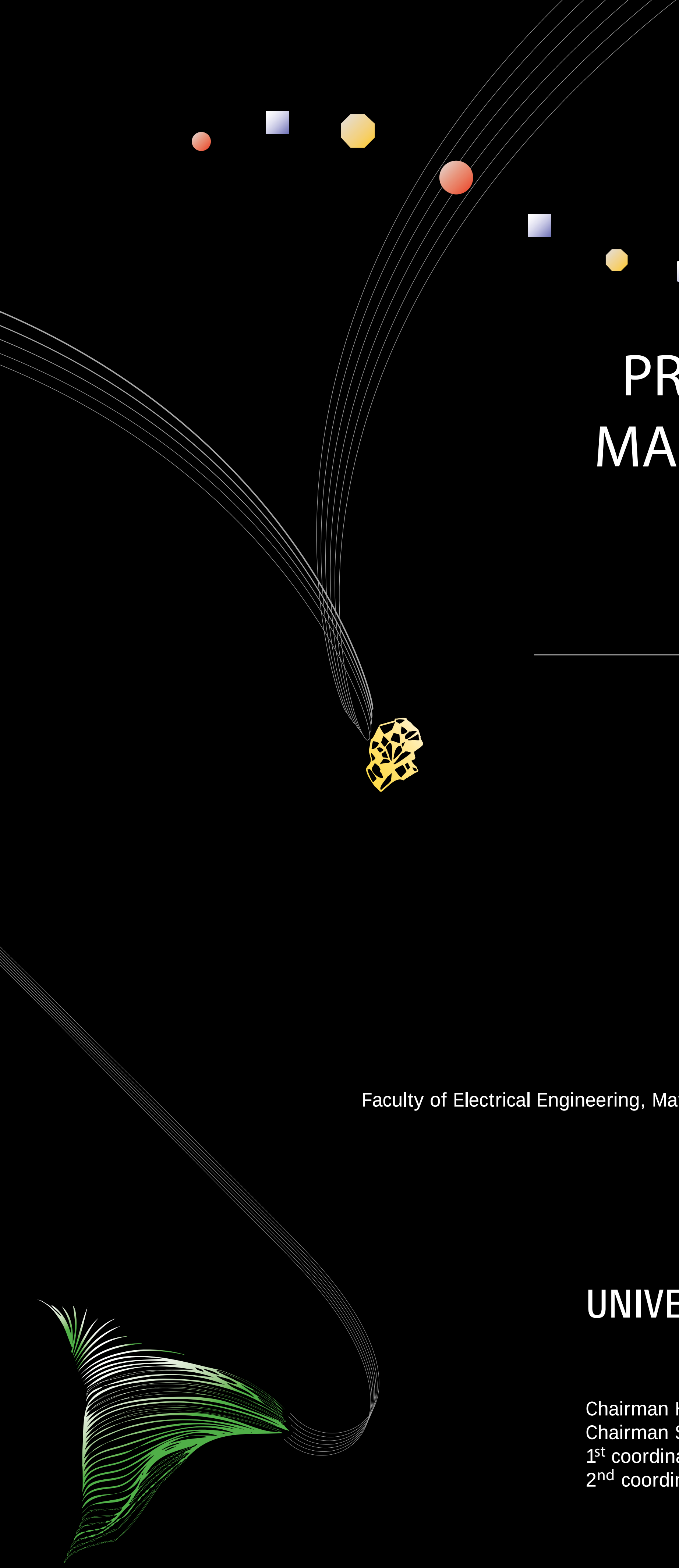


PROVENANCE MANAGEMENT IN PRACTICE

MATTHIJS OOMS

MASTER'S THESIS

Human Media Interaction Group

Software Engineering Group

Faculty of Electrical Engineering, Mathematics and Computer Science

UNIVERSITY OF TWENTE.

GRADUATION COMMITTEE

Chairman HMI:	dr. P.E. van der Vet
Chairman SE:	dr.ir. D. Hiemstra
1st coordinator:	ir. I. Wassink
2nd coordinator:	dr.ir. R. Langerak

September, 2009

Summary

Scientific Workflow Managements Systems (SWfMSs), such as our own research prototype e-BioFlow, are being used by bioinformaticians to design and run data-intensive experiments, connecting local and remote (Web) services and tools. Preserving data, for later inspection or reuse, determine the quality of results. To validate results is essential for scientific experiments. This can all be achieved by collecting provenance data. The dependencies between services and data are captured in a provenance model, such as the interchangeable Open Provenance Model (OPM).

This research consists of the following two provenance related goals:

1. Using a provenance archive effectively and efficiently as cache for workflow tasks.
2. Designing techniques to support browsing and navigation through a provenance archive.

Early in this research it was determined that a representative use case was needed. A use case, in the form of a scientific workflow, can show the performance improvements possibly gained by caching workflow tasks. If this use case is large-scale and data-intensive, and provenance is collected during its execution, it can also be used to show the levels of detail that can be addressed in the provenance data. Different levels of detail can be of aid whilst browsing and navigating provenance data.

The use case identified is called OligoRAP, taken from the life science domain. OligoRAP is casted as a workflow in the SWfMS e-BioFlow. Its performance in terms of duration was measured and its results validated by comparing them to the results of the original Perl implementation. By casting OligoRAP as a workflow and using parallelism, its performance is improved by a factor two.

Many improvements were made to e-BioFlow in order to run OligoRAP, among which a new provenance implementation based on the OPM, enabling provenance capturing during the execution of OligoRAP in e-BioFlow. During this research, e-BioFlow has grown from a proof-of-concept to a powerful research prototype.

For the OPM implementation, a profile for the OPM to collect provenance data during workflow execution has been proposed, that defines how provenance is collected during workflow enactment. The proposed profile maintains the hierarchical structure of (sub)workflows in the collected provenance data. With this profile, interoperability of the OPM for SWfMS is improved.

A caching strategy is proposed for caching workflow tasks and is implemented in e-BioFlow. It queries the OPM implementation for previous task executions. The queries are optimised by formulating them differently and creating several indices. The performance improvement of each optimisation was measured using a query set taken from an OligoRAP cache run. Three tasks in OligoRAP were cached, resulting in a performance improvement of 19%. A provenance archive based on the OPM can be used to effectively cache workflow tasks.

A provenance browser is introduced that incorporates several techniques to help browsing through large provenance archives. Its primary visualisation is the graph representation specified by the OPM. The following techniques have been designed:

- An account navigator that uses the hierarchy captured by the OPM-profile using composite tasks and subworkflows to visualise a tree structure of generic and detailed views towards the provenance data.
- The provenance browser can use several perspectives towards provenance data, namely the data flow, control flow and resource perspectives, identical to the perspectives used towards workflows in e-BioFlow. This enables the end-user to show detail on demand.
- A query panel that enables the end-user to specify a provenance query. The result is directly visualised in the provenance browser, allowing the user to query for certain data items, tasks or even complete derivation trails.
- Retrieve tasks or data items that are not loaded in the provenance browser, but are neighbours of currently visible tasks or data items.

These techniques have already proven their value whilst debugging OligoRAP: error messages, and more interestingly, their cause, were easily identified using the provenance browser. The provenance archive could be queried for all generated pie charts using the query panel, presenting a clear overview of the results of an OligoRAP run.

Preface

The work described in this thesis was carried out between March 2008 and September 2009 as the result of an extended final project combining the two Master of Science studies Human Media Interaction and Software Engineering at the University of Twente, Enschede.

During this period, many things have happened in my life, both good and bad. I happily remember the joyful days on which my nephew Yannick and niece Chiara were born, in contrast to the sad and somber days on which my mother and grandmother passed away. The day of my graduation would have been one of the happiest in both their lives.

I would like to take this opportunity to thank my supervisors personally. You have contributed to many of the good times, and have been a great support in the bad.

Ingo, I hope you do not withdraw your supervision, as you said you would when you were demoted from chairman to coordinator due to university regulations. Regardless of your position in the committee, you have been an excellent supervisor. In the three years that we have worked together, I think we have formed a great team. Under the influence of caffeine during our coffee breaks, many creative ideas were born leading to the improvement of e-BioFlow. I am proud to see how e-BioFlow has evolved, something neither of us had foreseen in the beginning. But at least equally important, I really enjoyed our pleasant non-work related discussions, over drinks at the NBIC conference for instance. Have you heard any fire alarms recently?

Paul, with great pleasure I will always recall the conversations during our meetings, in which you managed to combine sometimes totally off-topic but often surprisingly sharp and funny anecdotes with Ingo's dry sense of humour. Your enthousiasm for my work has been a great motivation that got the best out of me. I am really grateful for all the opportunities you have given me: attending the NBIC 2008 and 2009 conferences in Maastricht and Lunteren, resulting in a poster- and oral presentation of my work and attending the 3rd Provenance Challenge. Not to forget my nomination for the KNAW programme, during which I really enjoyed philosophising about the history of science and of which a publication and visit to the BIOINFORMATICS 2010 Conference in Valencia will hopefully be the result.

Djoerd, although we have not seen each other as much as I have spoken Ingo and Paul, I would like to thank you for always expressing your honest and critical opinion towards my methods. Your feedback was essential for the research results described in this thesis.

Rom, I would like to thank you for your flexibility, for the useful tips and for the inspirational words you gave me the final days before my graduation.

During this research, I have had the privilege to be introduced to Pieter Neerincx, creator of OligoRAP. Pieter, without your quick, accurate and elaborate responses during many debug sessions, OligoRAP in e-BioFlow would still be future work.

This research and thesis benefited from conversations with friends and fellow lab-mates, Sander "4-uur Cup-a-Choco" Bockting, and Wim Bos, but most notably, Viet Yen Nguyen. Viet, since you are probably the smartest person I have ever met, it can be logically implied your feedback has improved this thesis. It is comforting to always see a green dot in front of your name, but also without it, I know you are there for me.

Finally, I wish to express my greatest thanks to my father, mother, my sister and her family, for their love and support.

Matthijs Ooms *Enschede, September 2009*

Contents

Summary iii

Preface vii

List of Figures xiii

List of Tables xiv

List of Queries and Query Plans xv

1 Introduction 1
- 1.1 Provenance as Cache 3
- 1.2 Provenance visualisation 4
- 1.3 Outline of this thesis 5

2 Provenance in Scientific Workflows 9
- 2.1 Scientific Workflow Management Systems 9
- 2.2 Provenance 11
  - 2.2.1 The Open Provenance Model 13
  - 2.2.2 Provenance archive as cache 16

3 Improvements to e-BioFlow 19
- 3.1 Motivation for the use of e-BioFlow 19
- 3.2 Improvement implementation details 20
- 3.3 Proof-of-principle case: OligoRAP 23
  - 3.3.1 Motivation 24
  - 3.3.2 Casting OligoRAP as a Workflow in e-BioFlow 26
- 3.4 Provenance implementation 27
  - 3.4.1 Requirements for provenance implementations 29
  - 3.4.2 Database Design 30
- 3.5 Provenance Recording: an OPM-profile 31

3.6 Running OligoRAP: results 37
3.7 Discussion 40

4 Using provenance as cache **45**
4.1 Caching scheme 45
4.1.1 Cache phase 1 46
4.1.2 Cache phase 2 47
4.2 Implementation 48
4.2.1 Query for cache phase 1 49
4.2.2 Query used in cache phase 2 52
4.3 Optimising performance of phase 1 cache queries 52
4.3.1 Query set and database used in measurements 54
4.3.2 Optimising using subqueries 54
4.3.3 Optimising using indices 60
4.3.4 Performance of querying non-cached tasks 62
4.3.5 Performance of Phase 2 queries 63
4.4 Caching tasks in OligoRAP 64
4.4.1 Results 64
4.4.2 Discussion 65

5 Provenance Visualisation **69**
5.1 Provenance Browser 69
5.1.1 Navigating the Refinement Tree 70
5.1.2 Perspectives 72
5.1.3 Query interface 78
5.1.4 Loading neighbours 81
5.2 Browsing through an OligoRAP run 82
5.3 Debugging with the Provenance Browser 83
5.4 Discussion 85

6 Conclusion **87**
6.1 Summary 87
6.2 Future Work 90

Appendices

A Queries **95**

B Query plans **99**

Bibliography **115**

List of Figures

2.1 OPM entities and causal dependencies. 14

3.1 Exotic MI workflow pattern 28
3.2 Screenshot of OligoRAP in e-BioFlow 29
3.3 ERD of Open Provenance Model 31
3.4 Asynchronous BLAST workflow specification 35
3.5 OPM graph of the execution of a BLAST job 35
3.6 Matching pie chart results of OligoRAP 39
3.7 CPU load during OligoRAP run 41

4.1 Provenance graph of a cache candidate 47

5.1 Refinement tree structure 72
5.2 Refinement tree with provenance elements 73
5.3 Provenance perspectives. 74
(a) Data flow
(b) Resource
(c) Control flow
5.4 Screenshots of perspectives in e-BioFlow 76
(a) Normal perspective
(b) Data flow perspective
(c) Resource perspective
(d) Control flow perspective
5.5 Screenshots of pie charts in e-BioFlow 80
(a) All pie charts
(b) Only Transcriptome pie charts

List of Tables

3.1 e-BioFlow improvements for running OligoRAP 21
3.2 BioMOBY services used by OligoRAP 25
3.3 OPM Database table specification 32
3.4 The OPM-profile 34
3.5 Storage sizes and durations of OligoRAP runs 39
3.6 Provenance data statistics 40

4.1 Query used in cache phase 1 48
4.2 View used for data comparison 51
4.3 Query used in cache phase 2 52
4.4 Item value index sizes 59
4.5 Form 2 and 3 and view 1 and 2 query statistics. 59
4.6 Created indices and their size 61
4.7 Statistics of all indices. 62
4.8 Query performance of non-cached processes. 63
4.9 Query performance of phase 2 queries. 64
4.10 Durations of OligoRAP run with and without cache 65

5.1 Accounts created during an OligoRAP run 83

List of Queries and Query Plans

4.1 Deriving query form 2 from query form 1 55
4.2 Deriving query form 3 from query form 2 57
5.1 Query for retrieving SVG pie charts 79
5.2 Recursive query for retrieving a derivation trail 81
A.1 Ineffective cache query (form 1) 95
A.2 Cache query for MobyBlat in form 2 96
A.3 Cache query for MobyBlat in query form 3. 97
A.4 Cache query for Download url process 97
B.1 Query plan for query A.1 100
B.2 Query plan for query A.2 102
B.3 Query plan for query A.3 104
B.4 Query plan for query A.3 with all effective indices 106
B.5 Query plan for query A.4 108
B.6 Query plan for query A.4 with all effective indices 109
B.7 Query plan for query A.3 110
B.8 Query plan for query A.4 with index i7 112
B.9 Query plan for a phase 2 query 112

The greatest challenge to any thinker is stating the problem in a way that will allow a solution.

Bertrand Russell, 1872 - 1970

Chapter 1

Introduction

Whilst working for a long time on a single topic, this topic becomes so familiar that it is surprising if others, even fellow computer scientists, have never even heard of it. The interest in the main topic of this research, *provenance*, has grown in the last years and has become an established research field. Yet, the people working in this field still form a select group, therefore a proper introduction is in place.

Provenance means origin or derivation, and is also referred to as (data) lineage, audit trail or pedigree. Different techniques and provenance models have been proposed in many areas such as workflow systems and tools, visualisation, databases, digital libraries and knowledge representation. In library systems for example, the structure of articles and their citations form an audit trail that helps the reader determine the quality of articles by its derivation from previous work. In database context provenance is used to capture the changes of data records. The importance of provenance can hardly be overestimated. It can be used to inspect and validate (intermediate) results of workflow runs and pay credit to the owners. Provenance makes results reproducible, which is a very important factor in scientific research.

In this research, provenance is used in the context of workflows, where provenance is primarily used to capture the execution of a workflow run. All intermediate results, timestamps, tasks and metadata of local and/or remote services are recorded by means of a certain provenance model. Workflow specifications are closely related to provenance models, they both define the relation between tasks and data. The difference is that a workflow defines how tasks are *going to be* executed, whereas a provenance model describes tasks that *were* actually executed.

Many workflow systems exist nowadays, such as Taverna, Kepler, Chimera, Vistrails, Triana and our own research prototype e-BioFlow (see §2). Some of these only focus on a certain domain. Taverna for instance provides tools mainly used by life scientists. Kepler is more generally addressed to as a Scientific Workflow Management System, a group of workflow systems to which e-BioFlow belongs as well. These workflow systems are named Scientific, because they are used for running scientific experiments. The workflows they run are scientific experiments themselves, like an experiment performed by a life scientist in his lab. Diverging from the wet-lab, some life scientists now run their experiments in a completely automated fashion. These are called in-silico experiments. A new scientist evolved, called the bioinformatician. Most bioinformaticians have a stronger affiliation with biology than they have with information or computer science. They build software, and small tools, out of *need*, simply because the tools do not exist. Workflow systems, originally used for business administration, form a useful means to model the tasks used in experiments, enabling re-execution of experiments and sharing them. Workflow editors or workbenches have been made more user friendly, so that not only the informatician, but also the life scientist can design and run workflows.

Among the major challenges faced today, that would benefit end-users, among which bioinformaticians, is how to integrate provenance techniques and models so that complete provenance can be derived for complex data products during workflow execution. Some of the workflows systems mentioned earlier capture provenance data, all using their own storage models. Another challenge is to make these different provenance aware systems interoperable. The Open Provenance Model (OPM) is one of the few existing proposals working towards this goal. Yet another issue is scalability. The amount of provenance data captured during workflow execution can be enormous, depending on the granularity, the level at which provenance is captured. Processing all the amounts of data involved in the experiments can take a long time, so efficiency is an important factor. Davidson and Freire [11] have categorised these challenges into four major open research problems:

1. Information management infrastructure and information overload
2. Interoperability
3. Analyse and visualise provenance data
4. Connecting database and workflow provenance

This thesis is the result of a combined final project of two Master of Science studies, namely Human Media Interaction, and Software Engineering. It addresses all four challenges posed by Davidson and Freire, working towards two different research goals. Each study has its own research goal. The first goal, for Software Engineering (SE), is to improve the efficiency of workflow runs, by using a provenance archive as cache. The second goal, for Human Media Interaction (HMI), is to facilitate the interpretation of provenance data by means of a provenance browser that is able to navigate through a provenance archive. The studies share a common subgoal: collecting provenance data during workflow execution. A large-scale data-intensive use case from the life science domain has been identified and casted as a workflow.

1.1 Provenance as Cache

During workflow execution, some tasks can be computationally intensive and thus time-consuming. The task, its input and its results can be stored in some archive, such as a provenance archive. If the output of such a task can be predicted based on its input, and a previous execution does exist in the archive, the output can be retrieved from the archive, by way of cache. By fetching the output directly from cache instead of re-executing the task itself can improve performance. This would be very beneficial in cases a workflow is executed repeatedly, or only a small number of parameters is changed. In an 'ad-hoc' workflow design approach, when part of a workflow is executed, results inspected and new tasks added based on these results, it is desirable to not perform all previous tasks over and over again, especially when they are time-consuming. Caching these tasks would be very helpful.

Besides performance gain, there are various other practical reasons for caching workflow tasks. If a workflow crashes, caching makes it possible to resume execution. Webservices are frequently used in workflows. A drawback of webservices is that the server running them can be overloaded, resulting in a slow performance and they can be unreachable due to network problems. If a task invoking a webservice is stored in cache, the workflow can still be executed.

