problem description. The mutual understanding of how and why certain inputs generate a certain output can produce a better formulation of a logic program faster.

Negation as failure

However, there is a limitation of the inference-based framework in the context of logic programming domain. One of the major challenges is to represent Negation as Failure (NAF) logical rules in a provenance graph. It is a non-monotonic inference rule that is used to derive a negation of a fact from failure to derive that fact. In the current implementation of the inference-based framework, we represent NAF as a *NOT* processing element in the provenance graphs which cannot entirely reflect the semantics of this rule. The power of NAF would need to be addressed if the goal is to convey the semantics of the program which is beyond the scope of the inference-based framework. The target of the inference-based framework is to infer accurate provenance of scientific models at reduced storage costs without annotating semantics of the domain. Domain experts, scientists, developers of scientific models could then interpret provenance graphs by facilitating their expertise on that domain or application.

In this connection, we acknowledge the benefits of ongoing work on ASP debugging from a semantic perspective and the added value of IDE tools. As a potential future work, we will be exploring how data provenance can be embedded in those work for the next phases of ASP development.

9.10 SUMMARY

In this chapter, we presented a case study applying the proposed inference-based framework over a logic program which is developed in Answer Set

Programming (ASP), determines accessibility of road segment in a particular area. At the beginning, we presented a background of ASP briefly along with the relevant ASP syntax to formulate logical rules. Then, we introduced the use case of the model that facilitates twitter, rss and weather update data streams to infer the current traffic condition of road segments. Next, we identified the key characteristics of the model followed by the discussion of differences between two case studies presented in this thesis. In this case, the scientific model is developed in ASP, a declarative language, which is a provenance-unaware platform. Therefore, we applied the workflow provenance inference method to extract a workflow provenance graph automatically based on the grounding of the logic program. Later, the workflow provenance graph is used during the fine-grained provenance inference phase. Based on the input predicates at a particular point in time, we traversed through the workflow provenance graph to infer the fine-grained data provenance graph which can explain the derivation history of a particular output predicate of the model. Afterward, we presented both quantitative and qualitative performance analysis of inference-based framework on this use case. The quantitative analysis shows that the inference-based framework can infer accurate provenance information without any storage overhead faster than the method of documenting provenance information explicitly. Furthermore, we arranged an open-ended discussion session with two ASP domain experts mainly focusing on the usage of provenance graphs as a debugging tool in ASP. The outcome of this discussion is promising and it suggests that provenance graphs should be considered as an additional tool for debugging ASP programs.

10

CONCLUSION

Provenance of data products is a widely-studied research topic, attracting much attention from researchers now-a-days because of its several applications. Workflow provenance explicates the relationship among different activities within a scientific model. However, it cannot explain the complete derivation history of a particular data product/tuple. Fine-grained data provenance documents the relationship among contributing input data products, activities and produced output data products. Both workflow and fine-grained data provenance help scientists to validate their scientific models as well as to trace a particular output data product back to its contributing input data products. Therefore, a framework for managing both workflow and fine-grained data provenance at reduced cost in terms of time, training and storage consumption would be a timely solution for scientists who want to manage data provenance for their models.

Goal of the thesis

This thesis began with the goal of developing a generic, cost-efficient, self-adaptable provenance management framework. We envisioned a framework that can automatically extract workflow provenance from a scientific model built in a provenance-unaware platform, reducing effort of scientists to define workflow provenance of the model manually or by facilitating a specialized tool in terms of time and training. We also focused on the challenges of collecting fine-grained data provenance at reduced storage costs under different execution environments. Finally, the envisioned framework should be self-adaptable so that it can adjust the complete mechanism of extracting provenance information based on the characteristics of a given scientific model and underlying execution environment to always provide highly accurate provenance information.

In this chapter, we explain contributions of this thesis in the context of the research questions posed in Section 1.4. Finally, we briefly describe possible future work in this research direction.

10.1 CONTRIBUTIONS

This thesis centers around the primary research question which is given below.

> *Primary Research Question (RQ): How to manage data provenance with minimal user effort in terms of time and training and at reduced storage consumption for different kinds of scientific models?*

We divided the primary research questions into three research questions, satisfying each of them would lead us towards accomplishing such a framework managing data provenance. Next, we explain our contributions in the light of these research questions.

10.1.1 Capturing Workflow Provenance Automatically

Problem

There are existing tools and techniques that can extract workflow provenance of a scientific model [102, 84, 24, 116, 77]. These systems are referred to as *provenance-aware* platforms. Each of these tools has their own set of programming constructs and operators that can define a set of activities within a scientific model. These provenance-aware tools require scientists to re-define the activities in their scientific model which might have been developed in a *provenance-unaware* platform like high-level programming languages, for extracting workflow provenance. Re-defining the activities according to the constructs and operators of a provenance-aware platform may not only take a considerable amount of time but also require extensive training effort for scientists, adapting them to a particular platform. The first research question, *RQ 1*, focuses on this challenge.

> *RQ 1: How to capture automatically the workflow provenance of a scientific model developed in a provenance-unaware platform at reduced cost in terms of time and training?* (Chapter 3)

To address this challenge, we have proposed a novel technique, called *workflow provenance inference*, to capture workflow provenance automatically based on a given program which is used for actual processing of

a scientific model. In this connection, we have introduced a workflow provenance model, representing activities and data products as well as data-driven relationships between them as a graph. We have explained the working principle of workflow provenance inference method in the context of Python programs. First, an initial workflow provenance graph has been generated explicating relationships between Python operations (activities) and variables (data products). The initial workflow provenance graph might have control-flow based coordination between activities. Therefore, we have defined a set of re-write rules and applied them over the initial provenance graph to transform control dependencies into data dependencies and this process achieves the workflow provenance graph. *Solution*

We have evaluated the workflow provenance inference method over a diverse set of Python programs. Our evaluation has shown that the workflow provenance inference method is capable of generating provenance graphs for all 16 programs considered for the experiment. The accuracy of these generated provenance graphs are checked manually by experts in Python programming. For a complex program with more than 2 levels of nested compound statements (e.g. conditional branching), the provenance graph becomes very large and hence, it becomes difficult to check the accuracy of that particular provenance graph manually. Therefore, we narrow down the scope the experiment calculating accuracy by only considering programs with at most 2 levels of nested compound statements. There are 10 programs that satisfy the criterion and we have found that all 10 provenance graphs generated for these programs are accurate. *Results*

The workflow provenance inference method can reduce the effort of scientists in terms of time and training compared to other systems collecting workflow provenance where scientists need to put a lot of effort to learn the know-how of those systems. Therefore, the workflow provenance inference method, discussed in Chapter 3, definitely provides a solution, answering *RQ 1*.

10.1.2 Inferring Fine-grained Data Provenance

Workflow provenance explicates data-driven relationships between activities within a scientific model. However, it cannot explain the derivation of a particular data product, produced by executing that scientific model. Fine-grained data provenance documents the derivation history of output data products, allowing scientists to trace back to source data products *Problem*

or to reproduce the results. Existing annotation-based fine-grained provenance collection systems [21, 131, 109, 28, 15, 60, 56] documents provenance information explicitly which incurs a considerable amount of storage overhead for maintaining provenance data. Especially, in case of applications processing data streams, the storage consumed by provenance data can be several orders of magnitude higher than the storage required by the collection of actual data products. Existing systems addressing fine-grained data provenance for data streams [103, 129, 58, 108] maintain persistent provenance information and thus, incur a significant storage overhead. The second research question, *RQ 2*, highlights the challenge of maintaining data provenance at reduced storage costs.

> *RQ 2: How to infer fine-grained data provenance under different system dynamics at reduced cost in terms of storage consumption?* (Chapter 4, 5 and 6)

Solution

In this thesis, we have proposed *fine-grained data provenance inference* methods that can satisfy any fine-grained provenance request for a particular output data product by facilitating workflow provenance of the related scientific model and timestamps attached to the data products. The proposed inference-based methods do not maintain any persistent provenance storage rather the inference-based methods infer data dependencies between input and output data products by using timestamps attached to data products as the key element. Therefore, inference-based methods have less storage overhead, i.e., only keeping timestamps, to manage provenance data compared to explicit provenance collection techniques.

We have proposed three inference-based methods that can infer fine-grained data provenance as discussed in Chapter 4, 5 and 6. While the general principle of the inference mechanism remains the same in each method as discussed before, each inference-based method is suitable to handle a particular situation which is referred to as system dynamics. System dynamics includes a set of parameters such as *processing delay* and *sampling interval*. *Processing delay* refers to the amount of time required to execute a processing element/activity. *Sampling interval* refers to the amount of time between two successive arrivals of input data products in case of data streams. The *basic provenance inference* method, discussed in Chapter 4, is the most straightforward inference mechanism suitable for an environment where activities have constant processing delays and data products arrive

at a constant sampling interval. In Chapter 5, we have presented the *probabilistic provenance inference* method. This inference-based method is capable of inferring fine-grained data provenance under variable processing delay and variable sampling interval. Finally, we have proposed the *multi-step probabilistic provenance inference* method in Chapter 6. This inference-based method is an extension of the probabilistic provenance inference method and can handle a processing chain with non-persistent intermediate views under variable processing delays and variable sampling intervals.

We have evaluated each inference-based method in terms of storage consumption and accuracy of provenance information using both real datasets and simulations. Evaluation results reported in Chapter 4, 5 and 6 have proved that inference-based methods take less storage space than explicit provenance collection techniques. The ratio of the storage consumption between inference-based methods and explicit provenance collection method depends on the actual processing and we refer to this ratio as *storage gain*. The longer the processing chain of a scientific model and the bigger the input datasets, the higher the storage gain by using inference-based methods. Inference-based methods also achieve a higher magnitude of storage gain when a particular input data product contributes several times during the processing. This scenario is quite common in stream data processing with windowing constructs. Our evaluation shows that in case of processing data streams with sliding windows, the bigger the window size and the larger the overlaps between windows, the higher the storage gain using inference-based methods. *Results*

Furthermore, the inference-based methods infer around 90% accurate provenance information in most test cases with variable processing delay and variable sampling interval. If the processing delay remains constant or data is processed offline, the inference-based methods infer 100% accurate provenance. In sum, the proposed fine-grained provenance inference methods infer highly accurate provenance information at reduced storage costs under different execution environments, satisfying *RQ 2* of this thesis.

10.1.3 Incorporating Self-adaptability into the Framework

Existing provenance-aware systems are developed to address a particular application or a particular settings. One of the goals in this thesis is to achieve a generic framework to infer both workflow and fine-grained data provenance. To accomplish this goal, we have proposed the work- *Problem*

flow provenance inference method that can capture workflow provenance from a given program automatically. Furthermore, we have presented a suite of fine-grained provenance inference methods which can build provenance traces without maintaining explicit provenance data. Each inference-based method we proposed, is suitable to handle a particular environment. Therefore, it is important to be able to adapt the complete mechanism of extracting provenance information by the framework itself based on the characteristics of a given scientific model and underlying environment to provide highly accurate provenance information always. The last research question, *RQ 3*, introduces this challenge.

> *RQ 3: How to incorporate the self-adaptability into the framework managing data provenance at reduced cost?* (Chapter 7)

Solution

The characteristics of a scientific model should be examined first to accomplish this challenge. A scientific model could be developed either in a platform which collects provenance automatically or in a platform that has no provenance support. The former is referred to as *provenance-aware* platform, while the later is referred to as *provenance-unaware* platform. Depending on the *model developing platform* used, the framework can decide whether to apply the workflow provenance inference method, i.e., for models developed in a provenance-unaware platform, or not. Next, a fine-grained provenance request for a particular output data product can be sent to the framework based on the availability of workflow provenance and input data products. At this time, first, *types of activities* should be taken into consideration. Activities that produce a fixed number of output data products by processing a fixed number of input data products at each execution are referred to as *constant ratio* activities such as project, aggregate functions, cartesian product etc. Activities that do not hold this condition are referred to as *variable ratio* activities such as select operation. Fine-grained data provenance inference methods proposed in this thesis, are directly applicable over *constant ratio* activities. To handle *variable ratio* activities, a slight change during data processing should be adopted by the framework. Furthermore, the proposed inference-based framework should also consider *types of input data products*, i.e., either offline data (non-streaming) or data streams, to apply a particular inference-based method. Finally, *system dynamics*, describing parameters like processing delays and sampling intervals, also play an important part to infer accurate provenance by applying the most appropriate inference-based method.

We have considered the aforesaid characteristics of a scientific model and parameters describing execution environment to accomplish a *self-adaptable* framework. We have introduced a decision tree which has been facilitated to take the decision of which inference-based method should be executed to infer both workflow provenance and fine-grained data provenance based on the given model and current system dynamics. The outcome of this decision making process allows the framework to be self-adaptable and therefore, answers *RQ 3*.

10.1.4 Evaluating the Framework

We have evaluated the framework on the basis of two use cases as described in Chapter 8 and 9. The first case study involves a scientific model for estimating the global water demand [127], discussed in Chapter 8, developed by the researchers from Utrecht University, The Netherlands. This scientific model processes offline geospatial data, i.e., raster maps with timestamps, collected from various sources and produces raster maps representing global water demand from the year 1960-2000 at a monthly temporal resolution. The model is developed in the Python programming language. In this case study, first, we have applied the proposed workflow provenance inference method to extract workflow provenance information as discussed in Chapter 3. Later, we annotate appropriate nodes of the workflow provenance graph with actual values of input and output data products by applying the multi-step probabilistic provenance inference method. Our proposed framework has achieved 100% accurate fine-grained data provenance without having any storage overhead. Furthermore, we have conducted an interview with scientists who developed this model about the applicability of the framework. The scientists have admitted that both workflow and fine-grained provenance graphs are useful for debugging purposes. Especially, fine-grained provenance graphs can be used to easily identify the sources of an unexpected value in the output. We have also built a prototype that demonstrates the complete framework based on this use case [73]. *Case I*

The other case study is about estimating the degree of accessibility of a particular road segment, discussed in Chapter 9, developed by the scientists from Digital Enterprise Research Institute (DERI), Ireland. The scientific model realizing this use case, processes data streams collected from various sources like twitter, rss feeds, weather update sites etc. and pro- *Case II*

duces the accessibility rating of road segments in a particular city. The model is developed using a declarative language - Answer Set Programming (ASP). The scientific model used in this use case has a few distinctive features compared to the previous one. First, it processes data streams whereas the model used in first case study processes offline data. Second, this model is developed using ASP, a class of declarative languages while the model used in first case study is developed using Python, a procedural language. The difference in model developing platform introduces other differences such as this scientific model allows non-deterministic results unlike the model used in first use case.

Since ASP has no built-in provenance support, i.e., provenance-unaware platform, we have applied the workflow provenance inference method to extract the workflow provenance graph automatically. To address the characteristics of a logic program, developed in ASP, we have introduced new techniques to extract workflow provenance from the given program, discussed in Chapter 9. Next, we have also inferred fine-grained data provenance for a particular output data product/predicate by applying the multi-step probabilistic provenance inference method. The proposed framework has achieved 100% accurate provenance at less storage consumption compared to explicit provenance collection methods. Furthermore, we have reported opinions of two ASP domain experts mainly focusing on the potential usage of provenance graphs as a debugging tool in ASP. The outcome of this discussion is that both workflow and fine-grained provenance graphs provide indication about the correctness of the model as well as explain dependencies between data products/predicates.

These two case studies demonstrate the applicability and suitability of the proposed inference-based framework managing data provenance in the context of scientific data processing models. Developing such a framework satisfies the primary research question presented in this thesis.

10.2 FUTURE WORK

The proposed framework is capable of inferring workflow provenance from a given scientific model. Furthermore, given an output data product, it can infer the set of input data products which contribute to produce that particular output and thus, is capable of providing a fine-grained provenance trace at reduced storage costs. We believe that the framework can

serve as a solid platform for scientists who want to manage data provenance of their scientific experiments. In this regard, we identify several future work and research directions which could definitely improve the framework and also broaden the application domain of the framework.

10.2.1 Improvement of Workflow Provenance Inference Methods

The proposed workflow provenance inference method is capable of handling compound statements such as conditional branching (if-else), iterative operations (for loop) etc. In case of an iterative operation like looping, the workflow provenance inference method handles one looping operation at a time and infers its data dependencies. Sometimes, there could be several levels of nested iterations. In this case, it is possible that data dependencies in an inner loop could influence the outer loop characteristics. Currently, the proposed workflow provenance inference method does not address data dependencies occurring from such dependencies. Identifying these influences requires in-depth analysis of different looping techniques and possibly manual annotations. This is an interesting problem and we consider this as future work to achieve a more complete workflow provenance inference method.