When provenance data is collected during workflow execution and stored in a provenance archive, all task information, task inputs and outputs can be queried. This is exactly the data and functionality needed for caching workflow tasks.

The large data volumes and different ways to store and query provenance archives make caching of workflows a challenging task. This challenge is one of the main motivations for this thesis and is expressed in the following research question:

> **SE Research Question** *Can a provenance archive be used effectively and efficiently as cache for workflow tasks using the structure of the Open Provenance Model?*

The result of this research is a new caching strategy (see Chapter 4), which is implemented in e-BioFlow. The proposed caching scheme defines how collected provenance data can be used as cache, without affecting the workflow itself. For this cache implementation several improvements were necessary to e-BioFlow. Among the improvements is a direct implementation of the OPM to collect and store provenance data according to a newly defined OPM profile for workflow systems. The implementation of the OPM is tested with a large-scale data-intensive use case called OligoRAP, and the performance improvement that can be achieved by caching tasks in OligoRAP is measured. For a more elaborate description of OligoRAP, see §3.3.

1.2 Provenance visualisation

A provenance archive can be always represented as a directed acyclic graph, see §2.2.1. It is straightforward to use this representation when visualising provenance data, the OPM is a clear example. During workflow execution, many process, actor and data nodes are created, growing a huge provenance graph. Finding data in these large-scale graphs is a hard task: simply presenting the whole graph would not do the trick. Groth [21] showed, with six use cases, that the average overhead of collecting provenance data was about 13%. Of course this greatly depends on the type of use case that is used. In experiments performed with the Large Hadron Collider the amount of data that needs to be interpreted is expected to be hundreds, even thousands of petabytes [13]. Visualising a graph that uses 13% of a petabyte storage space can not easily be performed on a commodity pc, due to memory limitations and processing power, and would not benefit usability, nor facilitate a clearer understanding of the data by the end-user.

Having different levels of detail is useful when visualising data structures by only representing data that matches users needs, enabling to zoom in and out to a certain level. The OPM provides some means to specify provenance data hierarchically, which can be of help when surveying, inspecting and navigating through a large provenance archive. The challenge to facilitate this way of navigation through large provenance archives is expressed in the following research question:

> **HMI Research Question** *Can level of detail be captured in the Open Provenance Model to support browsing and navigation through large provenance graphs?*

The result of this research is a provenance browser that exploits the structure of the OPM by deducing levels of detail. Additionally, several other techniques have been designed and implemented in the browser, such as support for different perspectives, a query interface, and an account navigator that enables a user to load only interesting parts of a provenance graph. The implementation is tested with and illustrated by the same use case OligoRAP.

1.3 Outline of this thesis

First, a literature study is presented in Chapter 2, providing a background of previous work in the field of provenance and workflow systems. Chapter 3 describes several improvements made to e-BioFlow, which were needed in order to run a large-scale data-intensive use case, called OligoRAP. OligoRAP is used as a proof-of-principle case for e-BioFlow. The provenance model identified for collecting and storing provenance data is the OPM, which is implemented in e-BioFlow. A generic mapping between workflow events to OPM entities is made, to explicitly define what information is captured and how it is stored. This mapping is called a profile for the OPM. The OPM-profile presented in §3.5 facilitates provenance capturing at different levels of detail, making use of the hierarchical structure of subworkflows. Levels of detail can be captured in an OPM specification using a Refinement Tree. The proposed profile is used in the OPM implementation. In Chapter 4 a caching strategy is proposed that uses the provenance data generated according to the OPM-profile as cache for workflow tasks. Queries are defined and the database optimised to retrieve the cached tasks and results efficiently. This caching scheme is implemented and the performance improvement measured by caching specific tasks in the OligoRAP workflow. Chapter 5 presents a provenance browser, that is able to browse OPM provenance archives by navigating the Refinement Tree. In addition, users are able to switch between fine-grained and coarse-grained views on the provenance data. The use of the provenance browser is explained with provenance data of an OligoRAP run. The final Chapter 6 summarises this work and highlights future research directions.

Provenance? No, it is not a region in France.

skeptico.blogs.com, July 2009

Chapter 2

Provenance in Scientific Workflows

Workflow systems are being used extensively in the life science domain as well as in other scientific research areas. In this chapter an overview is presented of a selection of prominent scientific workflow systems, which appear to belong to the more popular workflow systems, based on their occurrence in literature. In §2.2 several provenance implementations are described, and more specifically the Open Provenance Model (see §2.2.1), which is the provenance model used throughout this research.

2.1 Scientific Workflow Management Systems

Many Scientific Workflow Management Systems (SWfMS) exist nowadays, such as Taverna [34], Kepler [2], Triana [45], Vistrails [8], Trident-[5] and our own research prototype e-BioFlow [52]. All these systems have in common they are able to compose workflows using a graphical user interface also referred to as workflow editor or workbench. Further, they are able to execute these workflows, by mapping the workflow tasks to either local or remote (web) services. There is much overlap between the functionality of these systems, yet all of them approach scientific workflows from a slightly different angle.

Taverna [34; 35] is developed mainly for life scientists, and is the most prominent workflow tool available in this area. At the current time of writing it provides a collection of over 3500 services, using a variety of protocols such as WSDL/SOAP [33], BioMOBY [53] and Soaplab [39];

the latter two are webservice collections providing a uniform data format and ontology structure. Exact numbers are not mentioned consistently in literature and on the web, but these services are mainly tools for bioinformaticians [12]. Still, Taverna has also been used to enact workflows in other domains, like meteorology. The workflow editor has quite a learning curve since the interface is not always that intuitive. It is announced that the user interface will be improved in future versions. Taverna workflows provide means to iterate over lists, and provide nested processors to define workflows hierarchically.

Kepler [2] has been designed not specifically with the life scientist in mind, but with scientists in general. It comes with some 350 services, called actors in Kepler. These are more general, such as R, Matlab, a generic WSDL actor and a database query actor. While other workflows systems are truly data oriented, like Taverna and Triana, Kepler was designed keeping in mind that workflows executed by scientists have a close resemblance to business process workflows. In addition to (and not in contrast with) business workflows, scientific workflows pose new challenges, such as being computationally intensive and dealing with large and complex derived data products [25]. Being general, life science workflows still can be modelled in Kepler. In a recent combination, called Kepler/pPOD [6] workflows are used for phylogenetic analysis, which belongs to the life science domain. In contrast with Taverna, where workflows are directed acyclic graphs (DAGs), Kepler supports loops as well.

Triana [44] also intends to be a generic workflow system, employable in different distributed and GRID environments. It is used in many domains, varying from gravitational wave analysis to galaxy visualisation to fleet management and biodiversity problems [42]. An important key aspect is its graphical user interface [12].

In Vistrails [8] too, as the name already suggests, the visualisation is an important factor. The focus lies on the visualisation of not only the workflow but also its data. During the exploration process, scientists can gain insights by comparing multiple visualisations, which Vistrails tries to facilitate. Another unique feature of Vistrails is how it deals with workflow modifications. Before the scientist is able to view and analyse final results, the workflow probably has undergone numerous changes. These changes are all stored, since these are considered part of the scientific process.

Trident [5] is a scientific workflow workbench built on top of the commercial workflow enactment engine Windows Workflow. It has been mainly applied and demonstrated in the field of oceanography [1]. It uses several technologies developed by Microsoft. Services can be programmed in .NET, workflows can be connected to other software, such as Microsoft Word, and Silverlight is used to be able to run Trident on multiple platforms.

e-BioFlow [52] started as being only a workflow editor rather than enactor, able to compose workflows using three different perspectives, the control flow, data flow and resource perspective. In a multi-disciplinary environment, an intuitive workflow editor can improve the collaboration between scientists of different research areas. Having only a graphical representation of workflows is not very useful, therefore the YAWL [47] workflow engine was added. YAWL intends to be a complete workflow language, supporting all control-, data and resource workflow patterns as specified by van der Aalst et al. [48]. Another advantage of YAWL is its formal foundation, Petri nets, which enables validation of the workflow. Evolving from these perspectives, new ways of designing workflows are embraced, such as the recent ad-hoc workflow design [50]. During development, usability has always been and still is a key factor.

2.2 Provenance

As was mentioned in the introduction, provenance means origin or derivation [21]. Some SWfMS, such as Taverna, Kepler and Trident, capture provenance information during workflow execution, which is essential to inspect (intermediate) result data [20] and validate experiment results [54]. Despite the high interest in provenance it is still an open research area [9]. Many workshops have been held about the topic, such as the International Provenance and Annotation Workshops of 2006 and 2008 [16; 27].

Provenance data make experiments reproducible, simplify the discovery of changes in the underlying data and can be used to pay credit to the owners of these data and resources [18]. In the life science domain as in any other scientific research field, the trace fulfills a vital function in the quality assurance of the scientific output [17]. SWfMS are ideal environments to automatically capture provenance data. They ‘know’ which resources are accessed, when they are accessed and what data are

[1]Project NEPTUNE, `http://www.neptune.washington.edu/`, last visited July 2009

exchanged between the resources. Therefore, they can manage what is called a process-oriented provenance model [40; 55].

The idea of capturing provenance during in-silico experiments was introduced by Stevens et al. [41]. They mention four different kinds of provenance that can be collected: process, data, organisational and knowledge level respectively. PASOA [29] for instance only captures provenance at the process level. Kepler has workflow provenance support, but its focus is slightly shifted: it records the provenance of changes in the workflow specification made by the user himself, in other words, the evolution of a workflow specification. This idea is also adopted in VisTrails [8]. Hence Kepler and VisTrails capture provenance at the organisational level. According to Barga and Digiampietri [4] workflow systems lack support for the collection of provenance at the data level, Stevens et al. [41] beg to differ and present a counter example: ^{my}GRID. ^{my}GRID (the engine of Taverna) captures provenance at all levels, using a combination of different provenance systems, such as PASAO for the process level, and it uses its own data format to capture and store data at other levels. All the above mentioned SWfMSs use their own models for capturing and storing provenance data. Since all systems use their own data formats, interoperability is a big challenge.

Standardisation improves interoperability. One of the few existing approaches to standardise on a provenance data model is the Minimal Information About Microarray Experiments (MIAME) [7]. MIAME is specifically designed to capture provenance data of microarray experiments. A SWfMS requires a more generic provenance model, since it is able to access a diversity of resources and is not limited to a single type of experiment, such as microarray experiments. The Open Provenance Model specification [31] is one of the few existing proposals to capture provenance in an interchangeable format, directly addressing the interoperability challenge. It is a generic model that intends to capture provenance data in a technology-agnostic manner. Despite all efforts, the OPM does not tackle the interoperability challenge completely yet. Identifying equivalent OPM features among workflow runs of different SWfMS seems intuitive but is often a difficult task [9]. The main idea for the Open Provenance Model was born at the 1st Provenance Challenge [30], in which all teams of the systems described above participated.

2.2.1 The Open Provenance Model

IPAW'06 brought forth the idea of the 1st Provenance Challenge [30], which concluded with a workshop in Washington, DC (September, 2006). Existing provenance models were investigated and compared. When provenance data is used as a means for publication, it is important that an interchangeable format is used. The 2nd Provenance Challenge addressed interoperability between provenance-aware systems and ended with a workshop held in California (June 2007), where an agreement was reached about a core provenance representation amongst the thirteen participating groups [2], called the Open Provenance Model, abbreviated OPM [28]. The 3rd Provenance Challenge ended with a workshop held in Amsterdam (June 2009), during which the OPM specification [31] was evaluated, focussing on interoperability.

One of the goals of the 3rd Provenance Challenge was to stimulate the development and use of concrete bindings and serialisations for the OPM. Currently schemas for XML and RDF exist [3]. A problem of serializing all provenance data (including all data passed between tasks) in a single OPM XML file is that it can result in very large files and will end in scalability problems [40]. How to include data in the value attributes of an OPM XML serialization is undefined. This is still an interoperability issue for hte OPM. Sahoo et al. [37] argue for a Semantic Web approach to the OPM. They present a provenance algebra based on OWL, with a lot of similarity to the OPM (but without accounts).

The OPM is a generic model, that represents the relation between processes (tasks), artifacts (data) and agents (actors, services). Every OPM is a directed acyclic graph (DAG), even when the underlying captured workflow contains loops. Therefore, OPM provenance data is also referred to as an OPM graph. The nodes represent processes, artifacts or agents. Edges represent causal dependencies, such as USED and WASGENERATEDBY. Views on a particular OPM subgraph are called accounts. An account can refine another, representing a more detailed view of the same execution.

DAGs are hard structures to represent: in a DAG two parents can have the same child, hence a DAG is not a tree. When serializing such data

[2]Provenance Challenges Wiki, `http://twiki.ipaw.info/bin/view/Challenge`, last visited July 2009

[3]Open Provenance Model website, `http://openprovenance.org/`, last visited July 2009

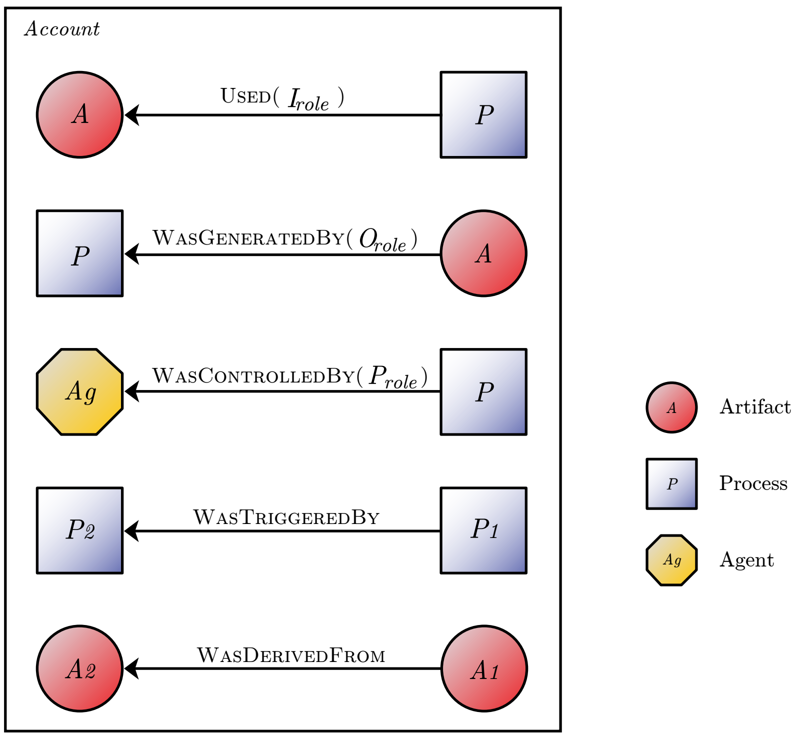


Figure 2.1: OPM entities and causal dependencies.

to XML for instance, cross links and references have to be made, the structure cannot be represented using the hierarchy of the XML directly. A relational database can be used to represent this structure.

OPM structure

Since the Open Provenance Model and its structure plays a major role throughout this thesis, the model is now explained in detail based on the OPM specification, version 1.01. [31].

Entities and causal dependencies The OPM consists of three entity types, Artifacts (data), Processes (tasks) and Agents (actors, services) respectively. In this thesis, entities are referred either as entities or elements. Further, five causal dependency types are defined, also referred to as relations, namely USED, WASCONTROLLEDBY, WASGENERATEDBY, WASTRIGGEREDBY and WASDERIVEDFROM. See Figure 2.1 for a visual representation of the entities and relations. Relations have a cause (its source) and an effect (its target).

For the USED relation, the cause is a Process and the effect an Artifact. It states a certain process P has used a certain artifact A.

For the WasControlledBy relation, the cause is a Process and the effect an Agent. It states a certain process P was controlled by a certain Agent Ag.

For the WasGeneratedBy relation, the cause is an Artifact and the effect is a Process. It states an artifact A was generated by a certain process P (the artifact is the result, or output of the process).

For the WasTriggeredBy relation, both cause and effect are processes. It states that a process P_1 was triggered by some other process P_2.

For the WasDerivedFrom relation, the cause is an Artifact and the effect is an Artifact. It states that artifact A_1 was derived from A_2. The OPM specification defines that the WasDerivedFrom can be derived from a combination of a Used and WasGeneratedBy between two artifacts A_1, A_2 and a process P. If A_1 is the input and A_2 is the output of process P, then A_2 is derived from A_1. During the 3rd Provenance Challenge, there was a long discussion about this relation and whether or not this derivation always applies. None of the participants use the relation in their provenance implementations.

For the Used, WasGeneratedBy and WasControlledBy relations a Role is defined. Roles capture additional information about a relation, in Figure 2.1, the roles I_{role}, O_{role} and P_{role} are used for these relations respectively. I_{role} captures information about the context in which the artifact was used. By a similar argument, O_{role} captures information about the context in which the artifact was generated. P_{role} captures information about the context in which an agent controlled a process.

Account views Relations and entities can belong to Accounts or Account views. Accounts are used to specify views towards the provenance data, at different levels of granularity for example. The granularity of a provenance graph, is determined during its recording stage. Suppose a computer performs some math calculation, say addition of two numbers, and provenance data is collected. At a very fine-grained level, all CPU steps, memory addresses and values are recorded. At a very coarse-grained level, only the calculation itself is recorded as a single process, with the two input numbers and the result.

If entities belong to multiple accounts, these accounts overlap, which is specified in the Overlap relation. The Overlap relation is a relation between two accounts. If one account is captured at a more fine-grained level than another, this can be specified in the Refinement relation.

2.2.2 Provenance archive as cache

Besides using provenance in the traditional way, the provenance archive can also be used as cache, as described by Altintas et al. [1] as a proposal to be implemented in Kepler. They have called this idea smart re-runs. In their approach, parts of the workflow that remain unchanged (when for example a simple parameter is updated) in future executions, are replaced by a StreamActor. This StreamActor fetches the necessary data from the provenance archive. This idea was the result of work previously done, collecting the provenance of the evolution of workflow specifications. In the GRID domain, decentralised caching schemes have been proposed where GRID jobs are represented as workflows [43].

Most caching schemes extend service invocation protocols directly, such as the SOAP extension by Seltzsam et al. [38]. The Taverna Webservice Data Proxy is developed to keep large data sets out of the Taverna engine [4]. However, it can also be used to store intermediate results to serve as a cache in order to speed up the re-execution of workflows. Caching is also useful in case a workflow crashes. Wassink et al. [51] have implemented a workflow to analyse large data sets related to microarray data. They have added additional tasks to support a restore and run option in case the workflow environment crashes. If a SWfMS can use its provenance archive as cache for workflow tasks then restore and run is directly supported.

[4]Taverna Webservice Data Proxy, last visited July 2009, `http://www.cs.man.ac.uk/~sowen/data-proxy/guide.html`

Models are to be used, not believed.

Henri Theil, 1924 - 2000

Chapter 3

Improvements to e-BioFlow

3.1 Motivation for the use of e-BioFlow

One of the main goals of this research and a requirement to reach both research goals is to collect provenance data during the execution of a large-scale data-intensive workflow. An excellent case study was found in the life science domain: OligoRAP. For a more elaborate description of OligoRAP, see §3.3.