Nested iterations

Furthermore, the workflow provenance inference method does not address any recursive operations at its current stage. The biggest challenge of handling recursive operations is to figure out the exact data-flow based coordination between associated activities. Since same variables with different values are used at different stages of a recursive function, it is difficult to transform control-flow based coordination into data-flow based coordination. Furthermore, it is also difficult to identify the end of execution of a recursive operation without interpreting the values of participating data products. For the aforesaid reasons, extracting workflow provenance from a recursive operation is a challenging task. Furthermore, recursive operations have also implications over the fine-grained data provenance graph. Due to repetitive nature of processing, the resulting fine-grained data provenance graph could become very large and complex to understand. Addressing recursive operations in the proposed framework could stimulate a new dimension of research in this domain.

Recursive operations

10.2.2 Assessment of Applicability of Fine-grained Data Provenance Methods

We have presented three methods to infer fine-grained data provenance in this thesis. Each of them is suitable for a particular setting to infer highly accurate fine-grained data provenance. These inference-based methods can reduce storage overhead to maintain provenance data in a great deal. However, our evaluation has shown that sometimes these inference-based methods could provide inaccurate provenance. Therefore, a cost metrics comparing proposed inference-based methods and other explicit provenance collection methods in terms of storage consumption and level of accuracy of provenance information could become beneficial for users of the proposed framework. In that case, users can get a hint about the efficiency of inference-based methods and can also compare their performance before actually applying any particular method. In future, we would like to complement the proposed framework by introducing such a cost metrics.

10.2.3 Extending Prototype of the Framework

Semantics of Provenance Graphs

We have developed a prototype[1] demonstrating the proposed framework [73]. The prototype integrates both workflow and fine-grained data provenance inference methods, proposed in this thesis. Sometimes, a workflow provenance graph could become very large depending on the program. It is also true in case of a fine-grained data provenance graph. These provenance graphs show data-driven relationships between activities with values of contributing data products. However, provenance graphs explicate neither semantics of activities nor metadata pertaining to data products like data source, data structure etc. The user has to interpret and understand the meaning of different activities and contributing data products. In this regard, if the user has expertise in that particular domain or if he/she is working on that scientific model, interpretation of provenance graphs could be easier. However, for a user working on other domains, interpretation of provenance graphs without semantic information could become a tedious job. Furthermore, including semantics into provenance graphs could make provenance information more useful for debugging purposes. In future, we would like to conduct further investigation in this direction.

1 Available at `https://github.com/rezwan4438/ProvenanceCurious`

Currently, the prototype can only handle Python programs to capture workflow provenance automatically. The core concept behind this inference method is to transform control dependencies between activities into data dependencies. The general principle of transforming dependencies could be applied over other block-structured programming/scripting languages such as Java, PHP etc. Extending the implementation of workflow provenance inference method to address these languages requires to integrate appropriate grammar of a particular language so that the method can parse a program written in that language and can capture the data dependencies between activities. This could be a potential addition to the current implementation of the proposed framework.

Addressing more languages

The current implementation of the framework facilitates a workflow provenance model to explicate data dependencies between different activities as a graph. We have kept the provision of annotating appropriate nodes of the workflow provenance graph with actual values in our provenance model so that the derivation history of a particular data product can be explained which we refer to as fine-grained data provenance graph. The workflow provenance model can be easily extended to adopt the primitives defined by PROV-DM [95]. PROV-DM is the conceptual data model that forms a basis for the W3C provenance (PROV) specification. A quick comparison shows that the proposed workflow provenance model has nodes similar to *entities* and *activities* as defined in PROV-DM. However, the workflow provenance model has no similar concept like *agent* type as discussed in the PROV-DM specification. Since the aim of this thesis is to extract provenance information efficiently, the provenance representation is not the focus of this work. Extending the framework on the basis of PROV specification to represent provenance graphs would surely increase the applicability and interoperability of the complete framework. Therefore, this is one of the major tasks which will be addressed in the future.

Provenance representation

10.2.4 Broadening the Application Domains of the Framework

Big data is the buzz word in these days. According to a recent article published by IBM[2], 2.5 quintillion bytes (2500 petabytes) of data are created everyday. This data comes from different sources: sensor data ranging from a simple data tuple to high resolution satellite images, posts to so-

Big data

2 Available at http://www-01.ibm.com/software/data/bigdata/

cial networking sites, meter readings of power consumption etc. In most cases, big data is unstructured or semi-structured data. Since a majority of existing provenance-aware systems are designed and developed over relational data in the backend, maintaining provenance in big data requires novel techniques. Furthermore, storage overhead of provenance information poses another challenge especially in big data case. It would be worth investigating the big data provenance - *big provenance* problem as well as to understand the opportunities of developing an inference-based provenance management framework addressing use cases involving big data.

Social networking

There are a few existing research in security and privacy domain that facilitate provenance of data products as a basis to compute the trustworthiness of a particular machine/node. Similar to this concept, provenance could also play an important role to establish trust over other users in a social networking platform. Everyday, we used to get a lot of posts from other users in our personalized news feed. Some of them could be spam. Provenance information associated with a post would easily identify the source (user) of the spam (post) and therefore, we could mark that user (source of the spam) as a less trustworthy. Since a particular post could be shared a thousand times by different users, annotation-based provenance would consume a lot of storage. Therefore, we would like to investigate the chance of inferring provenance information in a social networking platform.

Software engineering

Provenance information could be used for debugging the results of a scientific data processing model as discussed in this thesis. It could be also possible to apply the concept of program slicing [130], introduced in software engineering domain, over a provenance graph to understand better the impact of a particular data product. Furthermore, inclusion of semantics of different activities and data products in a provenance graph could make it more meaningful for users. Therefore, we could see the potential of the proposed framework as a debugging tool complementing the classical debugging techniques of IDEs. More investigation in this direction is one of the future work.

Furthermore, in a declarative settings such as Answer Set Programming (ASP), there are a few existing work that explain the logic program from a semantic perspective. In future, we will explore how the functionalities of the proposed framework can be embedded in those work for the next phases of ASP development.

APPENDIX

A.1 GRAPH RE-WRITE RULES IN RULEML

We have encoded graph re-write rules, discussed in Chapter 3, in a rule notation scheme, RuleML[3]. As an example, RuleML encoding of the graph re-write rule *GM 2.a* (see Section 3.7) is given below.

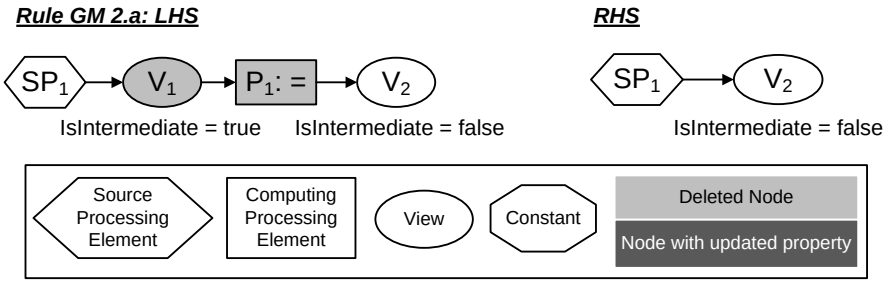


Figure A.1: Graph re-write rule *GM 2.a*

RuleML encoding of re-write rule *GM 2.a*

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleML xmlns="http://ruleml.org/spec"
      xmlns:xsi="http://www.w3.org/2001/
      XMLSchema-instance"
 xsi:schemaLocation="http://ruleml.org/
       spec http://ruleml.org/1.0/xsd/
       datalog.xsd">

 <Assert mapClosure="universal">
  <!-- Rule GM 2.a: adding new edge -->
  <Implies>
   <if>
    <!-- explicit 'And' -->
    <And>
     <Atom>
      <op>
       <Rel>SourcePE</Rel>
      </op>
      <Var>id1</Var>
      <Var>name1</Var>
      <Var>line1</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>ComputingPE</Rel>
      </op>
      <Var>id2</Var>
      <Ind>=</Ind>
      <Var>line2</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>View</Rel>
      </op>
      <Var>id3</Var>
      <Var>name3</Var>
```

3 Available at `http://ruleml.org/`

```
      <Ind>intermediate</Ind>
      <Var>persistent3</Var>
      <Var>line3</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>View</Rel>
      </op>
      <Var>id4</Var>
      <Var>name4</Var>
      <Ind>non-intermediate</Ind>
      <Var>persistent4</Var>
      <Var>line4</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>Edge</Rel>
      </op>
      <Var>id1</Var>
      <Var>id3</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>Edge</Rel>
      </op>
      <Var>id3</Var>
      <Var>id2</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>Edge</Rel>
      </op>
      <Var>id2</Var>
      <Var>id4</Var>
     </Atom>
    </And>
   </if>
   <then>
    <Atom>
     <op>
      <Rel>New_Edge</Rel>
     </op>
     <Var>id1</Var>
     <Var>id4</Var>
    </Atom>
   </then>
  </Implies>
  <!-- Rule GM 2.a: removing V1 -->
  <Implies>
   <if>
    <And>
     <Atom>
      <op>
       <Rel>SourcePE</Rel>
      </op>
      <Var>id1</Var>
      <Var>name1</Var>
      <Var>line1</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>ComputingPE</Rel>
      </op>
      <Var>id2</Var>
      <Ind>=</Ind>
      <Var>line2</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>View</Rel>
      </op>
      <Var>id3</Var>
      <Var>name3</Var>
      <Ind>intermediate</Ind>
      <Var>persistent3</Var>
      <Var>line3</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>View</Rel>
      </op>
      <Var>id4</Var>
      <Var>name4</Var>
      <Ind>non-intermediate</Ind>
      <Var>persistent4</Var>
      <Var>line4</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>Edge</Rel>
      </op>
      <Var>id1</Var>
      <Var>id3</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>Edge</Rel>
      </op>
      <Var>id3</Var>
      <Var>id2</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>Edge</Rel>
      </op>
      <Var>id2</Var>
      <Var>id4</Var>
     </Atom>
```