In order to collect provenance data for OligoRAP, which was originally written in Perl, OligoRAP had to be casted as a workflow, which requires a workflow system. Although Taverna is the most prominent tool for designing and running workflows in the life science domain, the workflow tool chosen to implement OligoRAP is e-BioFlow. This choice was motivated by the following reasons. Neither e-BioFlow nor Taverna had provenance support at the time, so this had to be implemented in either workflow system. A great plus for e-BioFlow is its workflow engine YAWL [47], that supports amongst others loops and conditional OR-split and joins. It was anticipated that loops and conditions are needed for the polling of asynchronous webservices, which Taverna does not support. A plus for Taverna on the other hand is its support for BioMOBY services [22], the protocol used in OligoRAP to invoke webservices, which e-BioFlow did not support. A drawback of both systems is the use of main memory for storage of (intermediate) results, which was destined to become a problem for the large amounts of XML data generated by OligoRAP.

To summarise, both tools needed many improvements in order to run

OligoRAP and collect provenance data. Adding the support of loops to Taverna means changing its engine, which requires an extensive knowledge of its architecture. Features in Taverna 1 have been developed by many parties in parallel, which did not benefit the design of its architecture. A complete redesign was needed and is currently ongoing work for Taverna 2. e-BioFlow on the other hand has a clearly documented architecture [49]. In addition, the engine core does not need any adaptations to support loops and conditions. A BioMOBY java framework already exists (JMoby [1]) that can be integrated in e-BioFlow with little effort. This made e-BioFlow the primary choice.

Some of the requirements needed to cast and run OligoRAP have been mentioned above, such as BioMOBY support and loops. A complete overview of the functionality needed to cast and run OligoRAP is listed in Table 3.1. The table indicates which functionality was already present in e-BioFlow, before and after casting OligoRAP as a workflow, presenting a clear overview of the implemented improvements.

First an overview is given of the minor implementation details, before continuing with the provenance implementation and details about OligoRAP.

3.2 Improvement implementation details

BioMOBY. JMoby is a Java BioMOBY framework supporting all features provided by BioMOBY registries, such as the invocation of Moby services and the construction of Moby data containers for the input and output of these services without the need to serialize XML. The ontology provided by a Moby service is used to create actors for each service available, amongst which the services needed by OligoRAP.

Moby services distinguish between primary and secondary inputs. Secondary inputs are parameters. A bug was found in the JMoby implementation: all secondary parameters are added with default values if not specified. This conflicts with some of the services used in OligoRAP: not all parameters in the BLAT service for instance can be combined. The bug was fixed in JMoby, only the specified secondary parameters are submitted.

[1]JMoby Project Website: `http://BioMOBY.open-bio.org/CVS_CONTENT/moby-live/Java/docs/`, last visited September 2009

Improvement	**Before**	**After**
Design workflows in different perspectives	X	X
Hierarchical workflow support	X	X
Workflow Engine (YAWL)	X	X
Scripting actor	X	X
Visualise executing tasks in engine view	X	X
Dependency checking using port values	X	X
Late binding	X	X
Loops	X	X
BioMOBY actor		X
BioMOBY data composers and splitters		X
Collection support		X
Pass data by reference		X
Database item manager		X
BioMOBY actor		X
GZIP actor		X
Base64 actor		X
User actor		X
Workflow event: link followed		X
Interleaved parallel routing		X
Run workflows without GUI		X
Data viewer supporting XML and SVG		X

Table 3.1: List of functionality provided by e-BioFlow before and after casting OligoRAP as a workflow

Collection support. BioMOBY supports collections as input and output of services, which was required by the use case services as well. The Perl actor and BioMOBY actor were adapted to enable the correct use of collections.

Database item manager. The architecture of e-BioFlow provides an item manager that stores all items in memory. A reference of a data item is passed to the YAWL engine, instead of the complete data value. This approach made it possible to implement a database item manager, that stores the data values in a database and provides a database item reference consisting of only an id with which the data value can be found.

The passing of data items as reference and storing values in the database partly solves the memory problem mentioned earlier: now only data items that are being processed (for example when checking data items or when passed to a service) are kept in memory. This approach no longer limits workflow execution to main memory size limits but raises the limit to disk storage space.

Actors. The OligoRAP client uses GZIP and Base64 encoding to transfer SVG images. These data transformations are implemented as two (local) actors in e-BioFlow, using the native GZIP and Base64 functionality provided by the Java API.

Another actor has been devised, mainly used for testing purposes: a User actor. This user actor shows an input screen consisting of all inputs the workflow task receives. The task outputs can be edited by the user, if the task has any. If the task only has inputs, it can be used to visualise data during workflow execution. Further, it can serve as a means for simple synchronisation: it waits for the user to continue.

Workflow event: link followed. When a workflow task is started, it is initiated by one or more previous tasks, unless it is the start task of the main workflow. YAWL throws events when a task starts and when a task finishes, but it is hard if not impossible to tell which process invoked which other process, especially when many processes are running in parallel.

YAWL is based on Petri nets [46]. Using Petri net terminology, the link follow event can be seen as the event of a transition that fires: the transition consumes a token from place A and places a new token at place B. In the YAWL source, tokens were extended with metadata: the identifier of their previous place. The YAWL engine was extended at the point where a token is removed from one place and a new one added to another. A new event is thrown in that case. Tasks are modelled as places in YAWL, thus the event can easily be translated to the workflow event of a link followed from task A to B.

Interleaved parallel routing. The workflow pattern 'interleaved parallel routing' , pattern 17 as specified by van der Aalst [48], was required for the use case. This pattern states that the order in which a set of tasks is executed is of no importance, but none of the tasks in the set are allowed to be executed at the same time. YAWL does not directly support this pattern, but it can be implemented in e-BioFlow without changing the YAWL engine, by allowing only a maximum number of instances per actor. If this maximum number of instances is set to one, none of the tasks performed by the actor are executed at the same time.

Data viewer. A data viewer component has been implemented that is able to visualise data items in e-BioFlow. Currently supported visualisations are normal string, XML data and SVG, JPG and PNG images. The viewer can easily be extended to support more. This component is used in the user actor.

3.3 Proof-of-principle case: OligoRAP

OligoRAP [32] is short for 'Oligo Re-Annotation Pipeline'. An essential component of genome-wide microarray-based gene-expression experiments is a high-quality oligonucleotide probe library. In order to maintain this quality, probes have to be updated when new sequence or annotation data is released. An OligoRAP client orchestrates BioMOBY web services to automatically update the annotation for oligonucleotide probes and also check their target specificity.

A widely used service by life scientist is BLAST [3], an RNA/DNA alignment tool that matches particular sequences against genomes present in a database and retrieves matching scores. The BLAST algorithm itself has evolved over the years, and several variants currently exist, improved BLAST algorithms, but also variants such as BLAT [23], which can be considered as 'BLAST on steroids', with the drawback that results are not always found. Several genome databases have been created, such as Ensembl [15] and Entrez Gene [26]. Both provide tools in the form of web services to query and BLAST against the databases.

The OligoRAP pipeline integrates amongst others the alignment tools like

BLAST and BLAT and genome annotations provided by the Entrez Gene and Ensembl project. The result of an OligoRAP run consists of XML files that provide detailed information per oligonucleotide and a quality assessment of the whole array. OligoRAP is a modular and distributed system originally written in Perl. The oligos are processed in chunks containing a configurable maximum number of sequences. A run of the Perl client for a microarray of the mouse (Mus Musculus) consisting of 20K+ oligos takes about 6 hours. For more elaborate details of running OligoRAP using the Perl client see §3.6.

The OligoRAP pipeline consists of eight major BioMOBY services, which can be categorised in six primary steps, see Table 3.2. All invocations of the same service are considered a primary step of the OligoRAP pipeline. The BLAST and concatenate services are secondary, since they depend on the result of the BLAT service and are not always performed.

The Perl client performs all primary steps sequentially. Parallelism is used only for the asynchronous jobs, but the client waits for all asynchronous jobs to be completed before initiating the following primary step. For example, all Oligo Annotation jobs are submitted simultaneously, but the merge step (4) does not start before the last annotation job is finished. The same holds for the BLAST jobs. The first concatenate task starts not before all BLAST jobs are finished. The last asynchronous task, the Oligo Quality analyses, is just a single task, that processes all merged results at once. Since the pie charts can only be generated once the quality is known, they can only by performed when the analysis is completed.

3.3.1 Motivation

OligoRAP makes an ideal proof-of-principle case for e-BioFlow. By specifying OligoRAP in e-BioFlow not only OligoRAP will be better maintainable and more easily customised, but it also enables end-users to better understand the pipeline without studying the Perl code and to share their results. The OligoRAP workflow can be shared, for example through the social sharing medium myExperiment [19].

The amount of data generated during a single OligoRAP run can be enormous. In the case of an OligoRAP run of the mouse, the amount of data of all intermediate and final results is about 3 gigabyte. Analysing data produced during a single OligoRAP run and relaterelating intermediate results is therefore a hard task. This makes it an ideal use case to measure the performance of the provenance archive. Furthermore, the overhead

	Service	Description	Async
1.	Tab2Multi-Sequence-FastaChunks	Convert a comma separated tab file of sequences to chunks of size N, where N is the maximum number of sequences per chunk. This results in an XML file of all chunks. Steps 2 - 4 are performed per chunk.	
2a.	BLAT	BLAT all sequences of a chunk against the transcriptome (UMT) and genome databases.	
2b.	BLAST	If no results were found using BLAT, a BLAST is performed for the particular sequences (only the sequences unmatched using BLAT are BLASTed, not the whole chunk.)	X
2c.	ConcatenateFile	Concatenate the results of BLAT and BLAST (if a BLAST was performed).	
3.	Annotation-Analyser	Analyse the annotations of the previous results for the BLAT/BLAST results of both Genome and UMT.	X
4.	OligoMergeXML	Merge the Genome and UMT results of the AnnotationAnalyser.	
5.	OligoQuality-Analyser	Perform a quality analyses over all merged OligoMergeXML results.	X
6.	MakePieChart	The results of the QualityAnalyser can be visualised using a pie chart service.	

Table 3.2: The BioMOBY services used by OligoRAP, categorised in six primary steps.

of provenance information versus intermediate and final results can be measured.

3.3.2 Casting OligoRAP as a Workflow in e-BioFlow

OligoRAP has been casted as a workflow in e-BioFlow. One of the main advantages of designing a workflow graphically instead of programming in Perl for example, is the intuitive way of modelling parallelism of tasks. Instead of dividing the pipeline in six major steps and perform them sequentially, which is a logical way of programming because it makes the code easier to read, the workflow specification does not wait for each step to complete. Instead, all chunks are processed in parallel, and tasks are started at the moment all their necessary input is known. Thus, once a BLAST is finished, the concatenate service directly starts processing the BLAT and BLAST results. Once the BLAT/BLASTS results are known of both the Genome and UMT, the OligoMerge service starts, and once that service is finished for the particular chunk, the OligoAnnotationAnalyser is invoked. Thus, the OligoAnnotationAnalyser task for chunk A can already be finished, while the BLAT service for chunk B has not even started yet.

By using this more efficient way of parallelism, OligoRAP is already optimised: the runtime was cut in half. Unfortunately the servers accessed by OligoRAP cannot handle the load of executing the synchronous services all at once, therefore the workflow pattern 'interleaved parallel routing' (one of the improvements of e-BioFlow) was used, pattern 17 as specified by van der Aalst [48]. This pattern states that the order in which a set of tasks is executed is of no importance, but none of the tasks in the set are allowed to be executed at the same time. This resulted in some tasks, such as the synchronous BLAT task, still being executed 'in sequence' because no two BLAT jobs are allowed to be executed at the same time.

The asynchronous jobs are performed on a GRID. A GRID manager schedules the jobs based on server load, resulting in a maximum of 20 jobs being performed at the same time. Hence, submitting all jobs together should not give any problems. Unfortunately, this was not the case. A bug was found in the Oligo Annotation Analyse service, jobs were not scheduled properly and too many were executed at the same time. Since the number of connections to the database is limited, at some point the maximum was reached and the OligoAnnotation service returned a connection error. .

Designing a workflow that processes all chunks in parallel turned out to be also quite a challenge. This can be achieved by using the 'Multiple Instances' pattern, pattern 14 as specified by van der Aalst [48], which states that several instances of a task can run concurrently, but have to be synchronised in the end. Although it is true that the YAWL language provides this pattern, the (bèta) YAWL engine implementation does not support it. To overcome this problem, an exotic workflow pattern was used, that has, to our knowledge, never been mentioned before: multiple instances by means of recursive workflow invocation. The pattern is presented in Figure 3.1.

Usually iteration is more efficient than tail recursion in terms of stack space and performance. This is also true for the workflow pattern presented here: Each subworkflow is started in its own thread, and extra tasks have to be executed to split and combine the results. The advantage of this pattern is that all chunks can be processed in parallel, which is not possible using iteration. The time saved by processing all chunks in parallel is, in the case of OligoRAP, greater than the overhead that is the result of the extra tasks that are being invoked, therefore the overall performance of the workflow increases in terms of speed.

Some OligoRAP implementation workflow facts The OligoRAP workflow contains fifteen subworkflows (plus one for the main) and a total of 149 tasks, 9 tasks on average per subworkflow. 35 tasks are composite tasks representing one of the fifteen subworkflows. A single subworkflow was used for all configuration parameters, providing a single location where all parameters can be specified. See Figure 3.2 for a screenshot of the Oligo Quality Analyser subworkflow in e-BioFlow.

3.4 Provenance implementation

In order to collect provenance data during workflow execution, a suitable provenance model had to be either designed or selected. In §2.2.1 the history of the OPM was described, being the result of several (still ongoing) challenges and combined ideas of existing provenance-aware workflow systems. OPM intends to be interoperable. Translated to scientific experiments, this means that scientists can read and understand

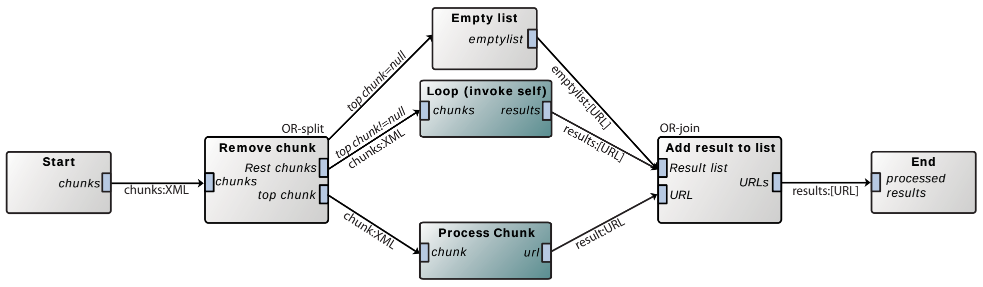


Figure 3.1: Exotic workflow pattern used for the parallel processing of all chunks. The pattern works in a similar way as tail recursion. The input of the workflow consists of all the chunks containing oligos. The 'remove chunk' task splits the chunks, into the head chunk (the first chunk) and the tail (all remainder chunks). The head and tail are the output of this task. If the tail is empty, the Emptylist task is executed next, otherwise the loop task is executed. The loop task is a special case of a composite task, which decomposes into the same subworkflow as the one currently running. The input of the loop task are the remainder chunks, its output a list of the results of all processed remainder chunks. In parallel with the loop task, the composite task 'process' is executed, which processes only the head (note the OR-split, only two of the three outgoing dependencies are enabled after each task invocation). The result of the process task is an URL, identifying where the processed results can be downloaded. The results of the process task and the results of the composite task 'loop' are combined in the task 'Add result to list', which combines the result of the processed top chunk with the results of the processed remaining chunks (note the OR-join, which synchronises on the previous OR-split). Thus, the result of an instance of this workflow is a list of all the URLs of all input chunks. The first invocation of this workflow can be called from another (sub)workflow.

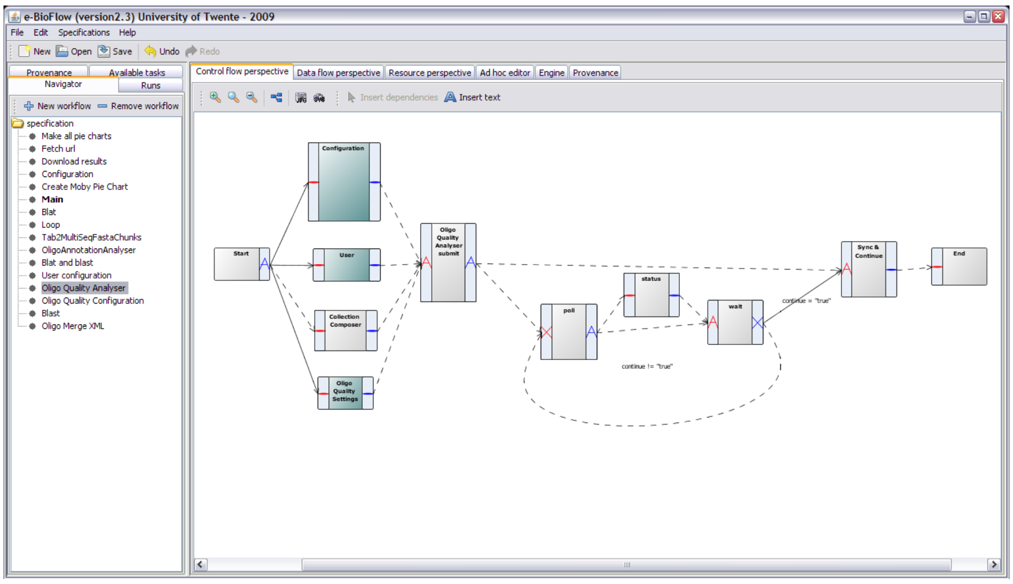


Figure 3.2: OligoRAP in e-BioFlow. In the left navigation panel all subworkflows are specified. The currently selected subworkflow is the Oligo Quality Analyser. It consists of several tasks for submitting, polling (which occurs in the loop) and retrieving the result of the asynchronous job.

each others' lab journals. In workflow context, it means workflow systems are able to exchange their workflow runs, even among different systems. Being designed for interoperability, the OPM is a very generic model.

Because of these reasons the OPM is a very suitable candidate. Unfortunately, no libraries or other direct OPM implementations exist. Therefore, e-BioFlow is improved with a new direct implementation of the OPM. This implementation can be easily adopted by other provenance aware systems, or other SWfMS. An advantage of a direct OPM implementation is that provenance data can be exported, serialized to XML or RDF for instance, without the need to translate from a different internal storage model to the OPM.

3.4.1 Requirements for provenance implementations

Groth [21] points out four non-functional requirements for provenance implementations in his PhD thesis. These are scalability, client independence, ease of installation and feature integration. The implementation proposed here, based on a PostgreSQL database, meets all of these requirements. Sahoo et al. [37] have taken a similar approach, using a relational database back-end as well. The most important requirement is a functional requirement: the archive should be optimised for efficient provenance queries. A relational database implementation in PostgreSQL provides a query language for the OPM, expressed in SQL. The SQL standard defines recursive queries since its fourth dialect, ISO SQL:1999 [14]. In the sixth and latest standard, ISO SQL:2008, recursive queries are referred to as Common Table Queries (CTE). CTE is supported in Postgres since the latest release of July 2009, version 8.4. Recursive queries can be used to query complete derivation trails, an example of this is given in §5.1.3.

3.4.2 Database Design

From the OPM specification, version 1.01. [31], an Entity Relationship Diagram (ERD) was derived, which is presented in Figure 3.3. Recall §2.2.1 for an explanation of the structure of the OPM itself.

The ERD is based on the Timeless Formal Model of the OPM specification. As a result of the 3rd Provenance Challenge workshop, the OPM specification is currently reviewed and one of the change proposals is to remove time from the OPM specification. Instead it is suggested to put the time formalism in a profile [2], such as the one presented in §3.5. The proposed time model in the OPM specification is indeed rather complex. In this implementation a `started` and `finished` timestamp is added only to processes.

Inheritance is used for the entities ELEMENT and RELATION. ELEMENT is the supertable of ARTIFACT, AGENT and PROCESS and RELATION has subtables USED, WASCONTROLLEDBY, WASTRIGGEREDBY and WASGENERATEDBY. The use of inheritance allows generic queries over all elements and relations, as well as specific queries over only a certain type of elements or relations.

For each OPM relation, foreign keys are specified for the columns `cause` and `effect`. In the supertable ELEMENT both foreign keys reference the id in the supertable ELEMENT. In the subtables, specific references to the corresponding subtables are made. For example, the foreign key constraint for the column `cause` of the USED table states it references the id of table

[2]OPM Wiki - Change proposals. Last visited September 2009. `http://twiki.ipaw.info/bin/view/OPM/ChangeProposalMoveTimeToProfile`

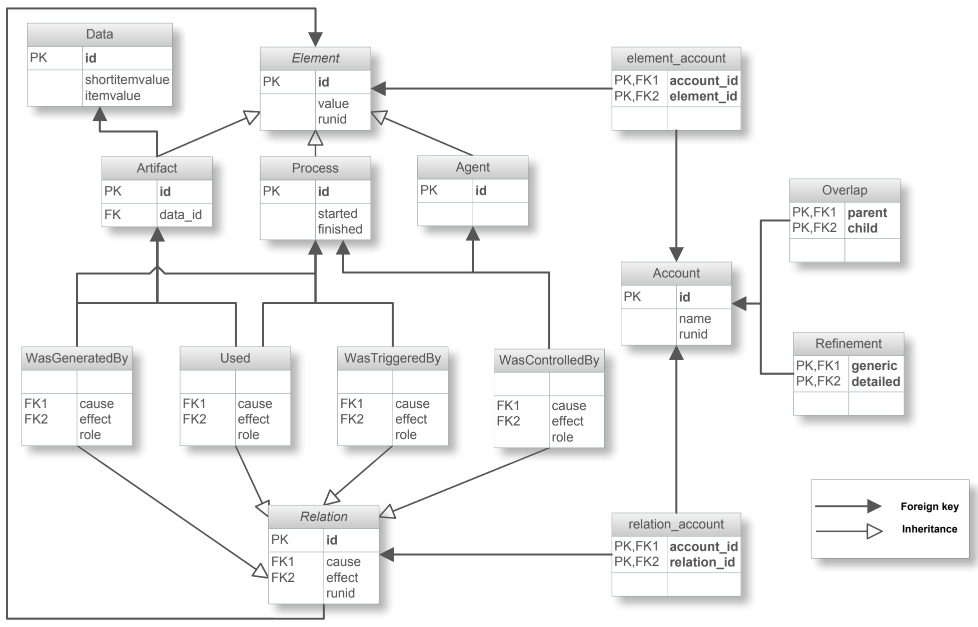


Figure 3.3: Database diagram (Entity Relationship Diagram) of the Open Provenance Model.

PROCESS. By similar argument, the effect column references an id of table ARTIFACT.

The ERD in its turn is used to devise a database schema for PostgreSQL. The schema can be found in Table 3.3. The first table, T0, already existed in e-BioFlow for storing input/output data in a database, instead of keeping this data in memory during workflow execution (see §3.2).

3.5 Provenance Recording: an OPM-profile

The OPM has some limitations because of its generality. Although it is intuitive to map a workflow run to the OPM, such a mapping is not part of the OPM specification, since the OPM intends to be technology-agnostic. It is ambiguous what is actually being captured in the model, how to capture provenance data during workflow execution. To disambiguate, an OPM profile needs to be defined, that makes the mapping between an application and the OPM explicit.

Kwasnikowska and van den Bussche have proposed an OPM profile for the NRC Dataflow Model [24]. This profile is a formal specification,

Table 3.3: OPM Database table specification

```
T0   DATA(ID INT, VALUE)

T1   ELEMENT(ID INT SEQUENCE (ELEMENT_SEQ) NOT NULL,
         VALUE VARCHAR(255),
         RUNID INT NOT NULL,
         PRIMARY KEY (ID))
T2   RELATION(ID INT SEQUENCE (RELATION_SEQ) NOT NULL,
         RUNID INT NOT NULL,
         CAUSE INT REFERENCES ELEMENT(ID) NOT NULL,
         EFFECT INT REFERENCES ELEMENT(ID) NOT NULL,
         PRIMARY KEY (ID))
T3   PROCESS(STARTED TIMESTAMP, FINISHED TIMESTAMP) INHERITS ELEMENT
T4   AGENT() INHERITS ELEMENT
T5   ARTIFACT(DATA_ID INT REFERENCES DATA(ID) NOT NULL) INHERITS ELEMENT
T6   WASTRIGGEREDBY(
         CAUSE INT REFERENCES PROCESS(ID),
         EFFECT INT REFERENCES PROCESS(ID)
       ) INHERITS RELATION
T7   USED(ROLE VARCHAR(255) NOT NULL,
         CAUSE INT REFERENCES PROCESS(ID),
         EFFECT INT REFERENCES ARTIFACT(ID)
       ) INHERITS RELATION
T8   WASGENERATEDBY(ROLE VARCHAR(255) NOT NULL,
         CAUSE INT REFERENCES ARTIFACT(ID),
         EFFECT INT REFERENCES PROCESS(ID)
       ) INHERITS RELATION
T9   WASCONTROLLEDBY(ROLE VARCHAR(255) NOT NULL,
         CAUSE INT REFERENCES ARTIFACT(ID),
         EFFECT INT REFERENCES PROCESS(ID)
       ) INHERITS RELATION
T10  ACCOUNT(ID SERIAL, VALUE VARCHAR(255), RUNID INT, PRIMARY KEY (ID))
T11  OVERLAP(PARENT INT REFERENCES ACCOUNT(ID),
         CHILD INT REFERENCES ACCOUNT(ID), PRIMARY KEY (PARENT,CHILD))
T12  REFINES(DETAILED INT REFERENCES ACCOUNT(ID),
         GENERIC INT REFERENCES ACCOUNT(ID), PRIMARY KEY
     (GENERIC,DETAILED))
T13  ELEMENT_ACCOUNT(
         ELEMENT_ID INT REFERENCES ELEMENT(ID)
         ACCOUNT_ID INT REFERENCES ACCOUNT(ID),
         PRIMARY KEY (ELEMENT,ACCOUNT))
T14  RELATION_ACCOUNT(
         RELATION_ID INT REFERENCES RELATION(ID)
         ACCOUNT_ID INT REFERENCES ACCOUNT(ID),
         PRIMARY KEY (RELATION,ACCOUNT))
```

based on the older v1.0 specification of the OPM (where the REFINES relationship was not yet defined). An implementation for this formal mapping still needs to be designed.

Therefore, an OPM profile is proposed here that makes the mapping explicit between the OPM and workflows executed by any hierarchal workflow system. This OPM-profile is implemented in e-BioFlow.

Table 3.4 presents and explains the OPM-profile. The left, middle and right column specify the workflow events that occur, the OPM entities that are created during the event and the workflow values that are assigned to these entities, respectively. The profile maps workflow tasks to processes, actors (services) to agents and data to artifacts. In the event of a task being executed, a USED relation is created for each input port, between the process and artifact corresponding to the task and data item. The role of the USED relation is given the name of the input port. Likewise WASGENERATEDBY relations are created in the event of a finished task, for each output port. In the event of a control flow link being followed between two tasks, a WASTRIGGEREDBY relation is created between the corresponding processes. For each service that executes a certain task, a WASCONTROLLEDBY relation is created between the corresponding agent and process. The role of the task is assigned to the role of the WASCONTROLLEDBY relation.

Accounts are used to capture different levels of execution detail. For each composite task, two accounts are being created, a coarse grained and a fine grained account. The first one, containing the composite task itself, has low detail and contains the process corresponding to the composite task and all related artifacts. The second provides more detail: it contains all the processes corresponding to tasks executed by the subworkflow, but it does not contain the composite task itself. Between these two accounts, a refinement relation is specified. The two created accounts overlap the account belonging to the (sub)workflow the composite task was executed by, which can be seen as the parent account of the two newly created accounts. This parent/child relationship between accounts is captured in the overlap relation. This relationship represents a tree of accounts, which is further referred to as the *Refinement Tree* (RT).

The OPM-profile is successfully employed in e-BioFlow. Provenance data is captured automatically during workflow execution and stored in a PostgreSQL database.

Table 3.4: The OPM-profile. Mapping OPM entities during workflow execution in e-BioFlow.

Workflow Event	Actions	Value assignments
Main workflow W_{main} started	Instantiate OPM graph Create ACCOUNT Ac_1 for workflow W_{main}	
Task T part of (sub)workflow W started	Create PROCESS P for task T Record start time for process P Create AGENT Ag Create WASCONTROLLEDBY(R,Ag,P) Wcb Create ARTIFACT A_i Create USED(R_i,P,A_i) U_i for each input port i Fetch account Ac_1 for workflow W Add all P, A_i, U_i and Wcb to Ac_1 *In case T is a composite task:* Create ACCOUNT Ac_2 for task T Add all P, A_i, U_i and Wcb to Ac_2 Create OVERLAPS $(Ac_1, Ac2)$	P = Task name Ag = Agent name R = Role name A_i = Input data R_i = Name of input port i
Subworkflow W_{sub} started by composite task T, T part of (sub)workflow W_{parent}	Create ACCOUNT Ac_1 for workflow W_{sub} Fetch ACCOUNT Ac_2 of task T Create REFINES (Ac_1, Ac_2) Fetch ACCOUNT Ac_3 for workflow W_{parent} Create OVERLAPS (Ac_1, Ac_3)	
Task T part of (sub)workflow W completed	Fetch PROCESS P for task T Fetch ACCOUNT Ac_1 for workflow W Record finish time for process P Create ARTIFACT A_i Create WASGENERATEDBY(R_i,P,A_i) Wgb_i for each output port i Add all A_i, Wgb_i to Ac_1 *In case T is a composite task:* Fetch ACCOUNT Ac_2 of task T Add all A_i and Wgb_i to Ac_2	A_i = Output data R_i = Name of output port i
Link followed from task T_1 part of (sub)workflow W_1 to task T_2 part of (sub)workflow W_2	Fetch processes P_1, P_2 for tasks T_1,T_2 Fetch ACCOUNT Ac_1 for workflow W_1 Fetch ACCOUNT Ac_2 for workflow W_2 Create WASTRIGGEREDBY(P_1,P_2) Wtb Add Wtb to Ac_1 and Ac_2	

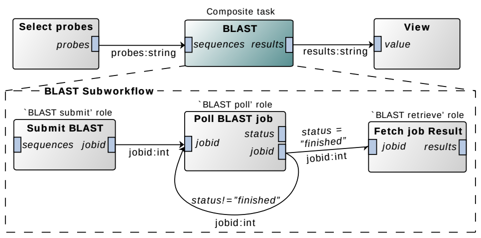


Figure 3.4: Workflow specification performing an asynchronous BLAST job on selected probes. The BLAST task is a composite task, the subworkflow it executes is presented below. Input and output ports are labeled italic. Predicates specifying which control flow links are enabled are italic too and positioned above the corresponding edge.

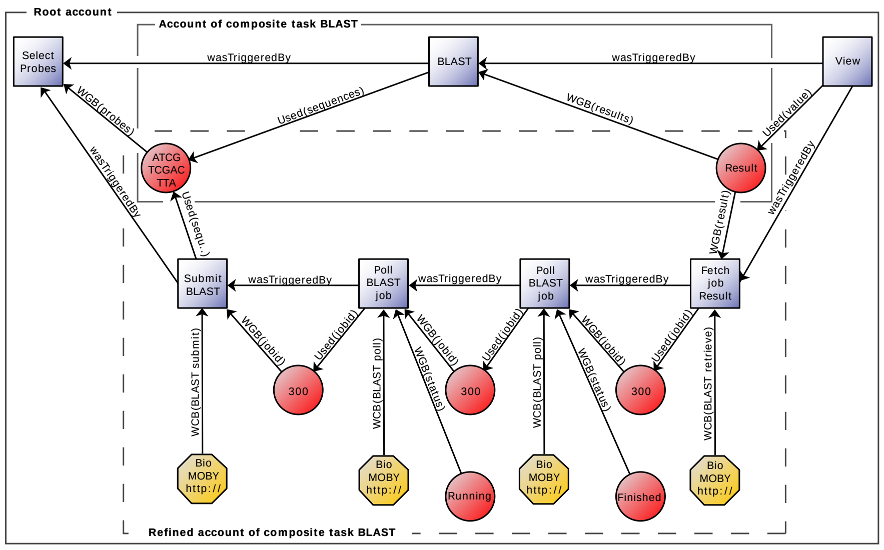


Figure 3.5: OPM graph of the execution of the workflow presented in Figure 3.4 captured using the OPM-profile. Three accounts were generated, the root account and two for the composite task. The rectangle indicating the account corresponding to the subworkflow of the composite task is dashed. Note these accounts overlap: they both contain the artifacts used and generated by the composite task. The root account is the parent of the other two accounts in the RT.

Example Figure 3.4 presents a screenshot of a workflow that executes an asynchronous BLAST job. The first task selects probes from an oligo library. The second task, BLAST, is a composite task that BLASTs all probes. The results are visualised in the third task. The subworkflow that corresponds to the composite task submits the BLAST job to the grid, polls until the job is finished and retrieves the results. The provenance graph generated during execution of this workflow is presented in Figure 3.5. To illustrate how the profile should be used, the first eight workflow events of the execution of this workflow are given that lead to the creation of the first three tasks of this provenance graph and their dependencies and accounts:

1. *Main workflow is started.*
   Create ACCOUNT('Main') Ac_{Main}.
2. *Task 'Select probes' started, part of workflow 'Main'.*
   Create PROCESS('Select probes') P.
   Record start time for process P.
   Create AGENT('Perl actor') Ag.
   Create WASCONTROLLEDBY('Script role',P,Ag) Wcb.
   Fetch account Ac_{main} for workflow 'Main'.
   Add P, Ag, Wcb to Ac_{main}.
3. *Task 'Select probes' part of workflow 'Main' completed.*
   Fetch process P for task 'Select probes'.
   Fetch account Ac_{Main} for workflow 'Main'.
   Record finish time for process P.
   Create ARTIFACT('ACTGCAGA') A_1 for datavalue of outputport 1.
   Create WASGENERATEDBY('Probes',P,A_1) Wgb_1 for output 1.
   Add A_1, Wgb_1 to Ac_{main}.
4. *Composite task 'BLAST', part of workflow 'Main' is started.*
   Create PROCESS('BLAST') P.
   Record start time for process P.
   Create AGENT('Workflow engine') Ag.
   Create WASCONTROLLEDBY('Composite task',P,Ag) Wcb.
   Fetch artifact A_1 for inputport 1.
   Create USED('sequences',P,A_1) U_1 for input port 1.
   Fetch account Ac_{Main} for workflow 'Main'.
   Add P, Ag, Wcb to Ac_{main}.
   Create ACCOUNT('BLAST generic') $Ac_{generic}$ for task P.
   Add P, A_1, U_1 and Wcb to $Ac_{generic}$.
   Create OVERLAPAc_{Main}, $Ac_{generic}$.
5. *Link followed from task 'Select probes' part of (sub)workflow 'Main' to task 'BLAST' part of workflow 'Main'.*
   Fetch process 'Select probes' P_{effect}.
   Fetch process 'BLAST' P_{source}.
   Fetch account Ac_{main} for workflow 'Main'. (Both workflows are the same.)
   Create WASTRIGGEREDBY(P_{effect},P_{source}) $Wtb1$.
   Add $Wtb1$ to Ac_{main}.

6. *Subworkflow 'BLAST subworkflow' started by composite task P, P part of workflow 'Main'.*
   Create ACCOUNT('BLAST detailed') $Ac_{detailed}$ for workflow 'BLAST subworkflow'.
   Fetch account $Ac_{generic}$ of task P.
   Create REFINES($Ac_{generic}$,$Ac_{detailed}$).
   Fetch account Ac_{main} for workflow 'Main'.
   Create OVERLAP(Ac_{main}, $Ac_{detailed}$)
7. *Task 'Submit BLAST' started, part of subworkflow 'BLAST subworkflow'.*
   Create PROCESS('Submit BLAST') P.
   Record start time for process P.
   Create AGENT('BioMOBY BLAST service') Ag.
   Create WASCONTROLLEDBY('BLAST',P,Ag) Wcb.
   Fetch artifact A_1 for inputport 1.
   Create USED('sequences',P,A_1) U_1 for input port 1.
   Fetch account $Ac_{detailed}$ for workflow 'BLAST subworkflow'.
   Add P, Ag, U_1 Wcb to $Ac_{detailed}$.
8. *Link followed from task 'Select probes' part of (sub)workflow 'Main' to task 'Submit BLAST' part of subworkflow 'BLAST subworkflow'.*
   Fetch process 'Select probes' P_{effect}.
   Fetch process 'BLAST' P_{source}. Fetch Account Ac_{main} for workflow 'Main'.
   Fetch Account $Ac_{detailed}$ for subworkflow 'BLAST subworkflow'.
   Create WASTRIGGEREDBY(P_{effect},P_{source}) Wtb.
   Add Wtb to Ac_{main}.
   Add Wtb to $Ac_{detailed}$
9. ...

3.6 Running OligoRAP: results

OligoRAP was run three times: first using the Perl client, second using e-BioFlow without provenance collection but with all data (intermediate and final results) stored in a database, and third using e-BioFlow with provenance capturing enabled. For each run, the same configuration parameters were used. The final results generated by the runs, the quality assessment of the oligos, were compared to each other, to validate the implementation of the workflow: since the configuration is the same, the results should be identical between all runs. Amongst the results of the OligoRAP pipeline is the generation of several pie charts that give an overview of the quality of the microarray. Per run twelve pie charts were generated. All pie charts of the Perl client were identical to those generated by the workflow run. Two of the matching pie charts are shown in Figure 3.6.

Statistics of all the generated (intermediate) results of the three runs, in terms of duration and storage size can be found in Table 3.5. A great advantage of PostgreSQL (and most DMBS) is that it compresses data on-the-fly in the storage layer. Therefore, the table lists compressed and uncompressed size: a Postgres database uses a simple form of LZ compression for text storage, which is fast and pretty efficient for text. Since most data is formatted in XML, a good compression ratio can be achieved. The OligoRAP Perl client does not use compression. Of course all files generated by an OligoRAP run can be compressed in the script, this is a small adaptation, but when they are needed to be read or searched through (which usually was performed using commandline tools by the end-user running OligoRAP), they have to be uncompressed again.