```
    </And>
   </if>
   <then>
    <Atom>
     <op>
      <Rel>Remove_Node</Rel>
     </op>
     <Var>id3</Var>
    </Atom>
   </then>
  </Implies>
  <!-- Rule GM 2.a: removing P1 -->
  <Implies>
   <if>
    <And>
     <Atom>
      <op>
       <Rel>SourcePE</Rel>
      </op>
      <Var>id1</Var>
      <Var>name1</Var>
      <Var>line1</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>ComputingPE</Rel>
      </op>
      <Var>id2</Var>
      <Ind>=</Ind>
      <Var>line2</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>View</Rel>
      </op>
      <Var>id3</Var>
      <Var>name3</Var>
      <Ind>intermediate</Ind>
      <Var>persistent3</Var>
      <Var>line3</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>View</Rel>
      </op>
      <Var>id4</Var>
      <Var>name4</Var>
      <Ind>non-intermediate</Ind>
      <Var>persistent4</Var>
      <Var>line4</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>Edge</Rel>
      </op>
      <Var>id1</Var>
      <Var>id3</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>Edge</Rel>
      </op>
      <Var>id3</Var>
      <Var>id2</Var>
     </Atom>
     <Atom>
      <op>
       <Rel>Edge</Rel>
      </op>
      <Var>id2</Var>
      <Var>id4</Var>
     </Atom>
    </And>
   </if>
   <then>
    <Atom>
     <op>
      <Rel>Remove_Node</Rel>
     </op>
     <Var>id2</Var>
    </Atom>
   </then>
  </Implies>
 </Assert>
</RuleML>
```

A.2 CASE STUDY I : MEETING MINUTES

In Chapter 8, we have presented a case study that demonstrates the viability of the proposed inference-based framework over a scientific model estimating global water demand [127], developed in Python programming language. To perform this case study, we had several meetings with scientists developing this scientific model. In this section, we provide a brief transcript of these meetings.

A.2.1 Introductory Meeting

The scientific model estimating global water demand is developed by a group of researchers working at the physical geography department, Faculty of Geosciences in the Utrecht University (UU). We had been introduced to them by Bram (B. D. van der Waaij) who had been collaborating with the group of researchers from Utrecht University in the context of the GLOWASIS[4] project. Table A.1 shows the date, time, place, duration and participants of this meeting.

Table A.1: Introductory Meeting

Goal	Discussion on the scope of collaboration
Date and Time	January 13, 2012 at 10:30 hours
Place	Utrecht University
Duration	Around 90 minutes
Participants	Prof. dr. M. F. P. Bierkens, UU Dr. L. P. H. van Beek, UU Y. Wada, UU B. D. van der Waaij, TNO Groningen Dr. A. Wombacher, UT M. R. Huq, UT

4 Available at `http://glowasis.eu/`

Meeting Transcript

After the introduction, Andreas (Dr. A. Wombacher) gave a short presentation about our research interests, focusing on data provenance. Andreas explained the applications of fine-grained data provenance especially for data intensive scientific models which includes debugging outputs of the scientific model, i.e., tracing an erroneous value back to it's source data. Afterward, Marc (Prof. dr. M. F. P. Bierkens) also explained their work briefly followed by an open discussion on possible future collaborations between Utrecht University and University of Twente. Researchers from Utrecht University accepted the fact that the provenance information could possibly help them to find out the origin of an abnormal/missing value. Later, both parties agreed to collaborate with each other to perform a case study. It was decided that Yoshi (Y. Wada) would provide related source code of the scientific model alongside the input and output files (data products) of the model and Rezwan (M. R. Huq) would develop a prototype based on their work reported in [70, 71, 72] which could infer provenance information based on the given scientific model and available data products.

A.2.2 Model and Data Collection

The next meeting was held to collect the source code (Python program) of the scientific model estimating global water demand and the available input and output data products. Table A.2 shows the date, time, place, duration and participants of this meeting.

Table A.2: Model and Data Collection Meeting

Goal	Collection of programs and original datasets
Date and Time	February 06, 2012 at 11:00 hours
Place	Utrecht University
Duration	Around 60 minutes
Participants	Y. Wada, UU M. R. Huq, UT

Meeting Transcript

During this meeting, Yoshi (Y. Wada) provided the source code (Python program) of the model and the available datasets used in this model. The datasets consist of a collection of more than 3000 PCRaster[5] files. Rezwan (M. R. Huq) collected the Python program and datasets in an external hard drive. Meanwhile, Rezwan asked a few questions to Yoshi about the structure of files, type of methods used in the program etc. Yoshi answered these questions. Each file has 360×720 values, representing a particular type of data (e.g. crop factor, irrigated areas etc.), collected over the whole world. Over this data, PCRaster operations (methods) are executed to compute the desired value. Both parties agreed that they would sit together for another meeting as soon as Rezwan had some initial results.

A.2.3 Initial Evaluation

This meeting was held to report the preliminary results, i.e., fine-grained data provenance trace, followed by an evaluation of the results by the researchers from Utrecht University. Table A.3 shows the date, time, place, duration and participants of this meeting.

Table A.3: Initial Evaluation Meeting

Goal	Reporting preliminary results and initial evaluation
Date and Time	March 02, 2012 at 11:30 hours
Place	Utrecht University
Duration	Around 90 minutes
Participants	Dr. L. P. H. van Beek, UU Y. Wada, UU Dr. A. Wombacher, UT M. R. Huq, UT

5 Available at `http://pcraster.geo.uu.nl/`

Meeting Transcript

This session was divided in three parts: i) presentation part, ii) demonstration part and iii) evaluation part. During the presentation part, Rezwan (M. R. Huq) provided an overview on data provenance and related terms. Then, he talked about different types of PCRaster maps (input files) processed by the scientific model. While some of these maps (e.g. map holding values of potential transpiration) are valid for a particular month in a given year, some other maps (e.g. map holding values of irrigation efficiency) are always valid irrespective of the year and the month. Rezwan also provided a brief description of PCRaster operations which were used in the model. All involved operations operate on a cell level and they have constant input-output ratio. Later, Rezwan presented a workflow of the model which had been developed manually by analyzing the source code, showing data dependencies between different operations and maps (data products).

Afterward, Rezwan gave a demonstration of the developed prototype that can infer fine-grained data provenance of a selected output data product by facilitating the given workflow of the model and available data products. The fine-grained data provenance graph shows contributing values from input maps and associated operations which produce the selected output data product. At this point, Rens (Dr. L. P. H. van Beek) asked a question about whether intermediate results were stored persistently or not. Rezwan replied that no intermediate results were stored but only input and output data products were stored persistently. Furthermore, Rens asked a question about whether the given workflow of the model could be saved so that the user could avoid entering it again. Andreas (Dr. A. Wombacher) replied that at this moment, it was not possible but it could be a potential improvement of the prototype.

Eventually, the researchers were asked a few questions about the applications of a fine-grained data provenance graph. Both researchers from Utrecht University saw the usefulness of a fine-grained data provenance graph. Yoshi (Y. Wada) told that the fine-grained data provenance graph could be very useful to trace a missing value to it's source values. He also mentioned that currently he had to do this manually which means that he had to look for a contributing source value out of thousands of maps which had been very time consuming. With this tool, he could complete this task within a few seconds. Moreover, Rens pointed out that the work-

flow of the scientific model could be also useful to validate the model or to debug the scientific model itself (e.g. looking for code repetitions etc.). However, the current version of the workflow had been prepared manually and hence, it might not capture the complete semantics of the source code.

Based on this discussion, we pointed out a few possible improvements that should be done in future. First, capturing the workflow of the scientific model based on the given source code should be automated to avoid manual interpretation which requires a lot of time and effort to understand the program. Second, it would be nice to make the prototype more user-friendly.

Since the initial evaluation on preliminary results was promising, both parties agreed to have another round of evaluation once the suggested improvements had been done.

A.2.4 Final Evaluation

This meeting was held to have a final evaluation on the prototype, inferring fine-grained data provenance information. Table A.4 shows the date, time, place, duration and participants of this meeting.

Table A.4: Final Evaluation Meeting

Goal	Final evaluation on the developed tool
Date and Time	July 03, 2012 at 14:00 hours
Place	Utrecht University
Duration	Around 120 minutes
Participants	Dr. L. P. H. van Beek, UU Y. Wada, UU Dr. A. Wombacher, UT M. R. Huq, UT

Meeting Transcript

This meeting had begun with a presentation on the prototype that can extract the workflow provenance of the scientific model automatically and

then can facilitate the workflow provenance and available data products to infer fine-grained data provenance. Rezwan (M. R. Huq) presented the approach briefly extracting the workflow provenance automatically from the given source code (Python program). This approach requires the user to provide a few information on each method signature on their first occurrence during the execution of the prototype. This information includes whether a method reads/writes data from/into persistent storage. Afterward, the prototype can extract a workflow provenance graph by analyzing the source code of the program. Based on this workflow provenance and available data products, the prototype can also infer fine-grained data provenance graph.

Then, the developed prototype had been demonstrated. Both researchers asked a few questions about the notations used in the fine-grained data provenance graph. We replied to their questions by providing explanation of these notations.

At last, the researchers were asked a few questions to evaluate the useability of provenance graphs and the prototype. The first question was about the application of a workflow provenance graph to debug a scientific model. Researchers replied that since a workflow provenance graph showed the complete data-flow of the program, it could be useful for code-level debugging to some extent. Rens (L. P. H. van Beek) mentioned that the workflow provenance graph could be also useful to compare different versions of the same scientific model, expected to produce the same output. Later, researchers were asked a question about the application of a fine-grained data provenance graph. At this point, Yoshi (Y. Wada) added that the fine-grained data provenance graph could be facilitated for error tracking and thus, it is more useful than the corresponding workflow provenance graph. Researchers liked the developed prototype. However, the prototype cannot handle recursive operations (e.g. neighborhood operations in PCRaster) at it's current stage. This could be a potential extension of the prototype in the future.

In general, researchers from Utrecht University recognized the usefulness of provenance graphs for debugging purposes. They also appreciated the developed prototype that is capable of inferring provenance information at reduced storage costs. They recommended to improve the user interface and to add more functionalities such as saving a workflow, customizing the result graph etc. We took their recommendations with high importance and we will try to extend the prototype in future.

A.3 CASE STUDY II : EXPLICIT PROVENANCE COLLECTION METHOD

In Chapter 9, we have described the mechanism to infer fine-grained data provenance in the context of a logic program, developed in Answer Set Programming (ASP). We have also documented provenance information explicitly during execution of the program. This technique is referred to as the explicit provenance collection method. The documented explicit provenance serves as a ground truth to evaluate the accuracy of the fine-grained data provenance inference method. In this section, we explain the mechanism of collecting explicit provenance for the scientific model developed in ASP as discussed in Chapter 9.

Explicit provenance collection method extends the given logic program with additional logical rules to document fine-grained data provenance explicitly, encoded as predicates. The augmentation of the logic program with these additional logical rules can be done manually (as done here) or can be automated, by parsing and analyzing the logic program. For the purpose of investigating the usefulness of fine-grained provenance graphs for debugging ASP programs and the pros and cons of explicit and inferred provenance information, the way of augmenting the logic program is not relevant.

We refer to the set of extra logical rules as explicit *provenance rules*. In this section, we describe the construction of the provenance rules per class of ASP rules, illustrate it with the aid of an example from the scenario and provide the translation of the derived provenance predicate into a graphical representation based on the workflow provenance model, discussed in Section 3.1. In particular, the different types of rule as indicated in Table 9.1 (see Section 9.1) are clustered into the following classes: i) logical rule, ii) projection rule, iii) choice rule with constraints and iv) logical rule with function.

A.3.1 Logical Rule

In general, a rule R in a logic program has two parts: head and body. If the predicates in the body are satisfied then the predicate in the head of the rule can be inferred. The structure of a logical rule is given below, which extends the version in Table 9.1 (see Section 9.1) by explicating the arguments

of the involved predicates as a vector $\overrightarrow{arg}_{a_i} = (args_{1,a_i}, \ldots, args_{n_{a_i},a_i})$ respectively:

$$h(\overrightarrow{arg}_h) :- a_1(\overrightarrow{arg}_{a_1}), ..., a_M(\overrightarrow{arg}_{a_M}) \\ [, not\ a_{M+1}(\overrightarrow{arg}_{M+1}), ..., not\ a_N(\overrightarrow{arg}_{a_N})].$$

The predicate $h(\overrightarrow{arg}_h)$ in the aforesaid rule R constitutes the head of the rule, where as the body of the rule R is comprised of the predicates a_1, ... , a_N. Each predicate in both head and body of the rule may have several arguments represented by the vector $\overrightarrow{arg}_h$ and $\overrightarrow{arg}_{a_i}$ respectively. The number of arguments for each predicate may vary.

A.3.1.1 *Provenance Extension*

To document explicit provenance, it is necessary to encode all predicates and argument bindings of the logical rule R, deriving the predicate h, into a provenance predicate. This is done by adding an extra logical rule for rule R to the original logic program. The construction of the corresponding provenance rule is quite straightforward. The provenance rule is a copy of the logical rule R with a modified head predicate.

To formulate the head predicate of the provenance rule R_{prov}, the keyword *'Prov'* is added as a suffix to the name of the predicate in the head part of rule R, thus hProv. The predicate hProv has the following arguments:

1. arguments of the head predicate of the rule R: $\overrightarrow{arg}_h$
2. a user defined rule identifier: rule_id
3. for each positive predicate, a_i in the body of the rule R:
   a) name of the predicate: a_i
   b) arguments of the predicate a_i: $\overrightarrow{arg}_{a_i}$

Negated predicates are not encoded, since in a provenance graph enumerating negated predicates may explode the graph and the negation is intended as Negation as Failure (NaF), which means that they have not been observed. As an example, for the aforesaid rule R, we add the following rule, RProv, which captures the explicit provenance information.

$$\begin{aligned} &hProv(\overrightarrow{arg}_h, rule\_id, a_1, \overrightarrow{arg}_{a_1}, \ldots, a_M, \overrightarrow{arg}_{a_M}) \\ &\quad :- a_1(\overrightarrow{arg}_{a_1}), \ldots, a_M(\overrightarrow{arg}_{a_M}), \\ &\qquad [not\ a_{M+1}(\overrightarrow{arg}_{a_{M+1}}), \ldots, not\ a_N(\overrightarrow{arg}_{a_N})]. \end{aligned}$$

A.3.1.2 *Example*

As an example, line 1 and 6 in Listing A.1 show two rules which are excerpted from the given logic program, realizing the use case as discussed in Section 9.2. Both of these rules follow the aforesaid basic structure of a logical rule and therefore, an extra provenance rule per logical rule will be added to document provenance explicitly. Based on the provenance rule formulation discussed above, line 4 and 9 in Listing A.1 show the provenance rules.

Listing A.1: Logical rules with corresponding provenance rules

```
1 riskvalue(rss, high, LOCATION) :- rss(STATUSTYPE, LOCATION, SEVERITY
      , TIME), negative(STATUSTYPE), SEVERITY>1.
2
3 % explicit provenance
4 riskvalueProv(rss, high, LOCATION, p5, rss, STATUSTYPE, LOCATION,
      SEVERITY, TIME, negative, STATUSTYPE) :- rss(STATUSTYPE,
      LOCATION, SEVERITY, TIME), negative(STATUSTYPE), SEVERITY>1.
5
6 riskvalue(rss, low, LOCATION) :- roadsegments(LOCATION), not
      riskvalue(rss, high, LOCATION).
7
8 % explicit provenance
9 riskvalueProv(rss, low, LOCATION, p8, roadsegments, LOCATION, null,
      null, null, null, null) :- roadsegments(LOCATION), not riskvalue
      (rss, high, LOCATION).
```

The `riskvalueProv` predicate in line 4 and 9 have different arity. Since a predicate in a logical program must always have the same arity, the remaining arguments of the predicate in line 9 are filled with *null* values. While constructing the corresponding provenance graph, the *null* values are being ignored.

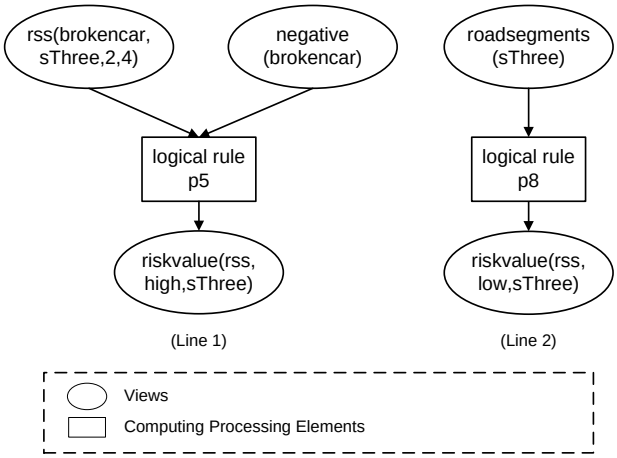


Figure A.2: Provenance graphs based on collected explicit provenance shown in Listing A.2

A.3.1.3 *Provenance Graph*

To construct the provenance graph, we interpret the provenance predicates as found in the answer set (output) of the logic program. Inverse to its construction the head and the positive body predicates can be derived knowing the arity of each predicate. The arity can either be derived from the remaining content of the answer set or be specified in a configuration file. The relation between input predicates and output predicate of the processing element is derived from the fact that the head predicate of the rule is defined by a single logical rule respectively. Therefore, a processing element represents a logical rule.

Listing A.2: Predicates containing provenance information which are derived based on provenance rules shown in Listing A.1

```
1 riskvalueProv(rss,high,sThree,p5,rss,brokencar,sThree,2,4,negative,
      brokencar).
2 riskvalueProv(rss,low,sThree,p8,roadsegments,sThree,null,null,null,
      null,null).
```

An example of provenance predicates as derived by the execution of the logic program is given in Listing A.2. Based on these explicit provenance predicates, corresponding provenance graphs are depicted in Figure A.2.

As described in Section 3.1, rectangles are used to represent computing processing elements (logical rules) and ellipses are used to represent

views (predicates). The computing processing elements, shown in Figure A.2, have a 'many to one' input-output ratio (see Section 3.1), indicating that all contributing input views/predicates must contain a tuple to actually derive the output view/predicate.

A.3.2 Projection Rule

A projection rule is a logical rule where the body of the rule contains a predicate with at least one argument, which is neither constrained by another predicate nor used in the head predicate. The schema of a projection rule is given below. Please note that for readability, we do not add the optional negated predicates in the body of the rule as in the previous section.

$$h(\overrightarrow{arg}_h) :- a_1(\overrightarrow{arg}_{a_1}), \ldots, a_M(\overrightarrow{arg}_{a_M}).$$
$$\text{where there exists an unbound } arg_{j,a_k} \in \overrightarrow{arg}_{a_k}$$
$$\text{with } a_k \in \{a_1, \ldots a_M\} \text{ and } 1 \leqslant j \leqslant n_{a_k}$$

The predicate h in the aforesaid projection rule constitutes the head of the rule, where as the body of the rule is comprised of the predicates a_1, ... , a_M. There must be at least one argument arg_{j,a_k} which is unbounded, i.e., not used by any other predicate in this rule. The determination of argument arg_{j,a_k} in a logical rule can be done by simply checking all variables for their occurrences in the head and the remaining body predicates. If there is no match for an argument, the condition for a projection rule is fulfilled.

A.3.2.1 *Provenance Extension*

Construction of the provenance rule follows the mechanism discussed in Section A.3.1.1. Please note that if there is a predicate h in the answer set for the head of the projection rule, there may be multiple instances of the provenance predicate $h\text{Prov}$ - one for each instance of argument arg_{j,a_k}.

$$h\text{Prov}(\overrightarrow{arg}_h, \text{rule\_id}, a_1, \overrightarrow{arg}_{a_1}, \ldots, a_M, \overrightarrow{arg}_{a_M})$$
$$:- a_1(\overrightarrow{arg}_{a_1}), \ldots, a_M(\overrightarrow{arg}_{a_M}).$$

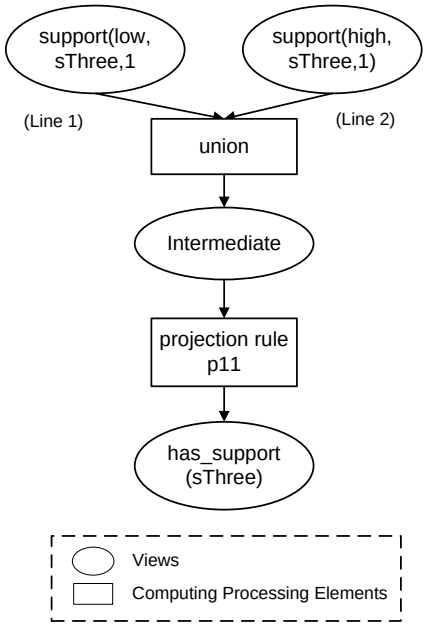


Figure A.3: Provenance graph based on collected explicit provenance shown in Listing A.4

A.3.2.2 *Example*

As an example, Listing A.3 line 1 shows an example of a projection rule excerpted from the given logic program. The variables `RISK` and `N` are unbounded in this example. Listing A.3, line 4 shows the explicit provenance rule related to the projection rule. Please be aware that the provenance rules for projection and logical rules do not deviate with regard to the construction of provenance rules, but they potentially deviate in the number of instances in the answer set (output) and therefore in the provenance graphs.

Listing A.3: A projection rule with it's corresponding provenance rule

```
1 has_support(LOC) :- support(RISK,LOC,N).
2
3 % explicit provenance
4 has_supportProv(LOC,p11,support,RISK,LOC,N) :- support(RISK,LOC,N).
```

A.3.2.3 *Provenance Graph*

Examples of the provenance predicates as derived by the execution of the logic program are shown in Listing A.4. The resulting provenance graph is depicted in Figure A.3.

Listing A.4: Predicates containing provenance information which are derived based on provenance rule shown in Listing A.3