Perl In the first run, the Perl client stores 3.3Gb of data, divided over a total of 994 xml files, grouped in directories per service. Additionally a log file is created. Because the Perl client does not use compression, the compressed storage size is not available. The run takes almost six hours to execute.

e-BioFlow run without provenance The second run by e-BioFlow, where no provenance is collected, stores all (intermediate) results in the database. The total (compressed) storage size on disk of the Data table is 345MB. A database dump, using `pg_dump`, was performed on the database, which contains only this Data table, resulting in a text file containing all data. The file size of the dump was 2.5GB.

e-BioFlow run with provenance In the third run, e-BioFlow stores over 2.9Gb of uncompressed data in the database. These are all the generated (intermediate) results of the OligoRAP run including all extra provenance information. The size is measured again using a dump of the database to a text file. The run takes about three hours to execute. The compressed storage is 392MB, 47MB more than the first run, which corresponds to an overhead in storage size of about 12%. See Table 3.6 for more detailed statistics about the number of created provenance elements and their size.

It might be noted that the (uncompressed) storage size is less for the provenance run than that of the Perl client. This is because the workflow handles intermediate results more efficiently than the Perl client. BLAST and OligoMergeXML are only executed if some oligos are not found with BLAT. OligoMergeXML merges the output of BLAT and BLAST. The

Property	Perl run	e-BioFlow run	Provenance run
Duration	5:56:42	2:50:14	3:03:02
Compressed size	NA	345MB	392MB
Uncompressed size	3.3Gb	2.5Gb	2.9Gb
Files	994	0	0

Table 3.5: Storage sizes and durations of three OligoRAP runs of the Perl client, e-BioFlow without provenance collection and e-BioFlow with provenance collection.

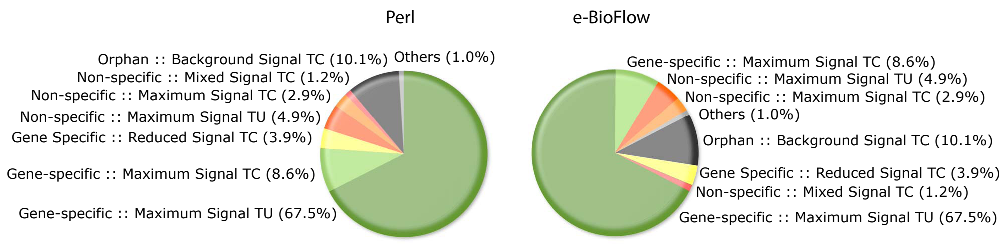


Figure 3.6: Matching pie charts of the microarray quality: transcriptome probe specificity for Mus musculus microarray Compugen Mouse standard + extension with lenient primary and secondary hits generated by the Perl client (left) and e-BioFlow workflow (right).

merged result is used by the OligoAnnotationAnalyser. The Perl client stores the result of BLAT, as well as the input of the OligoAnnotationAnalyser. In case all oligos are found with BLAT, the input of the OligoAnnotationAnalyser is identical to the output of BLAT. The Perl client stores this identical data twice, the workflow passes the intermediate result directly to the OligoAnnotationAnalyser, and the data is stored only once.

The Perl client and the workflow were run several times. The Perl client was executed 4 times, and all results were very close to 6 hours, the fastest being `05:56:42`, the slowest `06:03:54`.

The provenance run was repeated ten times. The fastest run finished in `02:57:03`, the slowest took `04:10:03` due to high server load. Most of the time, the servers were not very busy and the results very close to 3 hours. On average, the provenance run finished in `3:13:09`.

Table 3.6: Provenance data statistics

Table	**Rows**	**Size (in bytes)**	**% of total**	$\sum\%$
Data	64329	361324544	87,87	
Artifact	64329	8994816	2,19	
Task	36704	4325376	1,05	
Agent	33835	3710976	0,90	4,14
WasGeneratedBy	56063	6356992	1,55	
WasTriggeredBy	53080	5709824	1,39	
Used	50710	5537792	1,35	
WasControlledBy	33835	3645440	0,89	5,18
Account	5739	696320	0,17	
Overlap	5738	409600	0,10	
Refines	2869	212992	0,05	
Element_account	153745	10297344	2,50	2,82
Total		(392MB) 411222016	100	12,13

CPU usage During one of the OligoRAP runs, the CPU load of e-BioFlow was measured using JConsole [3]. Figure 3.7 shows a graph of the measured CPU load. The first 20 minutes show a high CPU load, on average about 62%. In this period, the XML is parsed and divided in chunks and for each chunk a separate subworkflow is started that processes it. Then the CPU load decreases and remains low for a rather long period, around 10%, until `00:02:40`. Here the graph peaks shortly, this is where all the results are merged together and the subworkflows of the process chunks are finished. The OligoQualityAnalyser job is started, which takes about 8 minutes during which e-BioFlow is only busy with polling once every 30 seconds, and then a final peak is shown: the pie charts are decompressed.

3.7 Discussion

e-BioFlow supports, amongst others, loops and invocation of BioMOBY services and it can handle large amounts of data, by storing the data in an SQL database and pass the data by reference during workflow execution.

[3]JConsole. Last visited September 2009. `http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html`

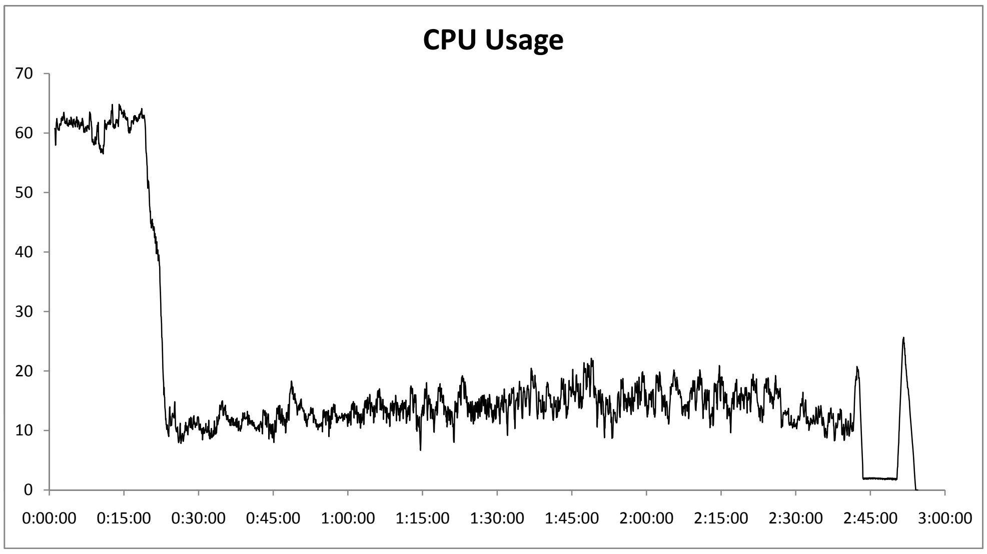


Figure 3.7: CPU load of e-BioFlow during an OligoRAP run.

Using an SQL database has other advantages too. The proven technology is robust and allows optimised querying using indices. The SQL database can easily be extended. The Artifacts table references the already existing Data table so data is not stored redundantly.

OPM-profile. How to store parameters in the OPM is ambiguous. The distinction between parameters and inputs might seem obvious, but, as of many 'obvious' distinctions, they become fuzzier as one discusses them. According to the OPM specification, anything that controls the process in some way should be modelled using agents and WasControlledBy relations. e-BioFlow on the other hand does not treat parameters as entities that control the process but serve as input. The e-BioFlow approach is followed in the OPM-profile, therefore parameters are not explicitly specified but treated as inputs in this OPM-profile.

With this OPM-profile, the mapping between workflows and the OPM is unambiguous. This profile can be adopted by other hierarchical SWfMS as well. The event of a control flow link being followed might not exist in a truly data oriented workflow system, but then it can simply be ignored, resulting in a less rich but still profile-conformant provenance archive.

First OPM implementation. This is one of the first direct OPM implementations. It uses the OPM as the underlying storage model. Other systems, such as Taverna and Kepler, use their own storage models and translate their models to the OPM when they export provenance data.

Overhead. Groth mentions an overhead of the provenance data versus intermediate and final results of about 13% determined by 6 use cases [21]. Of course, the overhead depends on the use case. Large data items, and few tasks generate a small overhead, many tasks but small data items generate a large overhead. Further, it depends on the provenance implementation what information is being stored exactly and how efficiently the data is stored. There are a lot factors involved, therefore it is an interesting result that a similar overhead was measured of about 12%.

BLAT duration. That BLAT is a faster implementation than BLAST was confirmed by the provenance data collected in the first run. On average, the duration of a BLAT task in the first run was 50 seconds, whereas the average BLAST job took 80 seconds to complete, and the BLAST was only performed on a small subset of the oligos processed by BLAT. However, since no two BLAT tasks can be performed in parallel because the server cannot handle the load, these BLAT jobs were performed in ‘sequence’ rather than parallel. This has probably great influence on the total workflow duration. Summing op all BLAT durations totalled 2 hours and 38 minutes.

Those who cannot remember the past are condemned to repeat it.

George Santayana, 1863 - 1952

Chapter 4

Using provenance as cache

In this chapter a caching scheme for workflow tasks is proposed that uses the provenance archive of the OPM implementation as cache. If a workflow is executed multiple times, perhaps with slight parameter changes, some or all of its tasks will be executed with identical input, and generate identical results as in a previous execution. Tasks whose output can be predicted based on its input are called deterministic. If provenance data is collected during workflow execution and a deterministic task is going to be reexecuted, the provenance archive can be queried for previous executions of such a task. If a previous execution is found, the result (output) of the task can be retrieved from the provenance archive, and as from a cache, used directly instead of executing the task again.

In this chapter a caching scheme for workflow tasks is proposed. The scheme has successfully been implemented in e-BioFlow, making use of the OPM implementation described in §3.4. In the following section the caching scheme is described in more detail, followed by implementation details in §4.2. The effectiveness of the cache implementation is measured by caching BLAT tasks in OligoRAP. The queries used for retrieving provenance data for use as cache are optimised, which is described in §4.3.

4.1 Caching scheme

A requirement of any cache implementation is that it is fast. In a workflow context, the lookup and retrieval of results from cache should take no longer than the execution time of the task itself, otherwise caching would be ineffective or even prolong execution time.

Another requirement for caching workflow tasks is that the output of these tasks can be predicted based on the input. In other words: the task should be deterministic. This requirement holds for atomic as well as composite tasks. Even when the composite task itself hides non-deterministic tasks it can still be deterministic. An example of such a deterministic composite task, is given in the subworkflow in Figure 3.4. The input of the composite task consists of some oligos, the output consists of the matching results. Although the intermediate results might differ every time the tasks are executed (the job-id) and even the control flow might differ (the loop can be repeated any number of times, polling until the job is finished), the final result of the subworkflow is always identical; assuming that the database and BLAST algorithm have not changed.

It is not always desirable to cache workflow tasks. Caching non-deterministic tasks might result in incorrect workflow execution. If the poll task *would* have been cached and the first time the poll result status is 'running', the workflow would livelock since the poll task will never return the status 'finished'.

It is very hard if not impossible to determine automatically whether or not a task is deterministic. Therefore, a caching scheme for workflow tasks is proposed here, in which the user determines which tasks are deterministic. Once the workflow is running and a deterministic task is about to be executed, the workflow engine determines whether the results can be fetched from cache.

The scheme for retrieving results from cache presented here consists of two phases.

4.1.1 Cache phase 1

A check is performed whether or not a certain task, that is about to be executed by the workflow engine, is already present in the provenance archive. Not only should the task itself be present in the archive, but its input should equal the input of the task that is going to be executed. Additionally, the actor or service that is going to execute the task should also be the same. The set consisting of such a task, its input and its actor are further referred to as a *cache candidate*. The provenance archive is

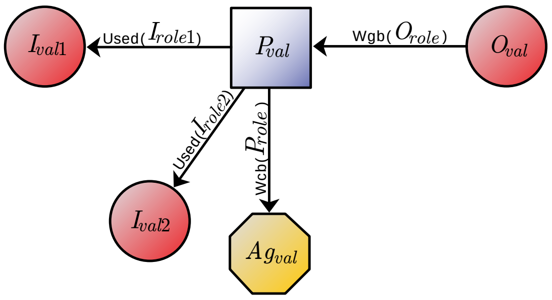


Figure 4.1: Provenance graph of a cache candidate. The provenance archive can be queried for task executions matching this cache candidate using query 1.

queried for previous task executions that match a cache candidate. If a match is found, the task of such a match is returned, so that it can be used in phase 2. Multiple matches can exist in the provenance archive, but only one match is needed. Since the tasks queried for are deterministic, the output of these tasks should be identical for all found matches. If no match is found, the workflow engine signals the actor that it should execute the task, as would have normally been the case when the task was not marked deterministic.

Tasks are defined to be a match if all of the following requirements hold:

1. Tasks have identical names
2. Tasks are (going to be) executed by an identical Actor and have identical task roles
3. Tasks have identical input and output ports
4. Tasks have identical data values for each input port
5. Tasks are finished (they have been executed completely)

4.1.2 Cache phase 2

If a task is found in phase one, the output of this task can be retrieved from the provenance archive and used directly by the workflow engine, instead of executing the task again. The output of a task can consist of multiple data items, depending on the number of output ports. Therefore the provenance archive is queried for all data items of the found task, corresponding to each output port.

Table 4.1: Query used in cache phase 1

Query 1. Phase 1, fetching process ids P_{id} of matching process named P_{val} with role P_{role}, actor named Ag_{val}, input values I_{val1} and I_{val2}, input roles I_{role1} and I_{role2} and output role O_{role}:

```
SELECT p.id as Pid
    FROM process as p, used as u, used as u2, wasgeneratedby as wgb,
      artifact a1, artifact a2, datacompare as d1,
      datacompare as d2, agent ag, wascontrolledby as wcb
    WHERE p.value=Pval AND ag.value=Agval
      AND d1.item1=Ival1 and a1.dataid=d1.item2 AND
      AND d2.item1=Ival2 AND a2.dataid=d2.item2
      AND wcb.cause=ag.id AND wcb.effect=p.id AND wcb.role=Prole
      AND u.effect=a1.id AND u.cause=p.id AND u.role=Irole
      AND u.effect=a2.id AND u.cause=p.id AND u.role=Irole
      AND wgb.effect=p.id AND wgb.role=Orole
      AND NOT p.finished IS NULL
      AND (SELECT count(u.id) FROM used as u WHERE u.cause=p.id)=2
      AND (SELECT count(wgb.id) FROM wasgeneratedby as wgb WHERE
         wgb.effect=p.id)=1
         LIMIT 1
```

4.2 Implementation

The caching scheme is successfully implemented in e-BioFlow for atomic tasks. In theory, the caching scheme can also be used for composite tasks. However, implementing the caching scheme for composite tasks in e-BioFlow would require changes to its workflow engine YAWL. Therefore it was chosen to only implement the caching scheme for atomic tasks. See the discussion section for more details on the difficulties on implementing the caching scheme for composite tasks using YAWL.

The user-interface (workbench) and workflow model of e-BioFlow have been extended with an option for the user to mark certain tasks deterministic. The engine has been extended as well. All tasks marked deterministic become cache candidates during workflow enactment. The provenance archive is queried for executions matching the cache candidate, and if such a task is found the output of this task is used directly by the workflow engine. Two SQL queries are defined for the provenance implementation described in §3.4, one for each cache phase.

4.2.1 Query for cache phase 1

The first query retrieves a single task execution that is present in the provenance archive and matches a cache candidate. Figure 4.1 presents a provenance graph representation of a cache candidate that can be queried using query 1. This cache candidate has two input ports and only one output port (which are stored in the provenance archive in USED and WASGENERATEDBY relations in combination with an artifact for the data value). An example query template is presented in Table 4.1. The query returns the identifier (P_{id}) of a task that matches the property values of a cache candidate, such as task name, actor name etc. These property values are substituted in the query template. Similar query templates can be defined for tasks that have different number of input- or output ports. In e-BioFlow the query template for any task is generated automatically by iterating over the input and output ports and extending the query with the necessary USED or WASGENERATEDBY relations and conditions.

Input- and output port names, input data, actor names and task roles and name, should be substituted in the template with the proper values of the cache candidate that is going to be executed. How these values should be substituted in query 1 is explained here, per requirement (as defined in §4.1):

1. The first requirement, equality of name, is specified in the query using P_{val}. P_{val} should be substituted with the name of the task that is going to be executed. By a similar argument Ag_{val} should be substituted with the name of the actor and P_{role} with the role of the task.
2. The second requirement, equality of actor, is specified in the query using the WASCONTROLLEDBY relation between the specific process (the effect of the WASCONTROLLEDBY relation) and an actor named Ag_{val} (the cause of the WASCONTROLLEDBY relation). The role P_{role} of the WASCONTROLLEDBY should be substituted by the role name of the workflow task.
3. The third requirement, identical input and output ports, is a bit harder to specify. In the OPM-profile input port names are mapped to the role of the USED relation, and output ports to the role of the WASGENERATEDBY relation. By substituting I_{role} and O_{role} with the corresponding input and output port names for the USED and WASGENERATEDBY relations respectively, it is assured a found task has identical input and output ports. However, a task in the provenance

archive could have been executed with even more input or output ports, and still match the query. Therefore, the last two conditions in the where clause of the query check whether the total number of input and output ports is the same.

4. The fourth requirement, tasks having identical data values for each input port, is specified using an artifact that refers to a data item in the data table (using the dataid column). This data item corresponds to the actual value of the data item of the input port. This artifact in its turn is the effect of the USED relation corresponding to the input port. In query 1, only the identifier of the data items has to be substituted (the identifiers I_{val1} and I_{val2}), see §4.2.1 how this data comparison is implemented.
5. The last requirement, tasks have to be finished, is specified in the query by checking that the task has a finished timestamp.

Data comparison

If an actor is about to execute a task, its input has already been stored in the database (and the corresponding artifacts already created, either at the moment the data was generated by a previous task, or at the moment the current task was instantiated). Since all data and artifacts are already stored in the database at this point, in query 1 only the item references have to be substituted in the query, instead of substituting the complete item values in the query.

Query 1 makes use of a view called 'datacompare'. This view is used to match stored data items that are equal. Short and long representations of data items are stored in the data table, in columns named 'shortitemvalue' and 'itemvalue' respectively. The long representation is the complete data value in e-BioFlow XML format, containing metadata such as its syntactic and semantic type. This long representation can be very large. In theory the maximum size that can be stored is of variable unlimited length according to the PostgreSQL documentation [1]. The short representation is a simple string representation of a data item that is no longer than 255 chars. This short representation is generated by the engine at storage time and is used mainly for visualisation, so that data can be represented quickly, but not all data has to be fetched from the database if unneeded. The short item representation can be seen as a (non-unique) hash value of the complete data item, but in a user understandable representation.

[1] `http://www.postgresql.org/docs/8.4/interactive/datatype-character.html`, last visited September 2009

Table 4.2: View used for data comparison.

View 1. A view for matching data items using an index on shortitemvalue.

```
 CREATE view datacompare AS
    SELECT d1.id AS item1, d2.id AS item2
         FROM data d1, data d2
         WHERE d1.shortitemvalue = d2.shortitemvalue
           AND d1.itemvalue = d2.itemvalue

CREATE INDEX shortitemindex ON data USING btree(shortitemvalue)
```

View 2. A view of matching data items using an index on itemvalue.

```
 CREATE view datacompare AS
    SELECT d1.id AS item1, d2.id AS item2
         FROM data d1, data d2
         WHERE d1.itemvalue = d2.itemvalue

CREATE INDEX completitemindex ON data USING hash(itemvalue)
```

The data comparison can be performed efficiently by creating an index on itemvalue. However, the data values used in OligoRAP turned out to be too large, trying to create a btree index (which is the default and fastest index according to the same PostgreSQL documentation as above) over the data values of an OligoRAP run resulted in the error that the index row size was too large. Creating a hash index was tried instead on PostgreSQL 7.4 (which was the version present on the commodity pc used for all experiments), but this resulted in a similar failure. By exporting the database and importing it in a PostgreSQL 8.4 database on another machine, a btree index could still not be created over the complete data values, but a hash index did work.