```
1 has_supportProv(sThree,p11,support,low,sThree,1).
2 has_supportProv(sThree,p11,support,high,sThree,1).
```

The processing element, representing a projection rule, differs from the processing element, representing a logical rule (see Section A.3.1.3), by having a single input only. Therefore, the input-output ratio of the projection processing element is 'one to one', indicating that each tuple in the input view are consumed into a single tuple in the output view. The union processing element has an input-output ratio of 'one to one', thus a tuple in an input view is directly transformed into a tuple in the output view without correlating it with tuples from the remaining input views. As a consequence, the union in Figure A.3 introduces two tuples in the intermediate view, which are then projected into a single tuple in the `has_support` predicate.

A.3.3 Choice Rule with Constraints

A logic program might also include choice rules where a non-deterministic choice in the head of the rule generates the search space and associated constraints reduce the search space again. The structure of a choice rule is given in Table 9.1 (see Section 9.1) which is a short-hand notation of the following:

$$l\{h(\overrightarrow{arg}_h^1), \ldots, h(\overrightarrow{arg}_h^n)\}u :- a_1(\overrightarrow{arg}_{a_1}), \ldots a_M(\overrightarrow{arg}_{a_M}).$$

where $l, u \in \mathbb{N}$ and the choice rule says that the ASP solver must choose at least l number of predicates from the set of $h(\overrightarrow{arg}_h^1), \ldots, h(\overrightarrow{arg}_h^n)$ predicates but not more than u predicates. A choice rule is often associated with constraints. Constraints are used to minimize the search space by excluding certain combinations of predicates that do not satisfy a particular

condition. The basic structure of a constraint in relation with a choice rule is given below:

$$:- h(\overrightarrow{arg}^{i}_{h}),\ b_1(\overrightarrow{arg}_{b_1}), \ldots b_M(\overrightarrow{arg}_{b_M}), not\ b_{M+1}(\overrightarrow{arg}_{b_{M+1}}).$$

It is important to understand that the core information is the $h(\overrightarrow{arg}^{i}_{h})$ predicate and the negated predicate b_{M+1}. Since the body of a constraint must always evaluate to be false, it actually means that the negated predicate b_{M+1} must always evaluate to be true if the predicate $h(\overrightarrow{arg}^{i}_{h})$ evaluates to be true. This information must be added to the provenance graph.

A.3.3.1 *Provenance Extension*

To document explicit provenance for choice rules with constraints, we add one explicit provenance rule for each predicate in the head of the corresponding choice rule. The formulation of the provenance rule hProv follows the same procedure as discussed in Section A.3.1.3. However, the body of the provenance rule contains now an additional predicate $h(\overrightarrow{arg}^{i}_{h})$ to ensure that the right instances of the provenance predicate hProv are available in the answer set (output). To represent the constraints, an additional provenance rule is added hProvConstr which contains arguments including the arguments of the chosen head predicate $h(\overrightarrow{arg}^{i}_{h})$ as well as the name and the arguments of the negated predicate b_{M+1}. The body of this additional rule consists out of the head predicate of the choice rule $h(\overrightarrow{arg}^{i}_{h})$ and the negated predicate b_{M+1}. It is to be noted that there can be multiple instances of the hProvConstr provenance rules for a single head predicate $h(\overrightarrow{arg}^{i}_{h})$. The formulation of the provenance rule for the aforementioned choice rule with constraints is given below where $i \in [1, n]$:

$$\begin{aligned}
&hProv(\overrightarrow{arg}^{i}_{h}, rule_id, a_1, \overrightarrow{arg}_{a_1}, ..., a_M, \overrightarrow{arg}_{a_M})\\
&\quad :- h(\overrightarrow{arg}^{i}_{h}), a_1(\overrightarrow{arg}_{a_1}), \ldots, a_M(\overrightarrow{arg}_{a_M}).\\
&hProvConstr(\overrightarrow{arg}^{i}_{h}, rule_id, b_{M+1}, \overrightarrow{arg}_{b_{M+1}})\\
&\quad :- h(\overrightarrow{arg}^{i}_{h}), b_{M+1}(\overrightarrow{arg}_{b_{M+1}}).
\end{aligned}$$

A.3.3.2 *Example*

Listing A.5, line 1 and 2 show a choice rule with constraint from the use case. Please be aware that the enumeration of all choices is done by

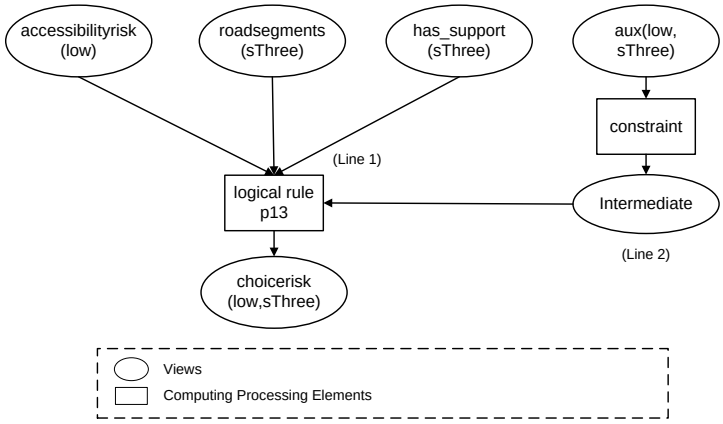


Figure A.4: Provenance graph based on collected explicit provenance shown in Listing A.6

providing the domain of variable `RISK` which is given by the predicate `accessibilityrisk(RISK)`. For the provenance rules in line 5 and 6 of Listing A.5, the enumeration of all possible arguments of `choicerisk` can also be avoided by using variables `RISK` and `LOC`. Please note that the answer set may contain multiple instances of the `choiceriskProvConstr` constraint - one for each constraint related to the choice rule.

Listing A.5: A choice rule with constraints and it's corresponding provenance rule

```
1 1{choicerisk(RISK,LOC): accessibilityrisk(RISK)}1:- roadsegments(LOC
      ), has_support(LOC).
2 :-choicerisk(RISK,LOC),not aux(RISK,LOC).
3
4 % explicit provenance
5 choiceriskProv(RISK, LOC, p13, accessibilityrisk, RISK, roadsegments
      , LOC, has_support, LOC) :- choicerisk(RISK,LOC), roadsegments(
      LOC), has_support(LOC).
6 choiceriskProvConstr(RISK, LOC, p13, aux, RISK, LOC) :- choicerisk(
      RISK,LOC), aux(RISK,LOC).
```

A.3.3.3 *Provenance Graph*

Examples of the provenance predicates as derived by the execution of the logic program are given in Listing A.6 and the resulting provenance graph is depicted in Figure A.4. The logical rule processing element has been ex-

plained in Section A.3.1.3. The constraint processing element is similar to the logical rule processing element having a 'many to one' input-output ratio and therefore they provide actually the same semantics. We use anyway different names in the provenance graph to increase the readability for the user, thus, ease the mapping back to the source code.

Listing A.6: Predicates containing provenance information which are derived based on provenance rules shown in Listing A.5

```
1 choiceriskProv(low,sThree,p13,accessibilityrisk,low,roadsegments,
      sThree,has_support,sThree).
2 choiceriskProvConstr(low,sThree,p13,aux,low,sThree).
```

A.3.4 Logical Rule with Function

In logic programming languages, there exist a few basic built-in support for operations on data products such as numerical functions `#count`, `#min`, `#max` etc. A logical rule facilitating a built-in function has the following form:

$$h(\overrightarrow{arg}_h,CMP) :- a_1(\overrightarrow{arg}_{a_1}),\ldots,a_M(\overrightarrow{arg}_{a_M}),$$
$$CMP = \#function\{a_{M+1}(\overrightarrow{arg}_{a_{M+1}}),\ldots,a_N(\overrightarrow{arg}_{a_N})\}.$$

The difference between a basic logical rule, (see Section A.3.1) and a logical rule with a function is that in a logical rule with function there is one extra argument in the head of the rule, i.e., CMP, which value is calculated using $\#function$ over a set of predicates $a_{M+1},\ldots,a_N$.

A.3.4.1 *Provenance Extension*

A logical rule with function is executed in two steps: first, $\#function$ is applied over the set of predicates $a_{M+1},\ldots a_N$ and second, the entire logical rule is executed. Since in the first step a set of predicates is used, a representation of the set as arguments in the provenance predicate results in an arbitrary number of arguments. Therefore, the provenance rule is split into two pieces. The first one addresses the entire logical rule, which is formulated following the same procedure discussed in Section A.3.1.1. However, the head of this provenance rule, $hProv$, does not contain the predicates

used by the function, i.e. $a_{M+1}, \ldots, a_N$, as it's arguments. The structure of the first provenance rule is given below:

$$\begin{aligned} &hProv(\overrightarrow{arg}_h, CMP, rule\_id, a_1, \overrightarrow{arg}_{a_1}, \ldots, a_M, \overrightarrow{arg}_{a_M}) \\ &\quad :- a_1(\overrightarrow{arg}_{a_1}), \ldots, a_M(\overrightarrow{arg}_{a_M}), \\ &\qquad CMP = \#function\{a_{M+1}(\overrightarrow{arg}_{a_{M+1}})\}. \end{aligned}$$

The second provenance rule addresses the set of predicates used to evaluate the function, where one predicate is created for each element of the set to capture explicit provenance information. The head predicate $hProv\#function$ of this provenance rule is named after the original rule appended with the key word '*Prov*' and as the suffix the name of the function, $\#function$. Adding the name of the function in the predicate encodes the type of the function used and therefore gives an indication on how to translate this information into a provenance graph later on. The head predicate $hProv\#function$ of this rule also contains the predicates $a_{M+1}, \ldots, a_N$ used as input to the function. The body of the rule is formulated following the procedure discussed in Section A.3.1.1. Furthermore, it contains one extra predicate which is the head of the original rule ensuring that the exact relationship is maintained between predicates once the rule is inferred. Finally, the body of the rule contains the predicates used inside the function, but does not apply the function to these predicates. The structure of the rule is given below:

$$\begin{aligned} &hProv\#function(\overrightarrow{arg}_h, CMP, rule\_id, a_1, \overrightarrow{arg}_{a_1}, \ldots, \\ &\qquad a_M, \overrightarrow{arg}_{a_M}, a_{M+1}, \overrightarrow{arg}_{a_{M+1}}, \ldots, a_N, \overrightarrow{arg}_{a_N}) \\ &\quad :- h(\overrightarrow{arg}_h, CMP), a_1(\overrightarrow{arg}_{a_1}), \ldots \\ &\qquad a_M(\overrightarrow{arg}_{a_M}), a_{M+1}(\overrightarrow{arg}_{a_{M+1}}), \ldots, a_N(\overrightarrow{arg}_{a_N}). \end{aligned}$$

A.3.4.2 *Example*

Listing A.7, line 1 shows a logical rule with the numerical function `#count` taken from the original logic program. Based on the procedure to formulate provenance rules discussed above, Listing A.7, lines 4 and 5 show the extended program containing the provenance rules. Please note that the answer set may contain multiple instances of the `supportProvCount` predicate - one for each predicate used for evaluating the function `#count`.

Listing A.7: A logical rule with #count function and it's corresponding provenance rules

```
support(RISK, LOCATION, N) :- accessibilityrisk(RISK), roadsegments(
    LOCATION), N=#count{riskvalue(STREAMTYPE, RISK, LOCATION):
    streamtype(STREAMTYPE)}, N>0.

% explicit provenance
supportProv(RISK, LOCATION, N, p10, accessibilityrisk, RISK,
    roadsegments, LOCATION) :- accessibilityrisk(RISK), roadsegments
    (LOCATION), N=#count{riskvalue(STREAMTYPE, RISK, LOCATION):
    streamtype(STREAMTYPE)}, N>0.
supportProvCount(RISK, LOCATION, N, p10, accessibilityrisk, RISK,
    roadsegments, LOCATION, riskvalue, STREAMTYPE, RISK, LOCATION,
    streamtype, STREAMTYPE) :- support(RISK, LOCATION, N),
    accessibilityrisk(RISK), roadsegments(LOCATION), riskvalue(
    STREAMTYPE, RISK, LOCATION), streamtype(STREAMTYPE).
```

A.3.4.3 *Provenance Graph*

Examples of the provenance predicates as derived by the execution of the logic program are given in Lisiting A.8. These provenance predicates are represented as a provenance graph, depicted in Figure A.5.

Listing A.8: Predicates containing provenance information which are derived based on provenance rules shown in Listing A.7