It would be preferred if the data comparison is supported in older database versions as well. Using a view provides a flexible solution for this problem. The actual comparison of data items is realised using a view called ‘datacompare’, see View 1 in Table 4.2.

By defining two conditions, equality of the shortitemvalue as well as the itemvalue itself in the where clause greatly enhances performance if an index is created on shortitemvalue, since shortitemvalue is a good hash value of the complete itemvalue. This method can be used on (older)

Table 4.3: Query used in cache phase 2

Query 2. Phase 2, retrieving output role O_{role} and output value O_{val} based on process id P_{id}:

```
SELECT wgb.value as $O_{role}$, a.value as $O_{val}$
    FROM wasgeneratedby as wgb, artifact as a
    WHERE wgb.cause=a.id AND wgb.effect=$P_{id}$
```

database implementations that do not support the creation of an index over the complete data value.

If PostgreSQL 8.4 is used as back-end, this view can be easily replaced for one that compares data only using the complete itemvalue, instead of both shortitemvalue and complete itemvalue. See View 2 in Table 4.2. It is expected that view 2 is faster than View 1, since in view 1 we need to recheck all matching shortitemvalues, while in view 2 this is not necessary. The difference in performance between those two views is measured and described in §4.3.

4.2.2 Query used in cache phase 2

In Table 4.3 the query for cache phase 2 is presented. It retrieves the output of one of the tasks found in phase 1, which can be used directly by the workflow engine instead of performing the task again. It uses the found P_{id} of query 1 to retrieve all output port names (in the example only one, O_{role}) and its corresponding item value identifier (since data items are passed by reference), which is stored in the provenance archive in the WASGENERATEDBY relation. This item reference is then used directly by the workflow engine.

4.3 Optimising performance of phase 1 cache queries

The cache implementation was first tested with some small workflows, consisting of a single atomic task that performed the addition of two integers. Once the small examples worked correctly, the implementation was tested more thouroughly by running OligoRAP and cache BLAT and Download tasks.

Since BLAT is one of the tasks that can not be executed in parallel due to server overload, together with its rather long duration, it makes an ideal candidate for shortening the total duration of the workflow if cached. The Download result task has no limitations on parallel execution.

Two variants of BLAT are used in OligoRAP, one for the Genome and one for the Transcriptome. The BLAT variant for the Genome uses two extra parameters to increase its sensitivity. This BLAT task is further referred to as 'BLAT extra'.

Thus in total three tasks were cached, BLAT, BLAT Extra and Download. These three tasks have different numbers of input/output ports, which results in different query templates. BLAT has four input ports and five output ports, BLAT Extra has six input ports and five output ports and Download has only one input port and one output port. Thus in total three different query templates were used.

All OligoRAP services, among which the BLAT service, do not return actual data, but only the URL where the actual data can be retrieved. The Download task retrieves this actual data. If both the BLAT task and the Download task are cached, the data does not have to be downloaded again since it is already present in the provenance archive. A new artifact is created that uses a reference to the previous data item in the data table, so the data item is not stored redundantly, saving storage space. Note that some task URLs in the subsequent run for the download task cannot be retrieved from cache (URLs newly generated by other tasks for example that were not cached, like BLAST). Again, all queries and query durations are logged during the OligoRAP run, so they can be analysed afterwards.

Unfortunately, whilst running OligoRAP for the first time with caching enabled, the workflow seemed to hang, due to the fact that the query for phase 1 for the BLAT task was running. After 5 minutes of running, it was interrupted, since the query was ineffective for caching purposes: a normal BLAT execution normally takes 2 minutes. Clearly the query presented was ineffective for caching purposes, and optimisations were needed.

Two different ways of optimising phase 1 cache queries were used: first by rewriting the query in two other forms, query form 2 and 3 respectively, which is described in §4.3.2. Using query form 2, query performance was improved and running OligoRAP with cache became possible. A second way of optimising query performance was achieved by creating several indices, which is described in §4.3.3. All performance differences of the different proposed optimisations are measured using a representative query set against a database containing the provenance data of two complete OligoRAP runs (a run with and without cache).

4.3.1 Query set and database used in measurements

As described earlier, during execution of an OligoRAP run, all phase 1 and phase 2 cache queries are logged. For the performance evaluation presented in the following sections, it would be rather inefficient to rerun OligoRAP over and over again and measure the performance difference. Instead, a subset of the logged queries of the first successful OligoRAP run was selected, containing all phase 1 queries of the tasks BLAT, BLAT extra and Download tasks that successfully retrieved a previous task execution from the provenance archive in the second run (the queries of the Download task that did not match any results were not included). This resulted in a total of 109 different queries for the query template of BLAT tasks, 109 different queries for the query template of the BLAT extra task and 218 different queries for the Download task.

In all performance measurements, the queries were executed using a simple PHP script that measured the query duration. The queries were executed against the provenance archive containing both OligoRAP runs. Thus, for every query executed, two different previous task executions are present in the provenance archive.

To measure the performance difference between a hash index over the complete data items and btree index over the shortitemvalues, the datacompare view was adapted as described in §4.2.1. Since both OligoRAP runs were performed initially on a 7.4 database, and the hash index can only be generated on 8.4, the complete database was exported and imported into a PostgreSQL 8.4 database, running on another commodity pc.

4.3.2 Optimising using subqueries

The one BLAT query that was ineffective for caching purposes, in query form 1, can be found in Appendix A, Query A.1. Postgres returned a

Query 4.1: Deriving query form 2 from query form 1.

```
Query form 1:

SELECT * FROM process p,
   used AS u1, artifact AS a1, dcompare AS d1,
    ...
   wasgeneratedby AS wgb1,
    ...
WHERE
   t.id = u1.cause AND u1.role=role 1 AND
     u1.effect=a1.id AND a1.dataid=d1.d1id AND d1.d2id=data id 1
    ...
   AND t.id = wgb1.effect AND wgb1.role=role 2
    ...
LIMIT 1

Query form 2:

SELECT * FROM process p
  WHERE
   p.id IN (
    SELECT u.cause FROM used AS u WHERE u.role=role 1 AND u.effect IN (
       SELECT a.id FROM artifact AS a, dcompare AS d WHERE
         a.dataid=d.d1id AND d.d2id = data id 1
      )
    ...
   AND p.id IN (
    SELECT wgb.effect FROM wgb WHERE wgb.role=role 2
   )
    ...
LIMIT 1
```

diversity of query plans for this query, each time a slightly different strategy was chosen. One of these strategies is presented in Appendix B, Query plan B.1. All the strategies have in common that the planner optimises them by relating USED causes and/or WASGENERATEDBY effects to each other, since these are both matched to the same value, namely the id of the process. A sequential scan is performed over the USED and WASGENERATEDBY relations in the query plan at an early stage. Note that in Appendix B those parts in the query plan that are costly or inefficient are colored red and in a bold font, and parts that are improved, either by an index or by rewriting the query, are colored green and underlined.

Phase 1 query form 2

In theory, a more optimal query strategy would be to use the (fast) data comparison using the created index and datacompare view as a first step in the query plan. These can be joined with their corresponding artifacts, which would result in only a couple of matching artifacts. The artifacts in turn can be joined with a corresponding USED relation, on the condition that this artifact is the effect of the relation. Then those tasks that are the cause of the relations can be joined. With this idea in mind, the query was rewritten, using subqueries. See Query 4.1 for the derivation of query form 2 from query form 1.

Query A.1 was rewritten into Query A.2, which can both be found in

Appendix A. A corresponding query plan for Query A.2 can be found in Appendix B, Query plan B.2. It shows that the index and artifact comparison is now performed as the initial step in the query plan (underlined and green). The ineffective query, executed manually, never returned a result in less than 813 seconds, sometimes running for over 1400 seconds, depending on the query strategy used. The new query form resulted in an enormous performance improvement: the query took on average 23 seconds.

Since the new query duration is shorter than the duration of a BLAT task, it can be used effectively for caching purposes. The query is almost six times faster than the actual BLAT execution (which takes on average one and a half minute). Detailed results of cache runs can be found in §4.4. Still, 23 seconds for retrieving data from cache is quite a lot and can be optimised, as we will show, for it is desirable to reduce the query duration even more.

Phase 1 query form 3

Query plan B.2, of Query A.2 shows that the subqueries for counting the number of input and output ports were performed at a very early stage in the query plan. Since selecting and counting all used relations for a certain process is a costly operation, this operation should be performed preferably over a small set of processes. In other words, at a late stage in the query plan, where only tasks matching all other properties remain. A trick that seems to work, is to place all subqueries for counting input and output ports in the SELECT clause. This has two drawbacks however. First, the application should check the total number of input/output ports. Second, the LIMIT clause has to be removed, because the application has to check all tasks that are returned, and this in its turn can even result in a reduction of performance instead of gain. The query form having subqueries moved to the SELECT clause and the LIMIT clause removed, is referred to as query form 3.

In Query 4.2 is shown how query form 3 can be derived form query form 2. A rewritten example of the original form 2 cache Query A.2 in form 3 can be found in Appendix A, A.3. In Query plan B.3 a query plan is shown for Query A.3. Indicated in green and underlined is the desired behaviour, the slow count operations are performed at the last stage in the query plan.

Query 4.2: Deriving query form 3 from query form 2. A relatively slow operation (counting number of used relations for instance) is performed only over a select group of processes that match all other conditions in the where clause.

```
Query form 2:

SELECT * FROM
  process p WHERE

       conditions(p)

   AND (count inputports(p))=x
   AND (count outputports(p))=y
LIMIT 1

Query form 3:

SELECT p.id,
   (count inputports(p)) AS inputports,
   (count outputports(p)) AS outputports
FROM process p WHERE

   conditions(p)
```

Performance difference between query forms and views

In this section, the performance difference is evaluated between query forms 2 and 3 in combination with data view 1 and 2 as described in the previous sections. All form 2 queries in the query set described above were rewritten into form 3, resulting in two query sets. The difference in performance was measured in four runs. Both query sets were executed against both view implementations. The two query sets were executed against the same PostgreSQL 8.4 database containing two OligoRAP runs, a run without cache, and a run with cache (during which the queries were logged). The view in the database was adapted accordingly and the extra index over all itemvalues created, in case performance was measured using view 2.

Additionally, index sizes are measured for the indices used for data comparison.

Results

The index sizes of both the short (shortitemindex) and long item representations (completeitemindex) are presented in Table 4.4 and their sizes are compared with respect to the total database size (provenance data plus data items included) and provenance structure data only (without data items). The hash index over the complete item value takes less space than the btree of the shortitemindex. Compared to the total provenance archive including data items, both indices only take about one percent of total storage space. In respect to the provenance data, without the actual data table, the indices take on average 7.5% of the total storage space.

The results of the performance measurements of form 1 and form 2 query sets (F_1 and F_2) executed using views 1 and 2 (V_1, and V_2) can be found in Table 4.5. The total query duration, average query duration and standard deviation are calculated over all queries, and over the specific queries of a query template (BLAT, BLAT Extra, Download).

As expected, query form 2 using view 1 performed the worst, which is essentially the same configuration as for the first OligoRAP cache run.

View 2 (V_2) has a performance improvement factor over View 1 (V_1) of $\frac{16335.99}{13935.34} = 1.17$, using query form 2. View 2 (V_2) has a performance improvement factor over View 1 (V_1) of $\frac{12010.05}{10045.43} = 1.20$, using query form 3.

Query form 3 (F_3) has a performance improvement over query form 2 (F_2) of $\frac{16335.99}{12010.05} = 1.36$, using view 1. Query form 3 (F_3) has a performance improvement over query form 2 (F_2) of $\frac{13935.34}{10045.43} = 1.39$, using view 2.

The improvement factor of both the combination of V_2 and F_3 over the combination of V_2 and F_2 is $\frac{2911.69}{195.56} = 1.63$.

In all measurements the Download query contributed the most to the total query duration. It has the largest average query duration.

The download query does not seem to benefit at all from the faster data comparison method, which is strange. See discussion for an explanation. It does improve slightly from query form 3, with a factor $\frac{11309.29}{9856.19} = 1.15$, the average drops from about 52 seconds to 45 seconds. The standard deviation of the Download task in query form 3 is much lower, it drops from about 38 seconds to only 2 seconds (on average between view 1 and 2).

The BLAT and BLAT Extra queries contribute the most to the overall performance improvement. As expected, especially the combination of view 2 and query form 3 is effective, a performance improvement of $\frac{2291.76}{80.61} = 28.43$ respectively $\frac{2716.80}{98.50} = 27.58$ was reached for BLAT and BLAT Extra.

Discussion

The query duration of BLAT tasks was improved using query form 3, and the query duration of Download tasks only slightly. This can have two possible causes. One, there exist more Download executions in the provenance archive than MobyBlat executions, 1746 against 436 respectively.

	Index size	**% of total (661 Mb)**	**% of provenance (86 Mb)**
View 1, shortitemindex	7400	1.09%	8.47%
View 2, completeitemindex	5640	0.83%	6.46%

Table 4.4: Item value index sizes.

	Query Set	V_1,F_2	V_2,F_2	V_1,F_3	V_2,F_3
$\sum$	All	16335.99	13935.34	12010.05	10045.43
μ		37.47	31.96	27.55	23.04
σ		30.95	34.03	18.08	22.32
$\sum$	BLAT extra	2716.80	1524.48	1108.84	98.50
μ		24.92	13.99	10.17	0.90
σ		9.50	9.53	1.47	0.32
$\sum$	BLAT	2291.76	1090.55	1034.15	80.61
μ		21.03	10.01	9.49	0.74
σ		6.94	6.78	1.02	0.05
$\sum$	Download	11309.29	11319.19	9780.77	9856.19
μ		52.12	52.16	45.07	45.42
σ		37.77	37.98	3.89	1.03

Table 4.5: View 1 and 2 (V_1, V_2) and Form 2 and 3 (F_2, F_3) query statistics.

Two, the BLAT and BLAT Extra tasks have four, respectively six input ports, against one input port for the Download task. Therefore, the BLAT tasks can benefit most from the combination of these two optimisations.

The Download task does not benefit from view 2, which is an unexpected result. The query plan of this query (see Appendix B, Query plan B.5) shows a sequential scan over the role of all used relations, matching the role name. Since there exist 1746 Download tasks in the provenance archive, which all have the same input port (and thus role name), all task instances are matched. The results are joined with the result of a sequential scan over all WasControlledBy relations, also matching the role name, which are then matched against the process identifier itself; although the completeitemindex is used in the plan, the query does not benefit because of its inefficient query strategy.

The query strategy chosen in the optimal combination, view 2 query form 3, is stable, the standard deviation is small for both the Download as well as the BLAT tasks. This is desirable behaviour for caching. If the retrieval from cache would take a long time in certain cases, this will influence overall performance. Especially whilst caching workflow tasks, since other tasks run in parallel and wait for synchronisation to continue.

4.3.3 Optimising using indices

In Appendix B, Query plan B.3, it is shown that there are many sequential scans performed in a BLAT query. If an index is created on the specific table column present in the condition of the sequential scan, an index scan can be performed instead, which improves query performance.

In an iterative approach, per iteration an index is created based on the query plan of a BLAT query. A sequential scan is indicated, and an index created accordingly. After each created index, all form 3 queries are executed against the same database as used in the previous section, using view 2 (the fastest combination), and their query durations measured.

If an index results in performance loss instead of gain, the index is removed and a new index created corresponding to another sequential scan indicated in the query plan. This is repeated until no more costly steps in the query plan can be pointed out.

Results
In seven iterations, seven indices were created. All created indices and their sizes are presented in 4.6. The names in the table match those found in the query plans presented in Appendix B, B.1-B.9. In these query plans, costly steps in the plan are marked bold and in red, efficient use or improvements in the query plan are underlined and colored green.

The results of the performance measurements of the indices can be found in Table 4.6. The total query duration, average query duration and standard deviation are calculated over all queries, and per query template (BLAT, BLAT Extra, Download).

The first index created, I_1 is an index over the cause column of the used relation, named usedcause. Total query duration of the query set took 197 seconds. Compared to the total duration of the previous section of query form 3 using view 2 the performance improvement factor is $\frac{10045.43}{196.98} = 50.99$.

The BLAT and BLAT extra tasks improved with a factor $\frac{21.03}{0.66} = 31.86$ and $\frac{24.92}{0.76} = 32.78$.

The Download task improved by a factor $\frac{52.12}{0.19} = 274.32$.

The performance of the queries was increased even more with the creation of the three indices I_2-I_4 by a factor $\frac{196.98}{115.12} = 1.71$

Two of the created indices resulted in a performance reduce instead of improvement, these are indices I_5, and I_6, having an improvement factor

Table 4.6: Created indices and their size

Index	Name	Column	Size(Kb)
I_1	`usedcause`	USED(cause)	1856
I_2	`wgbeffect`	WASGENERATEDBY(effect)	1896
I_3	`artdataid`	ARTIFACT(dataid)	2264
I_4	`wcbrole`	WASCONTROLLEDBY(role)	1704
I_5	`wgbrole`	WASGENERATEDBY(role)	3216
I_6	`usedrole`	USED(role)	2784
I_7	`processvaluefinished`	PROCESS(value, finished)	2368
		Total	16088

of $\frac{121.59}{115.12} = 0.94$ and $\frac{1426.7}{115.12} = 0.08$ respectively. Both indices I_5 and I_6 were removed.

The last created index, I_7, improved performance by a factor $\frac{115.12}{108.31} = 1.06$.

This results in a total performance improvement factor of $\frac{10045.43}{108.31} = 92.75$.

The effective indices are I_1-I_4 and I_7. The total index size of these indices is 10088 Kb. In combination with the hash index used in view 2, the completeitemindex, this yields a total index size of 15728 Kb. The size of the total provenance archive containing two runs is 661Mb, and 676Mb with the indices included. The indices thus form 2.27% overhead with respect to the total provenance archive.

The total size of the OPM tables, without the data table, is 85Mb. With indices, this yields a total of 100Mb. The indices thus form 15.27% overhead with respect to the OPM entities alone.