```
supportProv(low,sThree,1,p10,accessibilityrisk,low,roadsegments,
    sThree).
supportProvCount(low,sThree,1,p10,riskvalue,rss,low,sThree).
```

The logical rule processing element has been explained in Section A.3.1.3. The graph consists of using all related predicates for the evaluation of the function as input for the count processing element, which counts the number of tuples accessible in input views and outputs this number in the output view, i.e., the intermediate view. The processing element uses a 'many to one' input-output ratio. The *Intermediate1* view in addition with other predicates in the body of the logical rule are used as input for the logical rule processing element resulting in the *Intermediate2* output view. The *Intermediate2* view contains now implicitly the counted value, but it is not explicated in the name of a view. To explicate this information, the con-

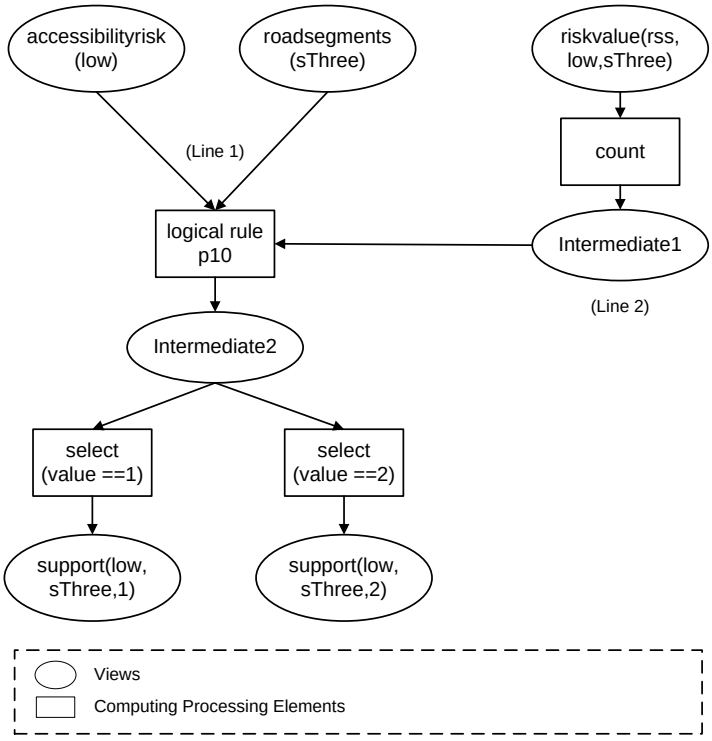


Figure A.5: Provenance graphs based on collected explicit provenance shown in Listing A.8

tent of the view must be analyzed by the *select* processing element. This is a processing element which actually considers the value of the tuples in a view to see whether it satisfies the given condition or not. Therefore we characterize the *select* processing element as a variable ratio processing element, while all others have been classified as constant ratio processing elements. While the later one allows easy inference, a variable ratio processing element requires a special inference, thus is less generic. The output view of the *select* processing element explicates the count result in the name of the view, thus corresponds to the head predicate of the logical rule with a function.

BIBLIOGRAPHY

[1] W. M. P. van der Aalst and K. van Hee. *Workflow Management - Models, Methods, and Systems*. MIT Press, 2002. (Cited on page 4.)

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003. (Cited on pages 5, 25, and 89.)

[3] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research*, pages 277–289, 2005. (Cited on pages 5, 25, 27, and 89.)

[4] 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 International Conference on Very Large Data Bases*, pages 1151–1154, 2006. (Cited on page 22.)

[5] M. K. Anand, S. Bowers, T. McPhillips, and B. Ludäscher. Efficient provenance storage over nested data collections. In *Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology*, pages 958–969. ACM, 2009. (Cited on page 20.)

[6] E. Angelino, D. Yamins, and M. Seltzer. Starflow: A script-centric data analysis environment. In *Proceedings of the International Provenance and Annotation Workshop*, pages 236–250, 2010. (Cited on pages 7, 31, and 41.)

[7] 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. Technical Report 2004-20, Stanford InfoLab, 2004. (Cited on page 25.)

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the ACM*

SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 1–16. ACM, 2002. (Cited on pages 5, 25, 37, and 89.)

[9] R. Baeza and B. Ribeiro. *Modern information retrieval*. Addison-Wesley Reading, 1999. (Cited on page 207.)

[10] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003. (Cited on page 241.)

[11] R. S. Barga and L. A. Digiampietri. Automatic generation of workflow provenance. In *Provenance and annotation of data*, volume 4145 of *LNCS*, pages 1–9. Springer, 2006. (Cited on pages 18, 19, 20, and 36.)

[12] R. S. Barga and L. A. Digiampietri. Automatic capture and efficient storage of e-science experiment provenance. *Concurrency and Computation: Practice and Experience*, 20(5):419–429, 2008. (Cited on pages 18 and 19.)

[13] D. Barseghian, I. Altintas, M. B. Jones, D. Crawl, N. Potter, J. Gallagher, P. Cornillon, M. Schildhauer, E. T. Borer, and E. W. Seabloom. Workflows and extensions to the Kepler scientific workflow system to support environmental sensor data access and analysis. *Ecological Informatics*, 5(1):42–50, 2010. (Cited on page 18.)

[14] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *Proceedings of the International Conference on Very Large Data Bases*, pages 953–964, 2006. (Cited on page 22.)

[15] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *VLDB Journal*, 14(4):373–396, 2005. (Cited on pages 21, 22, 23, 24, 37, and 264.)

[16] C. M. Bishop. *Patter Recognition and Machine Learning*. Springer Science+Business Media LLC, 2006. (Cited on pages 145, 175, and 194.)

[17] R. Bose and J. Frew. Lineage retrieval for scientific data processing: A survey. *ACM Computing Surveys (CSUR)*, 37(1):1–28, 2005. (Cited on page 15.)

[18] S. Bowers, T. M. McPhillips, and B. Ludäscher. Provenance in collection-oriented scientific workflows. *Concurrency and Computation: Practice and Experience*, 20(5):519–529, 2008. (Cited on page 20.)

[19] P. Buneman and W. C. Tan. Provenance in databases. In *Proceedings of the ACM SIGMOD international conference on Management of Data*, pages 1171–1173. ACM, 2007. (Cited on pages 2, 8, and 15.)

[20] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *Proceedings of the International Conference on Database Theory*, pages 316–330, 2001. (Cited on pages 2 and 21.)

[21] P. Buneman, S. Khanna, and W. C. Tan. On propagation of deletions and annotations through views. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 150–158. ACM, 2002. (Cited on pages 8, 21, 22, 23, 24, 37, and 264.)

[22] P. Buneman, J. Cheney, and E. V. Kostylev. Hierarchical models of provenance. In *Proceedings of the USENIX conference on Theory and Practice of Provenance*, pages 10–13, 2012. (Cited on page 19.)

[23] P. Buneman, E. V. Kostylev, and S. Vansummeren. Annotations are relative. In *Proceedings of the International Conference on Database Theory*, pages 177–188. ACM, 2013. (Cited on pages 23 and 30.)

[24] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. VisTrails: Visualization meets data management. In *Proceedings of the ACM SIGMOD international conference on Management of Data*, pages 745–747. ACM, 2006. (Cited on pages 5, 17, 19, 20, 36, 89, 215, and 262.)

[25] 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 ACM SIGMOD international conference on Management of Data*, page 668. ACM, 2003. (Cited on pages 24 and 25.)

[26] A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *Proceedings of the ACM SIGMOD international conference on Management of Data*, pages 993–1006. ACM, 2008. (Cited on pages 111 and 112.)

[27] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009. (Cited on page 15.)

[28] L. Chiticariu, W. C. Tan, and G. Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In *Proceedings of the ACM SIGMOD international conference on Management of Data*, pages 942–944. ACM, 2005. (Cited on pages 21, 22, 23, 24, 37, and 264.)

[29] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *VLDB Journal*, 12(1):41–58, 2003. (Cited on pages 21, 22, and 23.)

[30] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2):179–227, 2000. (Cited on pages 21, 22, and 23.)

[31] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989. (Cited on page 55.)

[32] S. Davidson, , S. C. Boulakia, A. Eyal, B. Ludäscher, T. M. McPhillips, S. Bowers, M. K. Anand, and J. Freire. Provenance in scientific workflow systems. *IEEE Data Engineering Bulletin*, 30(4):44–50, 2007. (Cited on pages 15 and 17.)

[33] J. de Vlieg, R. van Schaik, P. Aerts, S. Lusher, and F. Sienstra. Data-Stewardship in the Big Data Era: Taking Care of Data. Technical report, Netherlands eSciece Center, 2013. (Cited on page 16.)

[34] E. Deelman, G. Singh, M. H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005. (Cited on page 19.)

[35] E. Della Valle, S. Ceri, D. Barbieri, D. Braga, and A. Campi. A first step towards stream reasoning. In *Future Internet - FIS 2008*, volume 5468 of *LNCS*, pages 72–81. Springer Berlin / Heidelberg, 2009. (Cited on page 241.)

[36] E. Della Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a Streaming World! Reasoning upon Rapidly Changing Information. *IEEE Intelligent Systems*, 24(6):83–89, 2009. (Cited on page 241.)

[37] T. Do, S. Loke, and F. Liu. Answer set programming for stream reasoning. *Advances in Artificial Intelligence*, pages 104–109, 2011. (Cited on page 241.)

[38] J. Dozier and J. Frew. Computational provenance in hydrologic science: a snow mapping example. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1890): 1021–1033, 2009. (Cited on page 30.)

[39] E. Erdem, Y. Erdem, H. Erdogan, and U. Öztok. Finding answers and generating explanations for complex biomedical queries. In *AAAI Conference on Artificial Intelligence*, pages 785–790, 2011. (Cited on page 240.)

[40] K. S. Esmaili. *Data stream processing in complex applications*. PhD thesis, Eidgenössische Technische Hochschule (ETH), Zürich, 2011. (Cited on pages 26 and 27.)

[41] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987. (Cited on page 31.)

[42] J. Freire, C. Silva, S. P. Callahan, E. Santos, C. E. Scheidegger, and H. T. Vo. Managing rapidly-evolving scientific workflows. In *Provenance and Annotation of Data*, volume 4145 of *LNCS*, pages 10–18. Springer, 2006. (Cited on page 19.)

[43] J. Frew and R. Bose. Earth system science workbench: A data management infrastructure for earth science products. In *Proceedings of International Conference on Scientific and Statistical Database Management*, pages 180–189. IEEE, 2001. (Cited on page 29.)

[44] J. Frew and P. Slaughter. Es3: A demonstration of transparent provenance for scientific computation. *Provenance and Annotation of Data and Processes*, pages 200–207, 2008. (Cited on pages 30 and 90.)

[45] J. Frew, D. Metzger, and P. Slaughter. Automatic capture and reconstruction of computational provenance. *Concurrency and Computation: Practice and Experience*, 20(5):485–496, 2008. (Cited on pages 29 and 90.)

[46] J. Futrelle. Harvesting RDF triples. In *Provenance and Annotation of Data*, volume 4145 of *LNCS*, pages 64–72. Springer, 2006. (Cited on pages 23, 24, 37, and 122.)

[47] F. Gebali. *Analysis of Computer and Communication Networks*. Springer Science+Business Media LLC, 2008. (Cited on pages 148 and 196.)

[48] M. Gebser, O. Sabuncu, and T. Schaub. An incremental answer set programming based system for finite model computation. *AI Communications*, 24(2):195–212, 2011. (Cited on page 241.)

[49] M. Gebser, T. Schaub, S. Thiele, and P. Veber. Detecting inconsistencies in large biological networks with answer set programming. *Theory and Practice of Logic Programming*, 11(2-3):323–360, 2011. (Cited on page 240.)

[50] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub. Stream reasoning with answer set programming: Preliminary report. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 1–5, 2012. (Cited on pages 240 and 241.)

[51] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub. Stream reasoning with answer set programming: Extended version. Technical report, University of Potsdam, 2012. (Cited on page 241.)

[52] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Answerset solving in practice. Technical report, University of Postdam, 2012. (Cited on pages 85, 240, and 241.)

[53] B. Gedik, H. Andrade, K. L. Wu, P. S. Yu, and M. Doo. SPADE: The System S declarative stream processing engine. In *Proceedings of the ACM SIGMOD international conference on Management of Data*, pages 1123–1134. ACM, 2008. (Cited on page 25.)

[54] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the International Conference on Logic programming*, volume 161, pages 1–11, 1988. (Cited on page 241.)

[55] 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 National Conference on Artificial Intelligence*, pages 1767–1774, 2007. (Cited on page 19.)

[56] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 174–185. IEEE, 2009. (Cited on pages 21, 23, 24, 37, and 264.)

[57] B. Glavic, K. S. Esmaili, P. M. Fischer, and N. Tatbul. The case for fine-grained stream provenance. In *Proceedings of the BTW Workshop on Data Streams and Event Processing*, pages 58–61, 2011. (Cited on page 26.)

[58] B. Glavic, K. S. Esmaili, P. M. Fischer, and N. Tatbul. Ariadne: Managing Fine-Grained Provenance on Data Streams. Technical Report 771, Systems Lab, ETH, Zurich, 2012. (Cited on pages 8, 26, 27, 28, 37, 38, and 264.)

[59] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003. (Cited on pages 24 and 37.)

[60] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *Proceedings of the international conference on Very Large Data Bases*, pages 675–686, 2007. (Cited on pages 21, 23, 24, 37, and 264.)

[61] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40. ACM, 2007. (Cited on pages 21, 23, and 30.)

[62] P. Groth. *The Origin of Data: Enabling the Determination of Provenance in Multi-institutional Scientific Systems through the Documentation of Processes*. PhD thesis, University of Southampton, October 2007. (Cited on page 19.)

[63] P. Groth and L. Moreau. Recording process documentation for provenance. *IEEE Transactions on Parallel and Distributed Systems*, 20(9): 1246–1259, 2009. (Cited on pages 19 and 36.)

[64] P. Groth, S. Miles, W. Fang, S. C. Wong, K. P. Zauner, and L. Moreau. Recording and using provenance in a protein compressibility experiment. In *Proceedings of the International Symposium on High Performance Distributed Computing*, pages 201–208. IEEE, 2005. (Cited on pages 8 and 90.)

[65] P. Groth, S. Miles, and L. Moreau. PReServ: Provenance recording for services. In *Processings of the UK e-Science All Hands Meeting*. EPSRC, 2005. (Cited on page 19.)

[66] P. Groth, Y. Gil, and S. Magliacane. Automatic Metadata Annotation through Reconstructing Provenance. In *Semantic Web in Provenance Management*, volume 856. CEUR Workshop Proceedings, 2012. (Cited on pages 30, 32, and 36.)

[67] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the International Conference on Software Engineering*, pages 392–411. ACM, 1992. (Cited on pages 31 and 55.)

[68] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: A tool for building and running workflows of services. *Nucleic acids research*, 34:729–732, 2006. (Cited on pages 18, 20, and 36.)

[69] M. R. Huq, A. Wombacher, and P. M. G. Apers. Facilitating fine grained data provenance using temporal data model. In *Proceedings of the Workshop on Data Management for Sensor Networks*, pages 8–13. ACM, 2010. (Cited on pages 8, 37, 38, and 90.)

[70] M. R. Huq, A. Wombacher, and P. M. G. Apers. Inferring Fine-Grained Data Provenance in Stream Data Processing: Reduced Storage Cost, High Accuracy. In *Database and Expert Systems Applications*, volume 6861 of *LNCS*, pages 118–127. Springer, 2011. (Cited on pages 6 and 277.)

[71] M. R. Huq, P. M. G. Apers, and A. Wombacher. Probabilistic Inference of Fine-Grained Data Provenance. In *Database and Expert Systems Applications*, volume 7446 of *LNCS*, pages 296–310. Springer, 2012. (Cited on page 277.)

[72] M. R. Huq, P. M. G. Apers, and A. Wombacher. Fine-Grained Provenance Inference for a Large Processing Chain with Non-materialized Intermediate Views. In *Scientific and Statistical Database Management*, volume 7338 of *LNCS*, pages 397–405. Springer, 2012. (Cited on page 277.)

[73] M. R. Huq, P. M. G. Apers, and A. Wombacher. ProvenanceCurious: A tool to infer data provenance from scripts. In *Proceedings of the International Conference on Extending Database Technology*, pages 765–768. ACM, 2013. (Cited on pages 267 and 270.)

[74] R. Ikeda, J. Cho, C. Fang, S. Salihoglu, S. Torikai, and J. Widom. Provenance-based debugging and drill-down in data-oriented workflows. In *IEEE International Conference on Data Engineering*, pages 1249–1252. IEEE, 2012. (Cited on pages 16, 35, 233, and 239.)

[75] Y. Jararweh, A. Hary, Y. B. Al-Nashif, S. Hariri, A. Akoglu, and D. Jenerette. Accelerated discovery through integration of Kepler with data turbine for ecosystem research. In *IEEE/ACS International Conference on Computer Systems and Applications*, pages 1005–1012, 2009. (Cited on page 18.)

[76] G. Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974. (Cited on pages 24 and 48.)

[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*, 20(5):587–597, 2008. (Cited on pages 5, 19, 20, 36, and 262.)

[78] L. A. Klein. *Sensor and data fusion: a tool for information assessment and decision making*, volume 138. Society of Photo Optical, 2004. (Cited on page 1.)

[79] C. Koncilia. A Bi-Temporal Data Warehouse Model. In *Proceedings of the International Conference on Advanced Information Systems Engineering*, pages 77–80, 2003. (Cited on pages 92, 94, 98, 127, 128, 171, and 172.)

[80] D. P. Lanter. Design of a lineage-based meta-data base for GIS. *Cartography and Geographic Information Science*, 18(4):255–261, 1991. (Cited on pages 2 and 29.)

[81] J. Ledlie, C. Ng, D. A. Holland, K. K. Muniswamy-Reddy, U. Braun, and M. Seltzer. Provenance-aware sensor data storage. In *Proceeding of the Workshop on Networking Meets Databases*, 2005. (Cited on page 26.)

[82] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995. (Cited on pages 24 and 48.)

[83] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1):39–54, 2002. (Cited on page 241.)

[84] 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. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006. (Cited on pages 5, 17, 20, 36, 89, 215, and 262.)

[85] B. Ludäscher, M. Weske, T. McPhillips, and S. Bowers. Scientific workflows: Business as usual? *Business Process Management*, pages 31–47, 2009. (Cited on page 17.)

[86] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. (Cited on page 6.)

[87] 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*, pages 248–263. Springer, 2006. (Cited on page 20.)

[88] A. Mileo, D. Merico, and R. Bisiani. Reasoning support for risk prediction and prevention in independent living. *Theory and Practice of Logic Programming*, 11(2-3):361–395, 2011. (Cited on page 240.)

[89] A. Mileo, T. Schaub, D. Merico, and R. Bisiani. Knowledge-based multi-criteria optimization to support indoor positioning. *Annals of Mathematics and Artificial Intelligence*, 62(3-4):345–370, 2011. (Cited on page 240.)

[90] S. Miles. Automatically adapting source code to document provenance. In *Provenance and Annotation of Data and Processes*, volume 6378 of *LNCS*, pages 102–110. Springer, 2010. (Cited on pages 30, 32, and 36.)

[91] S. Miles, P. Groth, M. Branco, and L. Moreau. The requirements of using provenance in e-science experiments. *Journal of Grid Computing*, 5(1):1–25, 2007. (Cited on page 34.)

[92] S. Miles, P. Groth, S. Munroe, and L. Moreau. PrIMe: A methodology for developing provenance-aware applications. *ACM Transactions on Software Engineering and Methodology*, 20(3):1–42, 2011. (Cited on pages 5 and 31.)

[93] 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*, volume 5272 of *LNCS*, pages 253–265. Springer, 2008. (Cited on pages 26 and 27.)

[94] L. Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2–3):99–241, 2010. (Cited on page 15.)

[95] L. Moreau and P. Missier. The PROV data model and abstract syntax notation. Working draft, 2013. Available at `http://www.w3.org/TR/prov-dm/`. (Cited on pages 33 and 271.)

[96] 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*, volume 5272 of *LNCS*, pages 323–326. Springer, 2008. (Cited on page 32.)

[97] L. Moreau, P. Groth, S. Miles, J. Vazquez-Salceda, J. Ibbotson, S. Jiang, S. Munroe, O. Rana, A. Schreiber, V. Tan, and L. Varga. The provenance of electronic data. *Communications of the ACM*, 51(4):52–58, 2008. (Cited on page 33.)

[98] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, et al. The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743–756, 2011. (Cited on pages 32 and 33.)

[99] K. K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 43–56, 2006. (Cited on page 30.)

[100] H. B. Newman, M. H. Ellisman, and J. A. Orcutt. Data-intensive e-science frontier research. *Communication of the ACM*, 46(11):68–77, 2003. (Cited on page 1.)

[101] V. Novelli, M. De Vos, J. A. Padget, and D. D'Ayala. LOG-IDEAH: ASP for Architectonic Asset Preservation. In *Proceedings of the International Conference on Logic Programming*, pages 393–403, 2012. (Cited on page 240.)

[102] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Greenwood, T. Carver, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20 (17):3045–3054, June 2004. (Cited on pages 5, 17, 18, 20, 36, 89, 215, and 262.)

[103] U. Park and J. Heidemann. Provenance in sensornet republishing. In *Provenance and Annotation of Data and Processes*, volume 5272 of *LNCS*, pages 280–292. Springer, 2008. (Cited on pages 26, 28, 37, 38, 111, 122, and 264.)

[104] F. Portmann, S. Siebert, C. Bauer, and P. Döll. Mirca2000 - global monthly irrigated and rainfed crop areas around the year 2000: a new high-resolution data set for agricultural and hydrological modelling. *Global Biogeochemical Cycles*, 24(1), 2010. (Cited on page 225.)

[105] G. Pothier, É. Tanter, and J. Piquer. Scalable omniscient debugging. In *ACM SIGPLAN Notices*, volume 42, pages 535–552. ACM, 2007. (Cited on page 34.)

[106] S. Reddy, G. Chen, B. Fulkerson, S. J. Kim, U. Park, N. Yau, J. Cho, M. Hansen, and J. Heidemann. 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*, pages 11–16, 2007. (Cited on page 27.)

[107] J. Rohwer, D. Gerten, and W. Lucht. Development of functional types of irrigation for improved global crop modelling. *PIK Re-*

port 104, Potsdam Institute for Climate Impact Research, 2007. (Cited on page 225.)

[108] W. Sansrimahachai. *Tracing Fine-Grained Provenance in Stream Processing Systems using A Reverse Mapping Method*. PhD thesis, University of Southampton, April 2012. (Cited on pages 8, 26, 28, 29, 37, 38, 111, and 264.)

[109] A. D. Sarma, M. Theobald, and J. Widom. LIVE: A Lineage-Supported Versioned DBMS. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, pages 416–433, 2010. (Cited on pages 8, 22, 23, 24, 37, 38, 111, and 264.)

[110] P. Schneider, T. Vogt, M. Schirmer, J. Doetsch, N. Linde, N. Pasquale, P. Perona, and O. A. Cirpka. Towards improved instrumentation for assessing river-groundwaterinteractions in a restored river corridor. *Hydrology and Earth System Sciences*, 15(8):2531–2549, 2011. (Cited on page 91.)

[111] M. Shields. Control- versus data-driven workflows. In *Workflows for e-Science*, pages 167–173. Springer London, 2007. (Cited on pages 3, 6, 47, and 49.)

[112] S. Siebert and P. Döll. Quantifying blue and green virtual water contents in global crop production as well as potential production losses without irrigation. *Journal of Hydrology*, 384:198–217, 2010. (Cited on page 225.)

[113] C. A. Silles and A. R. Runnalls. Provenance-awareness in R. In *Provenance and Annotation of Data and Processes*, volume 6378 of *LNCS*, pages 64–72. Springer, 2010. (Cited on pages 30, 32, and 36.)

[114] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005. (Cited on pages 2, 15, 16, 34, and 239.)

[115] Y. L. Simmhan, B. Plale, D. Gannon, and S. Marru. Performance evaluation of the karma provenance framework for scientific workflows. In *Provenance and Annotation of Data*, volume 4145 of *LNCS*, pages 222–236. Springer, 2006. (Cited on pages 18, 20, and 36.)

[116] Y. L. Simmhan, B. Plale, and D. Gannon. Karma2: Provenance management for data driven workflows. *International Journal of Web Services Research*, 5:1–23, 2008. (Cited on pages 5, 18, 89, 103, 122, 215, and 262.)

[117] H. Stuckenschmidt, S. Ceri, E. Della Valle, F. Van Harmelen, and P. di Milano. Towards expressive stream reasoning. In *Proceedings of the Dagstuhl Seminar on Semantic Aspects of Sensor Networks*, 2010. (Cited on page 241.)

[118] M. Szomszor and L. Moreau. Recording and reasoning over data provenance in web and grid services. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *LNCS*, pages 603–620, 2003. (Cited on page 30.)

[119] W. C. Tan. Provenance in databases: Past, current, and future. *IEEE Data Engineering Bulletin*, 30(4):3–12, 2007. (Cited on page 15.)

[120] L. P. H. van Beek, Y. Wada, and M. F. P. Bierkens. Global monthly water stress: I. water balance and water availability. *Water Resources Research*, 47(W07517), 2011. (Cited on page 225.)

[121] B. F. van Dongen and W. M. P. van der Aalst. A meta model for process mining data. *EMOI-INTEROP*, 160, 2005. (Cited on page 30.)

[122] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst. The prom framework: A new era in process mining tool support. In *Applications and Theory of Petri Nets 2005*, volume 3536 of *LNCS*, pages 444–454. Springer, 2005. (Cited on page 30.)

[123] N. N. Vijayakumar. *Data managment in distributed stream processing systems*. PhD thesis, Indiana University, 2007. (Cited on page 26.)

[124] N. N. Vijayakumar and B. Plale. Towards low overhead provenance tracking in near real-time stream filtering. In *Provenance and Annotation of Data*, volume 4145 of *LNCS*, pages 46–54. Springer, 2006. (Cited on page 26.)

[125] N. N. Vijayakumar and B. Plale. Tracking stream provenance in complex event processing systems for workflow-driven computing. In

Proceedings of the International Workshop on Event-driven Architecture, Processing and Systems, pages 1–8, 2007. (Cited on page 26.)

[126] Y. Wada, L. P. H. van Beek, and M. F. P. Bierkens. Modelling global water stress of the recent past: on the relative importance of trends in water demand and climate variability. *Hydrology and Earth System Sciences Discussion*, 8:7399–7460, 2011. (Cited on page 227.)

[127] Y. Wada, L. P. H. van Beek, D. Viviroli, H. H. Dürr, R. Weingartner, and M. F. P. Bierkens. Global monthly water stress: II. Water demand and severity of water. *Water Resources Research*, 47(W07518), 2011. (Cited on pages 13, 224, 226, 267, and 276.)

[128] Y. Wada, L. P. H. van Beek, and M. F. P. Bierkens. Nonsustainable groundwater sustaining irrigation: A global assessment. *Water Resources Research*, 48(W00L06), 2012. (Cited on page 227.)

[129] 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 ACM SIGMOBILE international workshop on Systems and networking support for healthcare and assisted living environments*, pages 95–100. ACM, 2007. (Cited on pages 26, 27, 28, 37, 38, and 264.)

[130] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, (4):352–357, 1984. (Cited on pages 31 and 272.)

[131] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. Technical report, Stanford InfoLab, Stanford University, 2004. (Cited on pages 8, 22, 23, 24, 37, and 264.)

[132] A. Wombacher. Data workflow - a workflow model for continuous data processing. Technical Report TR-CTIT-10-12, Centre for Telematics and Information Technology, University of Twente, 2010. (Cited on page 103.)

[133] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proceedings of the International Conference on Data Engineering*, pages 91–102, 1997. (Cited on pages 21 and 23.)

[134] H. Yin, C. Bockisch, and M. Akşit. A fine-grained, customizable debugger for aspect-oriented programming. In *Transactions on Aspect-Oriented Software Development X*, volume 7800 of *LNCS*, pages 1–38. Springer, 2013. (Cited on page 34.)

[135] P. Yue and L. He. Geospatial data provenance in cyberinfrastructure. In *Proceedings of International Conference on Geoinformatics*, pages 1–4. IEEE, 2009. (Cited on pages 1 and 2.)

[136] P. Yue, J. Gong, and L. Di. Augmenting geospatial data provenance through metadata tracking in geospatial service chaining. *Computers & Geosciences*, 36(3):270–281, 2010. (Cited on page 29.)

[137] P. Yue, Z. Sun, J. Gong, L. Di, and X. Lu. A provenance framework for web geoprocessing workflows. In *Proceedings of the IEEE International Geoscience and Remote Sensing Symposium*, pages 3811–3814. IEEE, 2011. (Cited on page 29.)

[138] P. Yue, Y. Wei, L. Di, L. He, J. Gong, and L. Zhang. Sharing geospatial provenance in a service-oriented environment. *Computers, Environment and Urban Systems*, 35(4):333–343, 2011. (Cited on pages 29 and 32.)

PUBLICATIONS BY THE AUTHOR

REFEREED PUBLICATIONS

i. M. R. Huq, P. M. G. Apers, and A. Wombacher. An Inference-based Framework to Manage Data Provenance in Geoscience Applications. Accepted in *IEEE Transactions on Geoscience and Remote Sensing*, Early access article DOI: 10.1109/TGRS.2013.2247769, IEEE Geoscience and Remote Sensing Society, 2013. (Impact Factor: 2.895)

ii. M. R. Huq, P. M. G. Apers, and A. Wombacher. ProvenanceCurious: A tool to infer data provenance from scripts. In *Proceedings of the International Conference on Extending Database Technology (EDBT'13)*, pages 765–768. ACM, 2013.

iii. A. Wombacher and M. R. Huq. Towards Automatic Capturing of Semi-structured Process Provenance. In *Data-Driven Process Discovery and Analysis*, volume 162 of *LNBIP*, pages 84–99, Springer, 2013.

iv. M. R. Huq, P. M. G. Apers, and A. Wombacher. Probabilistic Inference of Fine-Grained Data Provenance. In *Database and Expert Systems Applications (DEXA'12)*, volume 7446 of *LNCS*, pages 296–310, Springer, 2012.

v. M. R. Huq, P. M. G. Apers, and A. Wombacher. From scripts towards provenance inference. In *Proceedings of the IEEE International Conference on E-Science (e-Science'12)*, pages 118–127, IEEE Computer Society, 2012.

vi. M. R. Huq, P. M. G. Apers, and A. Wombacher. Fine-Grained Provenance Inference for a Large Processing Chain with Non-materialized Intermediate Views. In *Scientific and Statistical Database Management (SSDBM'12)*, volume 7338 of *LNCS*, pages 397–405, Springer, 2012.

vii. M. R. Huq, A. Wombacher, and P. M. G. Apers. Inferring Fine-Grained Data Provenance in Stream Data Processing: Reduced Storage Cost,

High Accuracy. In *Database and Expert Systems Applications (DEXA'11)*, volume 6861 of *LNCS*, pages 118–127. Springer, 2011.

viii. M. R. Huq, A. Wombacher, and P. M. G. Apers. Adaptive Inference of Fine-grained Data Provenance to Achieve High Accuracy at Lower Storage Costs. In *Proceedings of the IEEE International Conference on E-Science (e-Science'11)*, pages 202–209, IEEE Computer Society, 2011.

ix. M. R. Huq, A. Wombacher, and P. M. G. Apers. Facilitating fine grained data provenance using temporal data model. In *Proceedings of the Workshop on Data Management for Sensor Networks (DMSN'10)*, pages 8–13. ACM, 2010.

x. M. R. Huq, A. Wombacher, and P. M. G. Apers. Identifying the challenges for optimizing the process to achieve reproducible results in e-science applications. In *Proceedings of the Ph.D. Workshop on Information and Knowledge Management (PIKM'10)*, pages 75–78. ACM, 2010.

NON-REFEREED PUBLICATIONS

x. M. R. Huq, A. Wombacher, A. Mileo. Data Provenance Inference in Logic Programming: Reducing Effort of Instance-driven Debugging. Technical Report TR-CTIT-13-11, Centre for Telematics and Information Technology, University of Twente, 2013.

xi. A. Wombacher, M. R. Huq. Towards Automatic Capturing of Manual Data Processing Provenance. Technical Report TR-CTIT-11-12, Centre for Telematics and Information Technology, University of Twente, 2011.

SIKS DISSERTATIONS SERIES

1998-01 Johan van den Akker (CWI) *DEGAS - An Active, Temporal Database of Autonomous Objects*

1998-02 Floris Wiesman (UM) *Information Retrieval by Graphically Browsing Meta-Information*

1998-03 Ans Steuten (TUD) *A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective*

1998-04 Dennis Breuker (UM) *Memory versus Search in Games*

1998-05 E. W. Oskamp (RUL) *Computerondersteuning bij Straftoemeting*

1999-01 Mark Sloof (VU) *Physiology of Quality Modeling; Automated modeling of Quality Change of Agricultural Products*

1999-02 Rob Potharst (EUR) *Classification using decision trees and neural nets*

1999-03 Don Beal (UM) *The Nature of Minimax Search*

1999-04 Jacques Penders (UM) *The practical Art of Moving Physical Objects*

1999-05 Aldo de Moor (KUB) *Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems*

1999-06 Niek J. E. Wijngaards (VU) *Re-design of compositional systems*

1999-07 David Spelt (UT) *Verification support for object database design*

1999-08 Jacques H. J. Lenting (UM) *Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation*

2000-01 Frank Niessink (VU) *Perspectives on Improving Software Maintenance*

2000-02 Koen Holtman (TUE) *Prototyping of CMS Storage Management*

2000-03 Carolien M. T. Metselaar (UVA) *Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief*

2000-04 Geert de Haan (VU) *ETAG, A Formal Model of Competence Knowledge for User Interface Design*

2000-05 Ruud van der Pol (UM) *Knowledge-based Query Formulation in Information Retrieval*

2000-06 Rogier van Eijk (UU) *Programming Languages for Agent Communication*

2000-07 Niels Peek (UU) *Decision-theoretic Planning of Clinical Patient Management*

2000-08 Veerle Coup (EUR) *Sensitivity Analysis of Decision-Theoretic Networks*

2000-09 Florian Waas (CWI) *Principles of Probabilistic Query Optimization*

2000-10 Niels Nes (CWI) *Image Database Management System Design Considerations, Algorithms and Architecture*

2000-11 Jonas Karlsson (CWI) *Scalable Distributed Data Structures for Database Management*

2001-01 Silja Renooij (UU) *Qualitative Approaches to Quantifying Probabilistic Networks*

2001-02 Koen Hindriks (UU) *Agent Programming Languages: Programming with Mental Models*

2001-03 Maarten van Someren (UvA) *Learning as problem solving*

2001-04 Evgueni Smirnov (UM) *Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets*

2001-05 Jacco van Ossenbruggen (VU) *Processing Structured Hypermedia: A Matter of Style*

2001-06 Martijn van Welie (VU) *Task-based User Interface Design*

2001-07 Bastiaan Schonhage (VU) *Diva: Architectural Perspectives on Information Visualization*

2001-08 Pascal van Eck (VU) *A Compositional Semantic Structure for Multi-Agent Systems Dynamics*

2001-09 Pieter Jan 't Hoen (RUL) *Distributed Development of Large Object-Oriented Models, Views of Packages as Classes*

2001-10 Maarten Sierhuis (UvA) *Modeling and Simulating Work Practice; BRAHMS: a multiagent modeling and simulation language for work practice analysis and design*

2001-11 Tom M. van Engers (VUA) *Knowledge Management: The Role of Mental Models in Business Systems Design*