Discussion

The first index, usedcause, was created initially because it was indicated in Query plan B.3 that a sequential scan was performed in Subplan 1, the counting of input ports. The large improvement factor, a 50 times faster than the same queryset without this index, is not only caused by the more efficient way of counting. Instead a new query strategy is chosen, where the usedcause index is used in an early stage in the query plan. Two used relations are joined together on the condition that their causes are a match. This query plan can be found in Appendix B, Query plan B.4. Indices I_1-I_4 and I_7 were present in the database. Note that there are still many sequential scans used over the USED relations and artifacts. The creation of index I_6 leads indeed to an improvement on the average query

		I_1	I_1, I_2	I_1, I_3	I_1-I_4	I_1-I_5	I_1-I_4,I_6	I_1-I_4,I_7
$\sum$	All	196.98	138.39	131.45	115.12	121.59	1426.70	108.31
μ		0.45	0.32	0.30	0.26	0.28	3.27	0.25
σ		0.27	0.20	0.22	0.21	0.22	2.93	0.21
$\sum$	BLAT Extra	82.85	61.19	62.16	58.30	61.72	46.24	55.92
μ		0.76	0.56	0.57	0.53	0.57	0.42	0.51
σ		0.10	0.05	0.03	0.03	0.03	0.01	0.05
$\sum$	BLAT	72.21	49.59	49.56	42.13	44.35	32.96	39.51
μ		0.66	0.45	0.45	0.39	0.41	0.30	0.36
σ		0.08	0.05	0.02	0.02	0.02	0.01	0.04
$\sum$	Download	40.86	26.59	18.70	13.75	14.54	1346.61	11.83
μ		0.19	0.12	0.09	0.06	0.07	6.21	0.05
σ		0.03	0.01	0.02	0.01	0.01	0.22	0.02

Table 4.7: Statistics of all indices. Per query template (All, BLAT, BLAT extra, Download and Tab2Multi) the total query duration $\sum$, average query duration μ and standard deviation σ is calculated.

duration of BLAT tasks, but has a negative influence on the Download task. If I_6 is created, it is used in both query strategies for BLAT as well as Download, see for their query plans B.7 and B.8 respectively.

The Download query factor improved most of all, with an improvement factor of 275. A query plan showing the query strategy for the Download task without indices can be found in Query plan B.5, and one with indices can be found in Query plan B.6. All indices are being used in this query plan. It is interesting to note that this is not the case in the query plan for BLAT tasks, where I_3, the index on the data id, is not being used. Apparently, the query planner decides that the query has been optimised enough and starts executing the plan. Using the index however would probably lead to a better query performance for BLAT as well, as is the case with the Download query.

4.3.4 Performance of querying non-cached tasks

Querying previous task executions that do not exist in a provenance archive can take a long time to execute, even longer than querying tasks that do exist in cache, since all tasks have to be searched through. This especially applies to queries that benefit from a `LIMIT` clause, since no early results are found and the query keeps on running. The `LIMIT` clause was removed in query form 3, thus this is not the case for phase 1 cache queries of form 3.

	Task	View 1	View 2
$\sum$	Download	4.118	3.650
μ		0.008	0.007
σ		0.0008	0.0005

Table 4.8: Query performance of non-cached Download processes.

The performance of phase 1 cache queries is measured using a query set that consists of all the Download queries (such as the retrieval of the results of the non-cached BLAST tasks) that were not available in cache during run 2. In total the set consists of 547 queries.

The queries are rewritten into form 3 and executed against a database only containing provenance data of the first run (not the results of the second cache run, otherwise previous task executions would be found). All effective indices are created (I_1-I_4,I_7). The set was executed twice, once using view 1, once using view 2.

Results

The results of both executions can be found in Table 4.8. As expected, this query also benefits from the faster data comparison in view 2, by a factor $\frac{4.118}{3.650} = 1.13$.

Discussion

Tasks that do not remain in cache are no bottleneck for the cache implementation presented here. Compared to the results of the Download queries that did retrieve a result (see §4.3.3), the non-cached Download queries performed on average ten times faster than the queries that did retrieve a result. It must be noted that the non-cached Download queries were executed against a smaller database, containing only the single run, thus the actual improvement factor is not measured.

4.3.5 Performance of Phase 2 queries

The performance of phase 2 queries is no bottleneck for the cache implementation either, the query can be performed very efficiently, by combining the indices on the effect of WasGeneratedBy (index I_2, see Table 4.6) and the automatically generated index on the primary key of Artifact. A query plan for a phase 2 query can be found in Appendix B, Query plan B.9.

	Task	Duration
$\sum$	Download	0.4271
μ		0.0010
σ		0.0003

Table 4.9: Query performance of phase 2 queries.

All 436 phase 2 cache queries are executed against the database containing the normal provenance run and the cached run, and all effective indices are created (I_1-I_4,I_7). The results can be found in Table 4.9.

4.4 Caching tasks in OligoRAP

As was already mentioned in the previous sections, the first complete OligoRAP cache run was achieved by rewriting the query to query form 2, see §4.3.2. But, it was not successful in terms of performance improvement. Instead, the workflow finished in 4 hours and 21 minutes, which is almost one and half hour longer than the query duration of a normal run without cache. Using the optimisations as described in the previous sections, two new cache runs were performed, using the data comparison view 1 and view 2.

In total, OligoRAP was run three times. In each run the complete OligoRAP workflow was executed and its duration measured. During the first run caching was disabled (no tasks were marked deterministic) but provenance data was collected, filling the archive so that it can be used as cache in the second run. In the two subsequent runs the three tasks, BLAT, BLAT Extra and Download were marked deterministic, becoming cache candidates. To measure the effect of the two different views on OligoRAP, view 1 was used in the second run and view 2 in the third. In cache run 2 and 3, the provenance archive serving as cache contained initially only run 1, which was achieved by exporting the database after run 1, and import it again for run 3.

4.4.1 Results

In Table 4.10 the total duration of the three runs is shown and the calculated sum and average duration of the BLAT tasks, and Download tasks, was calculated.

Table 4.10: Durations of OligoRAP run without cache, with cache and with optimised cache.

Duration	**No cache**	**Cache (V_1)**	**Cache (V_2)**
Total	02:58:45	02:25:08	02:46:07
$\sum$ BLAT	02:36:26	00:27:56	00:09:17
μ	00:01:26	00:00:15	00:00:05
$\sum$ Download	00:55:56	01:35:06	00:42:59
μ	00:00:04	00:00:07	00:00:03

The second run finished 33 minutes faster than the first run, in which results were fetched from cache and view 1 was used. Compared to a total duration of three hours of the first run, a relative performance improvement of about 19% was achieved.

The third run finished almost 10 minutes faster than the first, a performance improvement of 7%.

It is strange that the faster cache implementation using view 2 performed worse than the cache implementation of view 1. See discussion for an explanation.

4.4.2 Discussion

Faster cache implementation, slower OligoRAP

Although the total duration of the BLAT tasks is almost 20 minutes less in cache run V_2 over run V_1, OligoRAP itself performs worse. Since the BLAT tasks finish faster, more of the subsequent tasks, such as OligoAnnotationAnalyse, are submitted simultaneously. This leads to higher load on the server, resulting in worse performance. That the server was busy is supported by the fact that some of the OligoMergeXML tasks triggered a timeout. Timeouts only occured in the fasteset cache run. When a timeout occurs, e-BioFlow tries to execute the task again. Cache run V_2 was repeated another time, resulting in a duration of `02:48:02`, which is very close to the first result. In this run timeouts occurred as well. Apparently, the scheduling of tasks whilst using the faster cache implementation is unlucky for an OligoRAP run.

Parallelism

Because of parallelism, the impact of caching is less (the total time saved for all BLAT tasks is `2:08:30`, yet the total time saved on total workflow is execution is `00:33:37`. If all tasks were executed in sequence by the OligoRAP workflow, the effect of caching would have been greater than 19%, since the total workflow duration would not have been influenced by synchronisation tasks. By a similar argument, if more tasks were executed in parallel, the relative effect of caching would be less. It is hard to predict beforehand what the relative effect of caching will be on total workflow duration, but if retrieving a task from cache is faster than the normal execution time of a cached task, this will usually lead to performance gain.

Composite tasks

The caching scheme could, in theory, be applied for composite tasks as well, but the design of YAWL [46], the workflow engine used in e-BioFlow, is hard to extend. One of the difficulties is that a composite task itself does not have inputs nor outputs in YAWL, due to the fact that it is based on Petri nets. It uses global and local (net) variables to store data, and uses these variables to exchange data between tasks, which are modelled as places. e-BioFlow has built its own data layer, using the global and local variables and mappings of YAWL to overcome this problem.

YAWL starts the first task of the subworkflow in a seperate thread (a new net in YAWL terms) for each composite task, and that is basically all that happens. A composite task is not started ('checked out of the engine' in YAWL terms) or finished ('checked into the engine'). To implement caching for composite tasks in YAWL, if a task is found in cache, all tasks enabled after the composite task should be started. This implementation requires an in-depth understanding of the core of YAWL and how it uses Petri nets and tokens exactly, without breaking its formal fundament. A simple solution to this problem is to add an extra task in the workflow specification that checks whether or not the composite task exists in cache. This is not a very elegant solution, since it would generate divergent provenance data.

Influence of other processes on query times

It can be argued that the long query duration was influenced by the fact that e-BioFlow and other processes were running simultaneously, and indeed e-BioFlow demands quite a lot of CPU power orchestrating all the tasks needed to process the chunks in parallel, especially in the beginning when all chunks are processed and parallel tasks are started (see §3.6). Therefore all queries were reexecuted manually without e-BioFlow running when testing the performance improvements of the optimisations, and a large number of queries having identical query templates were executed to measure the average duration per query template.

Test environment vs. OligoRAP
In the test environment, all queries were executed against a database that contained the provenance data of two runs. The first run without cache and the second with caching enabled. Any cache candidate that was found during the second run, thus has at least two matching previous executions in the database. This is slightly different from an actual OligoRAP run, where the provenance archive grows during the second OligoRAP run and only a single previous execution of a task is present per cache candidate. For the measurements of the performance improvement of all optimisations, our main interest is in the relative difference (the improvement factor). Therefore the difference between test environment and OligoRAP is not important.

Significance of performance
The first OligoRAP cache run was run only once, and the second cache run twice. Therefore it might be argued that the difference in performance is not significant. However, the provenance run itself was run ten times (see §3.6) with an average of `3:13:09` and the fastest run finished in `02:57:03`. The duration of both cache runs is less, which makes it acceptable that caching is the reason for both performance improvements.

Correct workflow execution
To test the correct behaviour of the implementation whilst caching workflow tasks, the results of the cached OligoRAP run were validated by comparing the final output of both OligoRAP runs. The pie charts and the generated XML data in the second run were matched against the results of the first run (see §3.6 for a similar comparison of these pie charts). All results were identical.

Try out your ideas by visualising them in action.

David Seabury, 1885 - 1960

Chapter 5

Provenance Visualisation

Being able to easily inspect and validate results of scientific experiments is of great importance. If provenance data is collected while running scientific experiments, the structure of the provenance model can aid whilst navigating through results. The proposed OPM-profile in §3.5 captures the hierarchical structure of workflows in the provenance data using accounts, refinements and overlaps, called the Refinement Tree (RT). The hierarchical structure captured in the RT intuitively follows from the hierarchy of the workflow, by specifying both a coarse and fine grained view of the provenance data of each (sub)workflow. These views are called accounts in OPM terminology.

In this chapter, a provenance browser is introduced, that uses the hierarchical structure in provenance data for browsing and navigating through a provenance archive, visualising data on demand.

5.1 Provenance Browser

The primary visualisation of the provenance data uses the graph structure of the OPM, as described in §3.5. If an interesting actor or process is found, double-clicking it shows its properties. For processes a summary of all input and output ports is presented. The data of Artifacts can be visualised using specific viewers for their corresponding data types. e-BioFlow is improved with several data viewers (see §3.2), such as an SVG viewer which is used for the visualisation of the pie charts generated by OligoRAP, and an XML viewer that highlights XML data, improving readability. The data viewer component supports multiple data types for a single artifact. An SVG picture can be rendered, but its XML can also be inspected.

In addition to the primary visualisation, four more techniques have been designed that help the user navigate through a provenance archive. First, the structure of the RT can be used to navigate a provenance graph, using account views. Second, three perspectives can be chosen from which the provenance data can be viewed. Third, a query interface is presented, which allows a user to express a provenance query that returns artifacts, processes, and/or agents and visualises the results directly. Fourth, the browser can be instructed to load and highlight neighbours of any visible artifact, process or agent. These four techniques are explained in more detail in the following sections, but first the concept of neighbour elements is defined more precisely.

Neighbours. A neighbour of an OPM element, being either a process, artifact or agent, is an OPM element that is directly related to it, by a WasControlledBy, WasGeneratedBy or WasTriggeredBy or Used relation. The element can be the cause or effect in such a relation, the neighbour of a cause is its effect and vice versa. First order neighbours are direct neighbours. Second order neighbours are the neighbours of neighbours, etc. When an element is selected in the provenance browser, all elements except its neighbours and the selected element itself are painted in grayscale, thus highlighting the colored selected elements and their neighbours. The user can directly see which elements are first order neighbours, which is hard to determine when many relations exist between elements.

5.1.1 Navigating the Refinement Tree

As was previously shown in §3.6, provenance graphs can become very large. During an OligoRAP run over 3×10^5 elements and relations are created. Showing the complete graph for such an OligoRAP run would not be a very clear representation of the provenance data. Finding a single interesting artifact or process is as time consuming as browsing a log file. Instead of showing the complete graph, the provenance browser makes use of levels of detail specified in the RT to navigate through accounts. Once an interesting account is found, the elements belonging to this account are visualised. Only that part of the total provenance graph the user is interested in is shown.

Refinement Tree Structure. Figure 5.1 presents an example refinement tree. A refinement tree starts at a root (main) account. In the example given, the main account overlaps the accounts G1,D1,G2,D2,G3 and D3. D1 in its turn overlaps with G4 and D4, D2 with G5 and D5, etcetera. The structure of the refinement tree presented here is characteristic of an RT generated by the OPM-profile presented in §3.5: refinements form groups of two accounts, a generic and detailed account. This is specified in the refinement relation. Only detailed accounts are parent accounts, which is specified by the overlap relation.

The OPM-profile distributes the composite task and inputs and outputs over the generic account and detailed account. An example of such a distribution is visualised in Figure 5.2. The generic account only contains the composite task and its input and output. The detailed account does not contain the composite task, but does contain its input and output, since the input and output is used and generated by the subworkflow of the composite task. Thus the input and output of the composite tasks is contained in both generic and detailed account. Additionally, the detailed account contains all tasks and intermediate results of the subworkflow. The parent account (in Figure 5.2 the main account) of the detailed and generic account contains all the elements of both child accounts.

Account navigator. The complete RT is visualised in a component called the Account Navigator. If an account is selected in the Account Navigator, all elements and relations of this account are shown in the provenance browser. If the selected account is a parent account, all elements of its child accounts are shown as well.

If two child accounts form a refinement pair, only the elements of the generic account of the refinement pair are shown, the elements of the detailed account remain hidden.

Expanding and collapsing generic accounts A generic account view can be expanded to the detailed account view with which it forms a refinement pair. If the user expands an account, all elements of the generic account will be hidden and the elements of the detailed account shown. Elements that are contained by both remain visible. Thus, at first only the selected account and its overlapping generic accounts are shown, but it is possible to switch between the generic account view and detailed account views, hiding coarse grained and showing more fine grained provenance information.

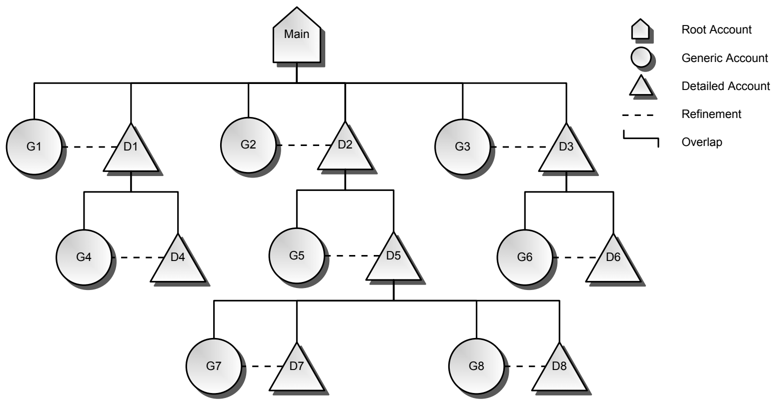


Figure 5.1: The structure of the refinement tree. Detailed accounts are named D_x, generic accounts G_x.

In a similar fashion, detailed accounts can be collapsed to a generic account view with which it forms a refinement pair. All elements of the detailed view are hidden, and the elements of the generic account are shown. Elements contained by both remain visible.

5.1.2 Perspectives

Hiding non-interesting entities results in a clearer view, providing detail on demand. e-BioFlow has adopted this idea in its user interface for designing workflows. It distinguishes between data flow, resource and control flow perspectives of workflows [52]. Similar perspectives can be used for visualising a provenance graph.

Normal perspective. Initially, the provenance browser shows all elements and relations. In Figure 3.5 the provenance graph of the asynchronous BLAST job subworkflow is presented in normal perspective. In Figure 5.3 the graph is presented in data flow, control flow, and resource perspective, which are now explained in more detail. See Figure 5.4(a) for a screenshot of this perspective in the provenance browser implemented in e-BioFlow.

Data flow perspective. When designing workflows in e-BioFlow in data flow perspective, processes and their input and output ports are shown.