2002-01 Nico Lassing (VU) *Architecture-Level Modifiability Analysis*

2002-02 Roelof van Zwol (UT) *Modeling and searching web-based document collections*

2002-03 Henk Ernst Blok (UT) *Database Optimization Aspects for Information Retrieval*

2002-04 Juan Roberto Castelo Valdueza (UU) *The Discrete Acyclic Digraph Markov Model in Data Mining*

2002-05 Radu Serban (VU) *The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents*

2002-06 Laurens Mommers (UL) *Applied legal epistemology; Building a knowledge-based ontology of the legal domain*

2002-07 Peter Boncz (CWI) *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*

2002-08 Jaap Gordijn (VU) *Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas*

2002-09 Willem-Jan van den Heuvel (KUB) *Integrating Modern Business Applications with Objectified Legacy Systems*

2002-10 Brian Sheppard (UM) *Towards Perfect Play of Scrabble*

2002-11 Wouter C.A. Wijngaards (VU) *Agent Based Modelling of Dynamics: Biological and Organisational Applications*

2002-12 Albrecht Schmidt (Uva) *Processing XML in Database Systems*

2002-13 Hongjing Wu (TUE) *A Reference Architecture for Adaptive Hypermedia Applications*

2002-14 Wieke de Vries (UU) *Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems*

2002-15 Rik Eshuis (UT) *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*

2002-16 Pieter van Langen (VU) *The Anatomy of Design: Foundations, Models and Applications*

2002-17 Stefan Manegold (UVA) *Understanding, Modeling, and Improving Main-Memory Database Performance*

2003-01 Heiner Stuckenschmidt (VU) *Ontology-Based Information Sharing in Weakly Structured Environments*

2003-02 Jan Broersen (VU) *Modal Action Logics for Reasoning About Reactive Systems*

2003-03 Martijn Schuemie (TUD) *Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy*

2003-04 Milan Petkovic (UT) *Content-Based Video Retrieval Supported by Database Technology*

2003-05 Jos Lehmann (UVA) *Causation in Artificial Intelligence and Law - A modeling approach*

2003-06 Boris van Schooten (UT) *Development and specification of virtual environments*

2003-07 Machiel Jansen (UvA) *Formal Explorations of Knowledge Intensive Tasks*

2003-08 Yongping Ran (UM) *Repair Based Scheduling*

2003-09 Rens Kortmann (UM) *The resolution of visually guided behavior*

2003-10 Andreas Lincke (UvT) *Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture*

2003-11 Simon Keizer (UT) *Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks*

2003-12 Roeland Ordelman (UT) *Dutch speech recognition in multimedia information retrieval*

2003-13 Jeroen Donkers (UM) *Nosce Hostem - Searching with Opponent Models*

2003-14 Stijn Hoppenbrouwers (KUN) *Freezing Language: Conceptualisation Processes across ICT-Supported Organisations*

2003-15 Mathijs de Weerdt (TUD) *Plan Merging in Multi-Agent Systems*

2003-16 Menzo Windhouwer (CWI) *Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses*

2003-17 David Jansen (UT) *Extensions of Statecharts with Probability, Time, and Stochastic Timing*

2003-18 Levente Kocsis (UM) *Learning Search Decisions*

2004-01 Virginia Dignum (UU) *A Model for Organizational Interaction: Based on Agents, Founded in Logic*

2004-02 Lai Xu (UvT) *Monitoring Multi-party Contracts for E-business*

2004-03 Perry Groot (VU) *A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving*

2004-04 Chris van Aart (UVA) *Organizational Principles for Multi-Agent Architectures*

2004-05 Viara Popova (EUR) *Knowledge discovery and monotonicity*

2004-06 Bart-Jan Hommes (TUD) *The Evaluation of Business Process Modeling Techniques*

2004-07 Elise Boltjes (UM) *Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes*

2004-08 Joop Verbeek (UM) *Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise*

2004-09 Martin Caminada (VU) *For the Sake of the Argument; explorations into argument-based reasoning*

2004-10 Suzanne Kabel (UVA) *Knowledge-rich indexing of learning-objects*

2004-11 Michel Klein (VU) *Change Management for Distributed Ontologies*

2004-12 The Duy Bui (UT) *Creating emotions and facial expressions for embodied agents*

2004-13 Wojciech Jamroga (UT) *Using Multiple Models of Reality: On Agents who Know how to Play*

2004-14 Paul Harrenstein (UU) *Logic in Conflict. Logical Explorations in Strategic Equilibrium*

2004-15 Arno Knobbe (UU) *Multi-Relational Data Mining*

2004-16 Federico Divina (VU) *Hybrid Genetic Relational Search for Inductive Learning*

2004-17 Mark Winands (UM) *Informed Search in Complex Games*

2004-18 Vania Bessa Machado (UvA) *Supporting the Construction of Qualitative Knowledge Models*

2004-19 Thijs Westerveld (UT) *Using generative probabilistic models for multimedia retrieval*

2004-20 Madelon Evers (Nyenrode) *Learning from Design: facilitating multidisciplinary design teams*

2005-01 Floor Verdenius (UVA) *Methodological Aspects of Designing Induction-Based Applications*

2005-02 Erik van der Werf (UM)) *AI techniques for the game of Go*

2005-03 Franc Grootjen (RUN) *A Pragmatic Approach to the Conceptualisation of Language*

2005-04 Nirvana Meratnia (UT) *Towards Database Support for Moving Object data*

2005-05 Gabriel Infante-Lopez (UVA) *Two-Level Probabilistic Grammars for Natural Language Parsing*

2005-06 Pieter Spronck (UM) *Adaptive Game AI*

2005-07 Flavius Frasincar (TUE) *Hypermedia Presentation Generation for Semantic Web Information Systems*

2005-08 Richard Vdovjak (TUE) *A Model-driven Approach for Building Distributed Ontology-based Web Applications*

2005-09 Jeen Broekstra (VU) *Storage, Querying and Inferencing for Semantic Web Languages*

2005-10 Anders Bouwer (UVA) *Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments*

2005-11 Elth Ogston (VU) *Agent Based Matchmaking and Clustering - A Decentralized Approach to Search*

2005-12 Csaba Boer (EUR) *Distributed Simulation in Industry*

2005-13 Fred Hamburg (UL) *Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen*

2005-14 Borys Omelayenko (VU) *Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics*

2005-15 Tibor Bosse (VU) *Analysis of the Dynamics of Cognitive Processes*

2005-16 Joris Graaumans (UU) *Usability of XML Query Languages*

2005-17 Boris Shishkov (TUD) *Software Specification Based on Re-usable Business Components*

2005-18 Danielle Sent (UU) *Test-selection strategies for probabilistic networks*

2005-19 Michel van Dartel (UM) *Situated Representation*

2005-20 Cristina Coteanu (UL) *Cyber Consumer Law, State of the Art and Perspectives*

2005-21 Wijnand Derks (UT) *Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics*

2006-01 Samuil Angelov (TUE) *Foundations of B2B Electronic Contracting*

2006-02 Cristina Chisalita (VU) *Contextual issues in the design and use of information technology in organizations*

2006-03 Noor Christoph (UVA) *The role of metacognitive skills in learning to solve problems*

2006-04 Marta Sabou (VU) *Building Web Service Ontologies*

2006-05 Cees Pierik (UU) *Validation Techniques for Object-Oriented Proof Outlines*

2006-06 Ziv Baida (VU) *Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling*

2006-07 Marko Smiljanic (UT) *XML schema matching – balancing efficiency and effectiveness by means of clustering*

2006-08 Eelco Herder (UT) *Forward, Back and Home Again - Analyzing User Behavior on the Web*

2006-09 Mohamed Wahdan (UM) *Automatic Formulation of the Auditor's Opinion*

2006-10 Ronny Siebes (VU) *Semantic Routing in Peer-to-Peer Systems*

2006-11 Joeri van Ruth (UT) *Flattening Queries over Nested Data Types*

2006-12 Bert Bongers (VU) *Interactivation - Towards an e-cology of people, our technological environment, and the arts*

2006-13 Henk-Jan Lebbink (UU) *Dialogue and Decision Games for Information Exchanging Agents*

2006-14 Johan Hoorn (VU) *Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change*

2006-15 Rainer Malik (UU) *CONAN: Text Mining in the Biomedical Domain*

2006-16 Carsten Riggelsen (UU) *Approximation Methods for Efficient Learning of Bayesian Networks*

2006-17 Stacey Nagata (UU) *User Assistance for Multitasking with Interruptions on a Mobile Device*

2006-18 Valentin Zhizhkun (UVA) *Graph transformation for Natural Language Processing*

2006-19 Birna van Riemsdijk (UU) *Cognitive Agent Programming: A Semantic Approach*

2006-20 Marina Velikova (UvT) *Monotone models for prediction in data mining*

2006-21 Bas van Gils (RUN) *Aptness on the Web*

2006-22 Paul de Vrieze (RUN) *Fundaments of Adaptive Personalisation*

2006-23 Ion Juvina (UU) *Development of Cognitive Model for Navigating on the Web*

2006-24 Laura Hollink (VU) *Semantic Annotation for Retrieval of Visual Resources*

2006-25 Madalina Drugan (UU) *Conditional log-likelihood MDL and Evolutionary MCMC*

2006-26 Vojkan Mihajlovic (UT) *Score Region Algebra: A Flexible Framework for Structured Information Retrieval*

2006-27 Stefano Bocconi (CWI) *Vox Populi: generating video documentaries from semantically annotated media repositories*

2006-28 Borkur Sigurbjornsson (UVA) *Focused Information Access using XML Element Retrieval*

2007-01 Kees Leune (UvT) *Access Control and Service-Oriented Architectures*

2007-02 Wouter Teepe (RUG) *Reconciling Information Exchange and Confidentiality: A Formal Approach*

2007-03 Peter Mika (VU) *Social Networks and the Semantic Web*

2007-04 Jurriaan van Diggelen (UU) *Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach*

2007-05 Bart Schermer (UL) *Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance*

2007-06 Gilad Mishne (UVA) *Applied Text Analytics for Blogs*

2007-07 Natasa Jovanovic' (UT) *To Whom It May Concern - Addressee Identification in Face-to-Face Meetings*

2007-08 Mark Hoogendoorn (VU) *Modeling of Change in Multi-Agent Organizations*

2007-09 David Mobach (VU) *Agent-Based Mediated Service Negotiation*

2007-10 Huib Aldewereld (UU) *Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols*

2007-11 Natalia Stash (TUE) *Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System*

2007-12 Marcel van Gerven (RUN) *Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty*

2007-13 Rutger Rienks (UT) *Meetings in Smart Environments; Implications of Progressing Technology*

2007-14 Niek Bergboer (UM) *Context-Based Image Analysis*

2007-15 Joyca Lacroix (UM) *NIM: a Situated Computational Memory Model*

2007-16 Davide Grossi (UU) *Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems*

2007-17 Theodore Charitos (UU) *Reasoning with Dynamic Networks in Practice*

2007-18 Bart Orriens (UvT) *On the development an management of adaptive business collaborations*

2007-19 David Levy (UM) *Intimate relationships with artificial partners*

2007-20 Slinger Jansen (UU) *Customer Configuration Updating in a Software Supply Network*

2007-21 Karianne Vermaas (UU) *Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005*

2007-22 Zlatko Zlatev (UT) *Goal-oriented design of value and process models from patterns*

2007-23 Peter Barna (TUE) *Specification of Application Logic in Web Information Systems*

2007-24 Georgina RamÃrez Camps (CWI) *Structural Features in XML Retrieval*

2007-25 Joost Schalken (VU) *Empirical Investigations in Software Process Improvement*

2008-01 Katalin Boer-SorbÃ¡n (EUR) *Agent-Based Simulation of Financial Markets: A modular, continuous-time approach*

2008-02 Alexei Sharpanskykh (VU) *On Computer-Aided Methods for Modeling and Analysis of Organizations*

2008-03 Vera Hollink (UVA) *Optimizing hierarchical menus: a usage-based approach*

2008-04 Ander de Keijzer (UT) *Management of Uncertain Data - towards unattended integration*

2008-05 Bela Mutschler (UT) *Modeling and simulating causal dependencies on process-aware information systems from a cost perspective*

2008-06 Arjen Hommersom (RUN) *On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective*

2008-07 Peter van Rosmalen (OU) *Supporting the tutor in the design and support of adaptive e-learning*

2008-08 Janneke Bolt (UU) *Bayesian Networks: Aspects of Approximate Inference*

2008-09 Christof van Nimwegen (UU) *The paradox of the guided user: assistance can be counter-effective*

2008-10 Wauter Bosma (UT) *Discourse oriented summarization*

2008-11 Vera Kartseva (VU) *Designing Controls for Network Organizations: A Value-Based Approach*

2008-12 Jozsef Farkas (RUN) *A Semiotically Oriented Cognitive Model of Knowledge Representation*

2008-13 Caterina Carraciolo (UVA) *Topic Driven Access to Scientific Handbooks*

2008-14 Arthur van Bunningen (UT) *Context-Aware Querying; Better Answers with Less Effort*

2008-15 Martijn van Otterlo (UT) *The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.*

2008-16 Henriette van Vugt (VU) *Embodied agents from a user's perspective*

2008-17 Martin Op 't Land (TUD) *Applying Architecture and Ontology to the Splitting and Allying of Enterprises*

2008-18 Guido de Croon (UM) *Adaptive Active Vision*

2008-19 Henning Rode (UT) *From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search*

2008-20 Rex Arendsen (UVA) *Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven*

2008-21 Krisztian Balog (UVA) *People Search in the Enterprise*

2008-22 Henk Koning (UU) *Communication of IT-Architecture*

2008-23 Stefan Visscher (UU) *Bayesian network models for the management of ventilator-associated pneumonia*

2008-24 Zharko Aleksovski (VU) *Using background knowledge in ontology matching*

2008-25 Geert Jonker (UU) *Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency*

2008-26 Marijn Huijbregts (UT) *Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled*

2008-27 Hubert Vogten (OU) *Design and Implementation Strategies for IMS Learning Design*

2008-28 Ildiko Flesch (RUN) *On the Use of Independence Relations in Bayesian Networks*

2008-29 Dennis Reidsma (UT) *Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans*

2008-30 Wouter van Atteveldt (VU) *Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content*

2008-31 Loes Braun (UM) *Pro-Active Medical Information Retrieval*

2008-32 Trung H. Bui (UT) *Toward Affective Dialogue Management using Partially Observable Markov Decision Processes*

2008-33 Frank Terpstra (UVA) *Scientific Workflow Design; theoretical and practical issues*

2008-34 Jeroen de Knijf (UU) *Studies in Frequent Tree Mining*

2008-35 Ben Torben Nielsen (UvT) *Dendritic morphologies: function shapes structure*

2009-01 Rasa Jurgelenaite (RUN) *Symmetric Causal Independence Models*

2009-02 Willem Robert van Hage (VU) *Evaluating Ontology-Alignment Techniques*

2009-03 Hans Stol (UvT) *A Framework for Evidence-based Policy Making Using IT*

2009-04 Josephine Nabukenya (RUN) *Improving the Quality of Organisational Policy Making using Collaboration Engineering*

2009-05 Sietse Overbeek (RUN) *Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality*

2009-06 Muhammad Subianto (UU) *Understanding Classification*

2009-07 Ronald Poppe (UT) *Discriminative Vision-Based Recovery and Recognition of Human Motion*

2009-08 Volker Nannen (VU) *Evolutionary Agent-Based Policy Analysis in Dynamic Environments*

2009-09 Benjamin Kanagwa (RUN) *Design, Discovery and Construction of Service-oriented Systems*

2009-10 Jan Wielemaker (UVA) *Logic programming for knowledge-intensive interactive applications*

2009-11 Alexander Boer (UVA) *Legal Theory, Sources of Law & the Semantic Web*

2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin) *Operating Guidelines for Services*

2009-13 Steven de Jong (UM) *Fairness in Multi-Agent Systems*

2009-14 Maksym Korotkiy (VU) *From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)*

2009-15 Rinke Hoekstra (UVA) *Ontology Representation - Design Patterns and Ontologies that Make Sense*

2009-16 Fritz Reul (UvT) *New Architectures in Computer Chess*

2009-17 Laurens van der Maaten (UvT) *Feature Extraction from Visual Data*

2009-18 Fabian Groffen (CWI) *Armada, An Evolving Database System*

2009-19 Valentin Robu (CWI) *Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets*

2009-20 Bob van der Vecht (UU) *Adjustable Autonomy: Controling Influences on Decision Making*

2009-21 Stijn Vanderlooy (UM) *Ranking and Reliable Classification*

2009-22 Pavel Serdyukov (UT) *Search For Expertise: Going beyond direct evidence*

2009-23 Peter Hofgesang (VU) *Modelling Web Usage in a Changing Environment*

2009-24 Annerieke Heuvelink (VUA) *Cognitive Models for Training Simulations*

2009-25 Alex van Ballegooij (CWI) *"RAM: Array Database Management through Relational Mapping"*

2009-26 Fernando Koch (UU) *An Agent-Based Model for the Development of Intelligent Mobile Services*

2009-27 Christian Glahn (OU) *Contextual Support of social Engagement and Reflection on the Web*

2009-28 Sander Evers (UT) *Sensor Data Management with Probabilistic Models*

2009-29 Stanislav Pokraev (UT) *Model-Driven Semantic Integration of Service-Oriented Applications*

2009-30 Marcin Zukowski (CWI) *Balancing vectorized query execution with bandwidth-optimized storage*

2009-31 Sofiya Katrenko (UVA) *A Closer Look at Learning Relations from Text*

2009-32 Rik Farenhorst (VU) and Remco de Boer (VU) *Architectural Knowledge Management: Supporting Architects and Auditors*

2009-33 Khiet Truong (UT) *How Does Real Affect Affect Affect Recognition In Speech?*

2009-34 Inge van de Weerd (UU) *Advancing in Software Product Management: An Incremental Method Engineering Approach*

2009-35 Wouter Koelewijn (UL) *Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling*

2009-36 Marco Kalz (OUN) *Placement Support for Learners in Learning Networks*

2009-37 Hendrik Drachsler (OUN) *Navigation Support for Learners in Informal Learning Networks*

2009-38 Riina Vuorikari (OU) *Tags and self-organisation: a metadata ecology for learning resources in a multilingual context*

2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin) *Service Substitution – A Behavioral Approach Based on Petri Nets*

2009-40 Stephan Raaijmakers (UvT) *Multinomial Language Learning: Investigations into the Geometry of Language*

2009-41 Igor Berezhnyy (UvT) *Digital Analysis of Paintings*

2009-42 Toine Bogers (UvT) *Recommender Systems for Social Bookmarking*

2009-43 Virginia Nunes Leal Franqueira (UT) *Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients*

2009-44 Roberto Santana Tapia (UT) *Assessing Business-IT Alignment in Networked Organizations*

2009-45 Jilles Vreeken (UU) *Making Pattern Mining Useful*

2009-46 Loredana Afanasiev (UvA) *Querying XML: Benchmarks and Recursion*

2010-01 Matthijs van Leeuwen (UU) *Patterns that Matter*

2010-02 Ingo Wassink (UT) *Work flows in Life Science*

2010-03 Joost Geurts (CWI) *A Document Engineering Model and Processing Framework for Multimedia documents*

2010-04 Olga Kulyk (UT) *Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments*

2010-05 Claudia Hauff (UT) *Predicting the Effectiveness of Queries and Retrieval Systems*

2010-06 Sander Bakkes (UvT) *Rapid Adaptation of Video Game AI*

2010-07 Wim Fikkert (UT) *Gesture interaction at a Distance*

2010-08 Krzysztof Siewicz (UL) *Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments*

2010-09 Hugo Kielman (UL) *A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging*

2010-10 Rebecca Ong (UL) *Mobile Communication and Protection of Children*

2010-11 Adriaan Ter Mors (TUD) *The world according to MARP: Multi-Agent Route Planning*

2010-12 Susan van den Braak (UU) *Sensemaking software for crime analysis*

2010-13 Gianluigi Folino (RUN) *High Performance Data Mining using Bio-inspired techniques*

2010-14 Sander van Splunter (VU) *Automated Web Service Reconfiguration*

2010-15 Lianne Bodenstaff (UT) *Managing Dependency Relations in Inter-Organizational Models*

2010-16 Sicco Verwer (TUD) *Efficient Identification of Timed Automata, theory and practice*

2010-17 Spyros Kotoulas (VU) *Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications*

2010-18 Charlotte Gerritsen (VU) *Caught in the Act: Investigating Crime by Agent-Based Simulation*

2010-19 Henriette Cramer (UvA) *People's Responses to Autonomous and Adaptive Systems*

2010-20 Ivo Swartjes (UT) *Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative*

2010-21 Harold van Heerde (UT) *Privacy-aware data management by means of data degradation*

2010-22 Michiel Hildebrand (CWI) *End-user Support for Access to Heterogeneous Linked Data*

2010-23 Bas Steunebrink (UU) *The Logical Structure of Emotions*

2010-24 Dmytro Tykhonov *Designing Generic and Efficient Negotiation Strategies*

2010-25 Zulfiqar Ali Memon (VU) *Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective*

2010-26 Ying Zhang (CWI) *XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines*

2010-27 Marten Voulon (UL) *Automatisch contracteren*

2010-28 Arne Koopman (UU) *Characteristic Relational Patterns*

2010-29 Stratos Idreos (CWI) *Database Cracking: Towards Auto-tuning Database Kernels*

2010-30 Marieke van Erp (UvT) *Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval*

2010-31 Victor de Boer (UVA) *Ontology Enrichment from Heterogeneous Sources on the Web*

2010-32 Marcel Hiel (UvT) *An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems*

2010-33 Robin Aly (UT) *Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval*

2010-34 Teduh Dirgahayu (UT) *Interaction Design in Service Compositions*

2010-35 Dolf Trieschnigg (UT) *Proof of Concept: Concept-based Biomedical Information Retrieval*

2010-36 Jose Janssen (OU) *Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification*

2010-37 Niels Lohmann (TUE) *Correctness of services and their composition*

2010-38 Dirk Fahland (TUE) *From Scenarios to components*

2010-39 Ghazanfar Farooq Siddiqui (VU) *Integrative modeling of emotions in virtual agents*

2010-40 Mark van Assem (VU) *Converting and Integrating Vocabularies for the Semantic Web*

2010-41 Guillaume Chaslot (UM) *Monte-Carlo Tree Search*

2010-42 Sybren de Kinderen (VU) *Needs-driven service bundling in a multi-supplier setting - the computational e3-service approach*

2010-43 Peter van Kranenburg (UU) *A Computational Approach to Content-Based Retrieval of Folk Song Melodies*

2010-44 Pieter Bellekens (TUE) *An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain*

2010-45 Vasilios Andrikopoulos (UvT) *A theory and model for the evolution of software services*

2010-46 Vincent Pijpers (VU) *e3alignment: Exploring Inter-Organizational Business-ICT Alignment*

2010-47 Chen Li (UT) *Mining Process Model Variants: Challenges, Techniques, Examples*

2010-48 Withdrawn

2010-49 Jahn-Takeshi Saito (UM) *Solving difficult game positions*

2010-50 Bouke Huurnink (UVA) *Search in Audiovisual Broadcast Archives*

2010-51 Alia Khairia Amin (CWI) *Understanding and supporting information seeking tasks in multiple sources*

2010-52 Peter-Paul van Maanen (VU) *Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention*

2010-53 Edgar Meij (UVA) *Combining Concepts and Language Models for Information Access*

2011-01 Botond Cseke (RUN) *Variational Algorithms for Bayesian Inference in Latent Gaussian Models*

2011-02 Nick Tinnemeier (UU) *Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language*

2011-03 Jan Martijn van der Werf (TUE) *Compositional Design and Verification of Component-Based Information Systems*

2011-04 Hado van Hasselt (UU) *Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference learning algorithms*

2011-05 Base van der Raadt (VU) *Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.*

2011-06 Yiwen Wang (TUE) *Semantically-Enhanced Recommendations in Cultural Heritage*

2011-07 Yujia Cao (UT) *Multimodal Information Presentation for High Load Human Computer Interaction*

2011-08 Nieske Vergunst (UU) *BDI-based Generation of Robust Task-Oriented Dialogues*

2011-09 Tim de Jong (OU) *Contextualised Mobile Media for Learning*

2011-10 Bart Bogaert (UvT) *Cloud Content Contention*

2011-11 Dhaval Vyas (UT) *Designing for Awareness: An Experience-focused HCI Perspective*

2011-12 Carmen Bratosin (TUE) *Grid Architecture for Distributed Process Mining*

2011-13 Xiaoyu Mao (UvT) *Airport under Control. Multiagent Scheduling for Airport Ground Handling*

2011-14 Milan Lovric (EUR) *Behavioral Finance and Agent-Based Artificial Markets*

2011-15 Marijn Koolen (UvA) *The Meaning of Structure: the Value of Link Evidence for Information Retrieval*

2011-16 Maarten Schadd (UM) *Selective Search in Games of Different Complexity*

2011-17 Jiyin He (UVA) *Exploring Topic Structure: Coherence, Diversity and Relatedness*

2011-18 Mark Ponsen (UM) *Strategic Decision-Making in complex games*

2011-19 Ellen Rusman (OU) *The Mind ' s Eye on Personal Profiles*

2011-20 Qing Gu (VU) *Guiding service-oriented software engineering - A view-based approach*

2011-21 Linda Terlouw (TUD) *Modularization and Specification of Service-Oriented Systems*

2011-22 Junte Zhang (UVA) *System Evaluation of Archival Description and Access*

2011-23 Wouter Weerkamp (UVA) *Finding People and their Utterances in Social Media*

2011-24 Herwin van Welbergen (UT) *Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior*

2011-25 Syed Waqar ul Qounain Jaffry (VU)) *Analysis and Validation of Models for Trust Dynamics*

2011-26 Matthijs Aart Pontier (VU) *Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots*

2011-27 Aniel Bhulai (VU) *Dynamic website optimization through autonomous management of design patterns*

2011-28 Rianne Kaptein (UVA) *Effective Focused Retrieval by Exploiting Query Context and Document Structure*

2011-29 Faisal Kamiran (TUE) *Discrimination-aware Classification*

2011-30 Egon van den Broek (UT) *Affective Signal Processing (ASP): Unraveling the mystery of emotions*

2011-31 Ludo Waltman (EUR) *Computational and Game-Theoretic Approaches for Modeling Bounded Rationality*

2011-32 Nees-Jan van Eck (EUR) *Methodological Advances in Bibliometric Mapping of Science*

2011-33 Tom van der Weide (UU) *Arguing to Motivate Decisions*

2011-34 Paolo Turrini (UU) *Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations*

2011-35 Maaike Harbers (UU) *Explaining Agent Behavior in Virtual Training*

2011-36 Erik van der Spek (UU) *Experiments in serious game design: a cognitive approach*

2011-37 Adriana Burlutiu (RUN) *Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference*

2011-38 Nyree Lemmens (UM) *Bee-inspired Distributed Optimization*

2011-39 Joost Westra (UU) *Organizing Adaptation using Agents in Serious Games*

2011-40 Viktor Clerc (VU) *Architectural Knowledge Management in Global Software Development*

2011-41 Luan Ibraimi (UT) *Cryptographically Enforced Distributed Data Access Control*

2011-42 Michal Sindlar (UU) *Explaining Behavior through Mental State Attribution*

2011-43 Henk van der Schuur (UU) *Process Improvement through Software Operation Knowledge*

2011-44 Boris Reuderink (UT) *Robust Brain-Computer Interfaces*

2011-45 Herman Stehouwer (UvT) *Statistical Language Models for Alternative Sequence Selection*

2011-46 Beibei Hu (TUD) *Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work*

2011-47 Azizi Bin Ab Aziz (VU) *Exploring Computational Models for Intelligent Support of Persons with Depression*

2011-48 Mark Ter Maat (UT) *Response Selection and Turn-taking for a Sensitive Artificial Listening Agent*

2011-49 Andreea Niculescu (UT) *Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality*

2012-01 Terry Kakeeto (UvT) *Relationship Marketing for SMEs in Uganda*

2012-02 Muhammad Umair (VU) *Adaptivity, emotion, and Rationality in Human and Ambient Agent Models*

2012-03 Adam Vanya (VU) *Supporting Architecture Evolution by Mining Software Repositories*

2012-04 Jurriaan Souer (UU) *Development of Content Management System-based Web Applications*

2012-05 Marijn Plomp (UU) *Maturing Interorganisational Information Systems*

2012-06 Wolfgang Reinhardt (OU) *Awareness Support for Knowledge Workers in Research Networks*

2012-07 Rianne van Lambalgen (VU) *When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions*

2012-08 Gerben de Vries (UVA) *Kernel Methods for Vessel Trajectories*

2012-09 Ricardo Neisse (UT) *Trust and Privacy Management Support for Context-Aware Service Platforms*

2012-10 David Smits (TUE) *Towards a Generic Distributed Adaptive Hypermedia Environment*

2012-11 J.C.B. Rantham Prabhakara (TUE) *Process Mining in the Large: Preprocessing, Discovery, and Diagnostics*

2012-12 Kees van der Sluijs (TUE) *Model Driven Design and Data Integration in Semantic Web Information Systems*

2012-13 Suleman Shahid (UvT) *Fun and Face: Exploring non-verbal expressions of emotion during playful interactions*

2012-14 Evgeny Knutov (TUE) *Generic Adaptation Framework for Unifying Adaptive Web-based Systems*

2012-15 Natalie van der Wal (VU) *Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes*

2012-16 Fiemke Both (VU) *Helping people by understanding them - Ambient Agents supporting task execution and depression treatment*

2012-17 Amal Elgammal (UvT) *Towards a Comprehensive Framework for Business Process Compliance*

2012-18 Eltjo Poort (VU) *Improving Solution Architecting Practices*

2012-19 Helen Schonenberg (TUE) *What's Next? Operational Support for Business Process Execution*

2012-20 Ali Bahramisharif (RUN) *Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing*

2012-21 Roberto Cornacchia (TUD) *Querying Sparse Matrices for Information Retrieval*

2012-22 Thijs Vis (UvT) *Intelligence, politie en veiligheidsdienst: verenigbare grootheden?*

2012-23 Christian Muehl (UT) *Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction*

2012-24 Laurens van der Werff (UT) *Evaluation of Noisy Transcripts for Spoken Document Retrieval*

2012-25 Silja Eckartz (UT) *Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application*

2012-26 Emile de Maat (UVA) *Making Sense of Legal Text*

2012-27 Hayrettin Gurkok (UT) *Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games*

2012-28 Nancy Pascall (UvT) *Engendering Technology Empowering Women*

2012-29 Almer Tigelaar (UT) *Peer-to-Peer Information Retrieval*

2012-30 Alina Pommeranz (TUD) *Designing Human-Centered Systems for Reflective Decision Making*

2012-31 Emily Bagarukayo (RUN) *A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure*

2012-32 Wietske Visser (TUD) *Qualitative multi-criteria preference representation and reasoning*

2012-33 Rory Sie (OUN) *Coalitions in Cooperation Networks (COCOON)*

2012-34 Pavol Jancura (RUN) *Evolutionary analysis in PPI networks and applications*

2012-35 Evert Haasdijk (VU) *Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics*

2012-36 Denis Ssebugwawo (RUN) *Analysis and Evaluation of Collaborative Modeling Processes*

2012-37 Agnes Nakakawa (RUN) *A Collaboration Process for Enterprise Architecture Creation*

2012-38 Selmar Smit (VU) *Parameter Tuning and Scientific Testing in Evolutionary Algorithms*

2012-39 Hassan Fatemi (UT) *Risk-aware design of value and coordination networks*

2012-40 Agus Gunawan (UvT) *Information Access for SMEs in Indonesia*

2012-41 Sebastian Kelle (OU) *Game Design Patterns for Learning*

2012-42 Dominique Verpoorten (OU) *Reflection Amplifiers in self-regulated Learning*

2012-43 Withdrawn

2012-44 Anna Tordai (VU) *On Combining Alignment Techniques*

2012-45 Benedikt Kratz (UvT) *A Model and Language for Business-aware Transactions*

2012-46 Simon Carter (UVA) *Exploration and Exploitation of Multilingual Data for Statistical Machine Translation*

2012-47 Manos Tsagkias (UVA) *Mining Social Media: Tracking Content and Predicting Behavior*

2012-48 Jorn Bakker (TUE) *Handling Abrupt Changes in Evolving Time-series Data*

2012-49 Michael Kaisers (UM) *Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions*

2012-50 Steven van Kervel (TUD) *Ontologogy driven Enterprise Information Systems Engineering*

2012-51 Jeroen de Jong (TUD) *Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching*

2013-01 Viorel Milea (EUR) *News Analytics for Financial Decision Support*

2013-02 Erietta Liarou (CWI) *MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing*

2013-03 Szymon Klarman (VU) *Reasoning with Contexts in Description Logics*

2013-04 Chetan Yadati (TUD) *Coordinating autonomous planning and scheduling*

2013-05 Dulce Pumareja (UT) *Groupware Requirements Evolutions Patterns*

2013-06 Romulo Goncalves (CWI) *The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience*

2013-07 Giel van Lankveld (UT) *Quantifying Individual Player Differences*

2013-08 Robbert-Jan Merk (VU) *Making enemies: cognitive modeling for opponent agents in fighter pilot simulators*

2013-09 Fabio Gori (RUN) *Metagenomic Data Analysis: Computational Methods and Applications*

2013-10 Jeewanie Jayasinghe Arachchige (UvT) *A Unified Modeling Framework for Service Design*

2013-11 Evangelos Pournaras (TUD) *Multi-level Reconfigurable Self-organization in Overlay Services*

2013-12 Maryam Razavian (VU) *Knowledge-driven Migration to Services*

2013-13 Mohammad Zafiri (UT) *Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly*

2013-14 Jafar Tanha (UVA) *Ensemble Approaches to Semi-Supervised Learning Learning*

2013-15 Daniel Hennes (UM) *Multiagent Learning - Dynamic Games and Applications*

2013-16 Eric Kok (UU) *Exploring the practical benefits of argumentation in multi-agent deliberation*

2013-17 Koen Kok (VU) *The PowerMatcher: Smart Coordination for the Smart Electricity Grid*

2013-18 Jeroen Janssens (UvT) *Outlier Selection and One-Class Classification*

2013-19 Renze Steenhuisen (TUD) *Coordinated Multi-Agent Planning and Scheduling*

2013-20 Katja Hofmann (UVA) *Fast and Reliable Online Learning to Rank for Information Retrieval*

2013-21 Sander Wubben (UvT) *Text-to-text generation by monolingual machine translation*

2013-22 Tom Claassen (RUN) *Causal Discovery and Logic*

2013-23 PatrÃcio de Alencar Silva (UvT) *Value Activity Monitoring*

2013-24 Haitham Bou Ammar (UM) *Automated Transfer in Reinforcement Learning*

2013-25 Agnes Berendsen (UM) *Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System*

2013-26 Alireza Zarghami (UT) *Architectural Support for Dynamic Homecare Service Provisioning*