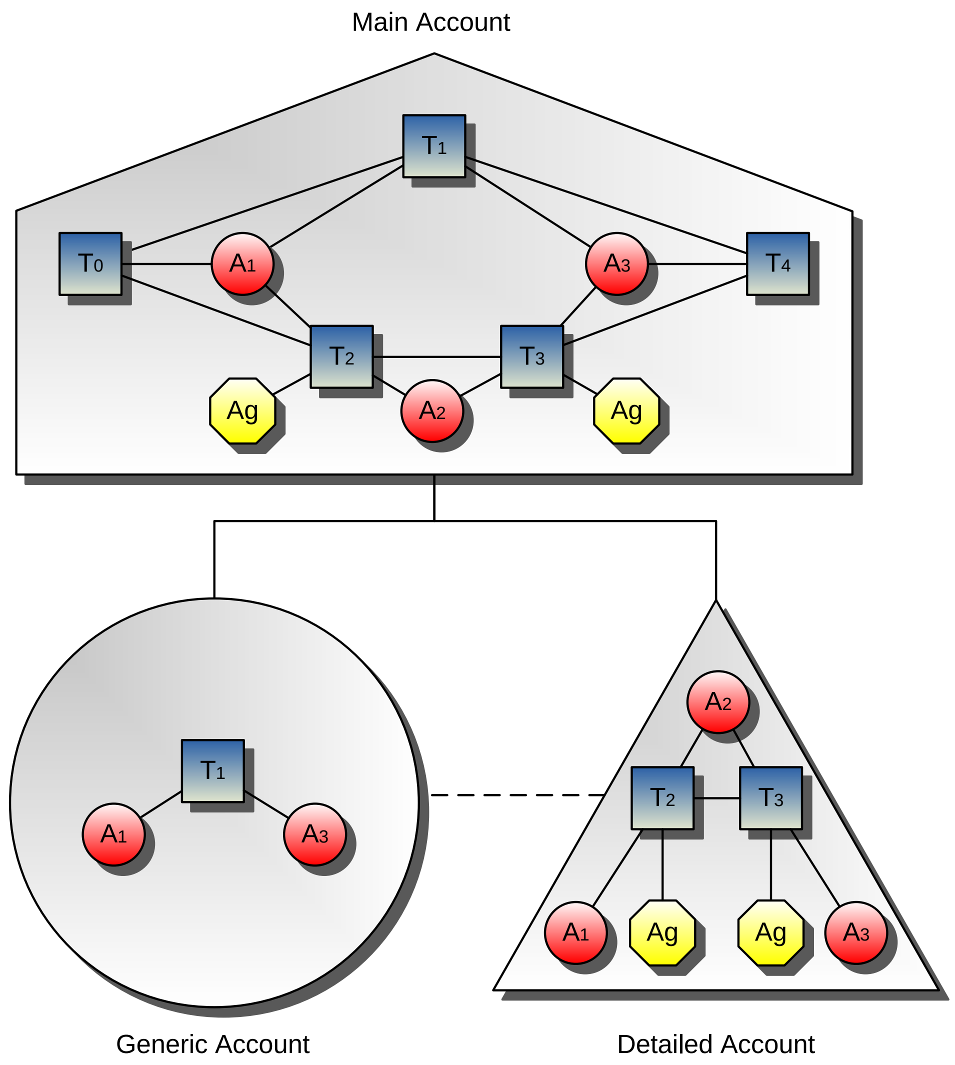
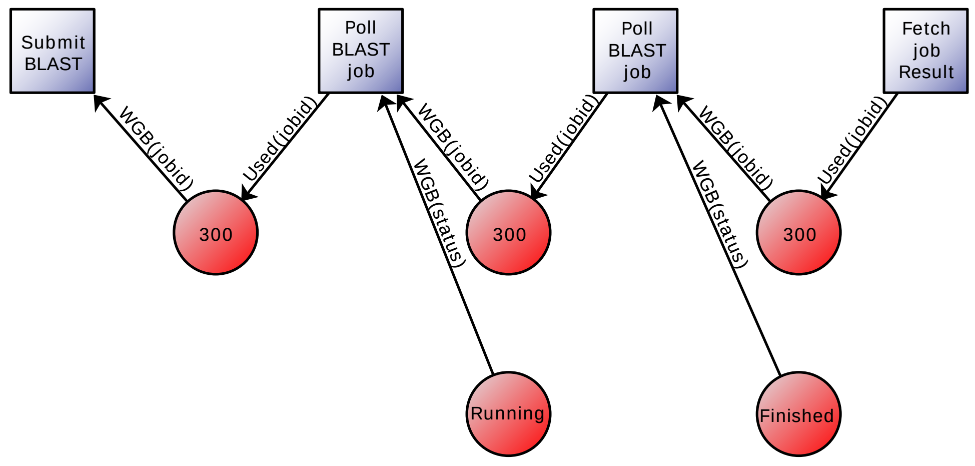
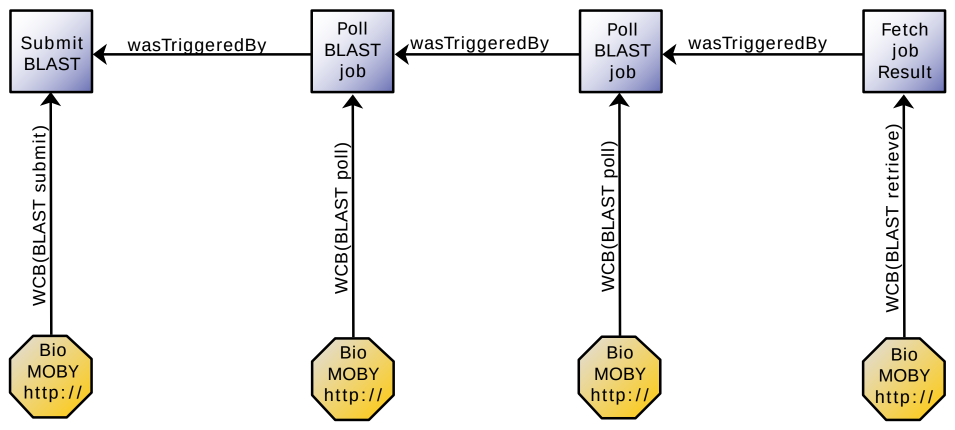


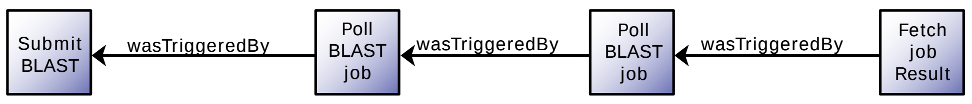
Figure 5.2: Refinement tree with provenance elements for the three accounts. Note that the provenance elements are contained by multiple accounts. The main account, that overlaps the generic and refined account, contains all elements of both accounts, plus two additional atomic tasks T_0 and T_4. The generic account only has artifact A_1 and A_3 in common with the detailed account. They do not share the composite task T_1.



(a) Data flow perspective: only artifacts, processes, WasGeneratedBy and Used relations are shown.



(b) Resource perspective: only processes, agents WasControlledBy, and WasTriggeredBy relations are shown.



(c) Control flow perspective: only processes and WasTriggeredBy relations are shown.

Figure 5.3: Provenance perspectives.

Links between processes indicate the flow of data, connecting output ports to input ports.

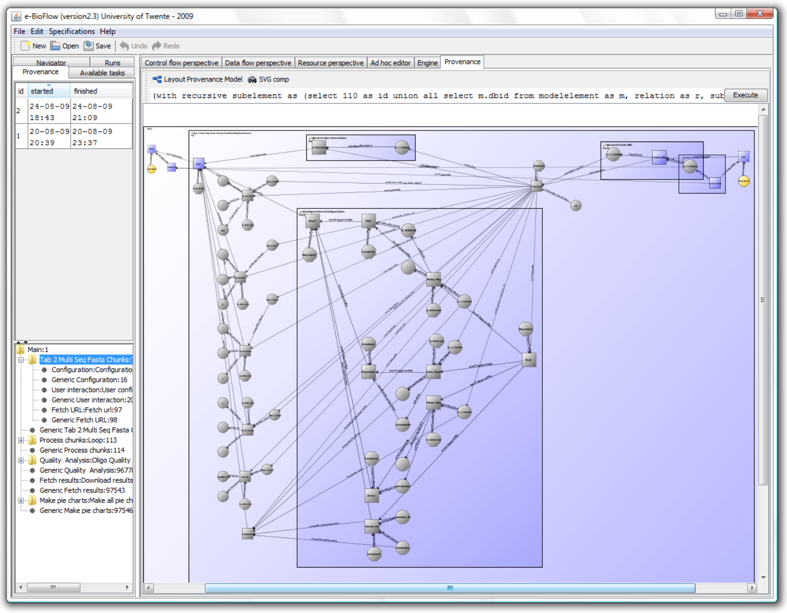
When this perspective is placed in provenance context, only artifacts (data) and processes are shown. The links that exist between processes and artifacts are WasGeneratedBy, Used and WasTriggeredBy. Of these only the WasGeneratedBy and Used relations are shown in this perspective. The WasTriggeredBy relation captures control flow information, connecting two processes, and is therefore hidden. Agents tell something about the service or resource that is being used, nothing about the data flow, and are therefore hidden as well (and any corresponding WasControlledBy relation too). See Figure 5.3(a) for an example representation, and 5.4(b) for a screenshot of the provenance browser in e-BioFlow using this perspective.

Control flow perspective. When designing workflows in e-BioFlow in control flow perspective, processes are shown, input and output ports are hidden. Instead, control flow constructs, such as XOR-, AND- and OR-splits and -joins are shown. Links between processes indicate the control flow between processes.

When this perspective is placed in provenance context, only processes are visible. Agents and artifacts are hidden. The only relation between processes is WasTriggeredBy, and therefore the only visible relation. See Figure 5.3(c) for an example representation, and 5.4(d) for a screenshot of the provenance browser in e-BioFlow using this perspective.

Resource perspective. When designing workflows in e-BioFlow in resource perspective, this is the same view as the control flow perspective, but additionally, roles are visualised. A role of a task determines which actor (service) is assigned during workflow enactment.

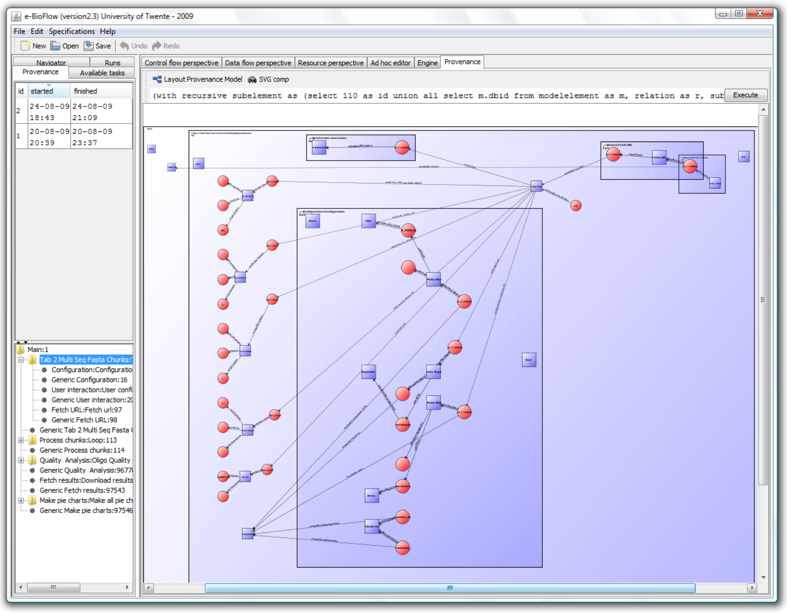
When this perspective is placed in provenance context, the resource perspective is similar to the control flow perspective, but additionally agents (which correspond to services according to the OPM-profile presented in §3.5) are visible. The relation between processes and agents, WasControlledBy, is visible as well. Artifacts, Used and WasGeneratedBy relations are hidden. See Figure 5.3(b) for an example representation, and 5.4(a) for a screenshot of the provenance browser in e-BioFlow using this perspective.


e-BioFlow (version2.3) University of Twente - 2009
File Edit Specifications Help
New Open Save Undo Redo
Navigator Runs
Provenance Available tasks
Control flow perspective Data flow perspective Resource perspective Ad hoc editor Engine Provenance
Layout Provenance Model SVG comp
Execute
Main:1
Tab 2 Multi Seq Fasta Chunks
Configuration:Configuratio
Generic Configuration:16
User interaction:User confi
Generic User interaction:2
Fetch URL:Fetch url:97
Generic Fetch URL:98
Generic Tab 2 Multi Seq Fasta
Process chunks:Loop:113
Generic Process chunks:114
Quality Analysis:Oligo Quality
Generic Quality Analysis:9677
Fetch results:Download results
Generic Fetch results:97543
Make pie charts:Make all pie ch
Generic Make pie charts:97546

```
(with recursive subelement as (select 110 as id union all select m.dbid from modelelement as m, relation as r, sub
```

id	started	finished
2	24-08-09 18:43	24-08-09 21:09
1	20-08-09 20:39	20-08-09 23:37

(a) Normal perspective

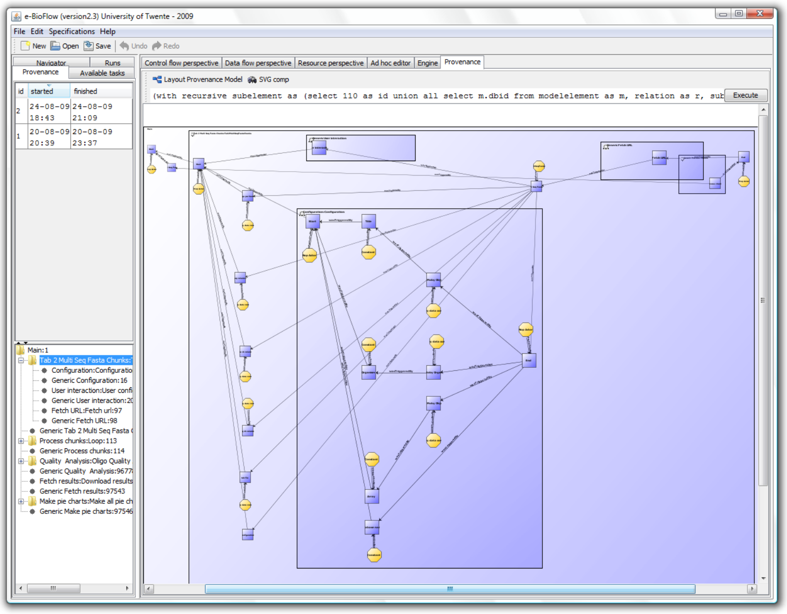

e-BioFlow (version2.3) University of Twente - 2009
File Edit Specifications Help
New Open Save Undo Redo
Navigator Runs
Provenance Available tasks
Control flow perspective Data flow perspective Resource perspective Ad hoc editor Engine Provenance
Layout Provenance Model SVG comp
Execute
Main:1
Tab 2 Multi Seq Fasta Chunks
Configuration:Configuratio
Generic Configuration:16
User interaction:User confi
Generic User interaction:2
Fetch URL:Fetch url:97
Generic Fetch URL:98
Generic Tab 2 Multi Seq Fasta
Process chunks:Loop:113
Generic Process chunks:114
Quality Analysis:Oligo Quality
Generic Quality Analysis:9677
Fetch results:Download results
Generic Fetch results:97543
Make pie charts:Make all pie ch
Generic Make pie charts:97546

```
(with recursive subelement as (select 110 as id union all select m.dbid from modelelement as m, relation as r, sub
```

id	started	finished
2	24-08-09 18:43	24-08-09 21:09
1	20-08-09 20:39	20-08-09 23:37

(b) Data flow perspective

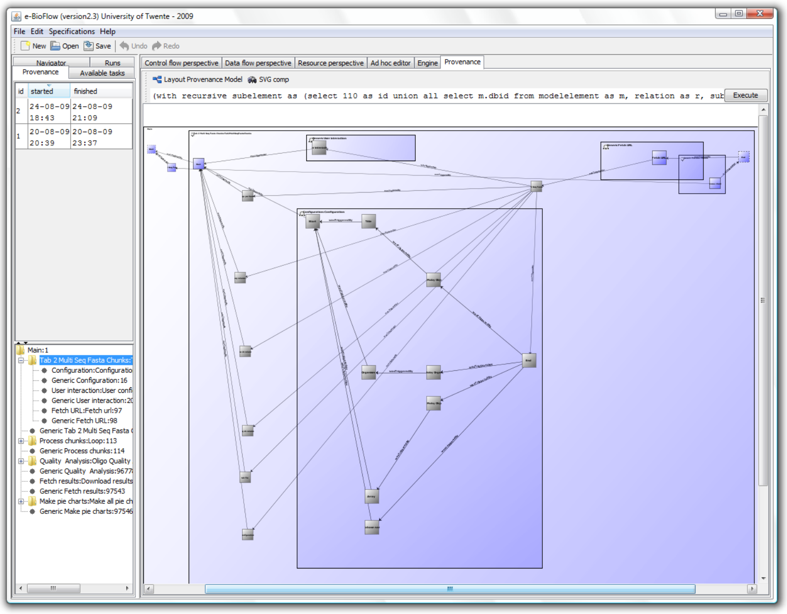
Figure 5.4: Screenshots of perspectives in e-BioFlow of a partial graph retrieved using a recursive query executed by the query panel.


e-BioFlow (version2.3) University of Twente - 2009
File Edit Specifications Help
New Open Save Undo Redo
Navigator Runs
Provenance Available tasks
Control flow perspective Data flow perspective Resource perspective Ad hoc editor Engine Provenance
Layout Provenance Model SVG comp
Execute
Main:1
Tab 2 Multi Seq Fasta Chunks:
Configuration:Configuratio
Generic Configuration:16
User interaction:User confi
Generic User interaction:2
Fetch URL:Fetch url:97
Generic Fetch URL:98
Generic Tab 2 Multi Seq Fasta
Process chunks:Loop:113
Generic Process chunks:114
Quality Analysis:Oligo Quality
Generic Quality Analysis:9677
Fetch results:Download results
Generic Fetch results:97543
Make pie charts:Make all pie ch
Generic Make pie charts:97546

```
(with recursive subelement as (select 110 as id union all select m.dbid from modelelement as m, relation as r, sub
```

id	started	finished
2	24-08-09 18:43	24-08-09 21:09
1	20-08-09 20:39	20-08-09 23:37

(c) Resource perspective


e-BioFlow (version2.3) University of Twente - 2009
File Edit Specifications Help
New Open Save Undo Redo
Navigator Runs
Provenance Available tasks
Control flow perspective Data flow perspective Resource perspective Ad hoc editor Engine Provenance
Layout Provenance Model SVG comp
Execute
Main:1
Tab 2 Multi Seq Fasta Chunks:
Configuration:Configuratio
Generic Configuration:16
User interaction:User confi
Generic User interaction:2
Fetch URL:Fetch url:97
Generic Fetch URL:98
Generic Tab 2 Multi Seq Fasta
Process chunks:Loop:113
Generic Process chunks:114
Quality Analysis:Oligo Quality
Generic Quality Analysis:9677
Fetch results:Download results
Generic Fetch results:97543
Make pie charts:Make all pie ch
Generic Make pie charts:97546

```
(with recursive subelement as (select 110 as id union all select m.dbid from modelelement as m, relation as r, sub
```

id	started	finished
2	24-08-09 18:43	24-08-09 21:09
1	20-08-09 20:39	20-08-09 23:37

(d) Control flow perspective

Figure 5.4: continued.

5.1.3 Query interface

Navigating and browsing through a provenance graph helps in finding results. Being able to search for a specific process, artifact or agent would also be of great advantage while inspecting provenance data. In our use case OligoRAP, it would be very useful to retrieve all generated pie charts of a specific run. Since each pie chart is generated in a subworkflow, for which an account is generated using the OPM-profile, it is not a very hard task to find the corresponding accounts with the Account Navigator.

A drawback of this approach is that the accounts can not be loaded and visualised at the same time in the browser, giving a clear overview of all generated pie charts. Therefore, the provenance browser is extended with a query panel. In the query panel the user can formulate a query in SQL, based on the OPM database implementation as presented in §3.4. A requirement for queries formulated in the query panel is that they should select the id of an OPM element, either from the supertable element itself, or one of its subtables artifact, task or agent. The query is adapted automatically, so that first order neighbours are also retrieved. Further, all the relations of the corresponding elements are loaded into the provenance browser, as well as the accounts to which they belong.

Example queries. Two examples are presented here, retrieving pie charts, and deriving all elements to the root using a recursive query.

Example 1. Using the query panel it becomes possible to visualise all generated pie charts of a single run at once. It is known from the workflow, that the last step in generating the pie chart is an unzip operation, performed by a workflow task named ‘UngzipRole’. If a query is formulated that retrieves these processes for a certain run, and this query is executed by the query panel, their first order neighbours are loaded as well, which are the artifacts. In natural language, the query would read: retrieve all processes named ’UngzipRole’ of run 1. The corresponding query expressed in SQL can be found in Query 5.1. Query A represents the query as formulated by the user, query B is the adapted query that also retrieves the first order neighbours.

The result is a cross section of the provenance graph, consisting of twelve unconnected graphs. Each graph contains both the unzip process and (since first order neighbours are retrieved) the generated pie chart artifact, grouped together by their corresponding account. Twelve pie charts are generated in this OligoRAP run, six for the genome, and six for the transcriptome. The query can be extended to retrieve only the pie

Query 5.1: Query for retrieving SVG pie charts in the query panel.

Query A:

```
    SELECT wgb.cause FROM process AS p, wasgeneratedby AS wgb WHERE
      p.value='UngzipRole' AND wgb.effect=p.id AND p.run=1
```

Query B, neighbours included:

```
    SELECT * FROM element WHERE
        id IN (Query 1) or
        id IN (SELECT cause FROM relation WHERE effect IN (Query 1)) or
        id IN (SELECT effect FROM relation WHERE cause IN (Query 1))
```

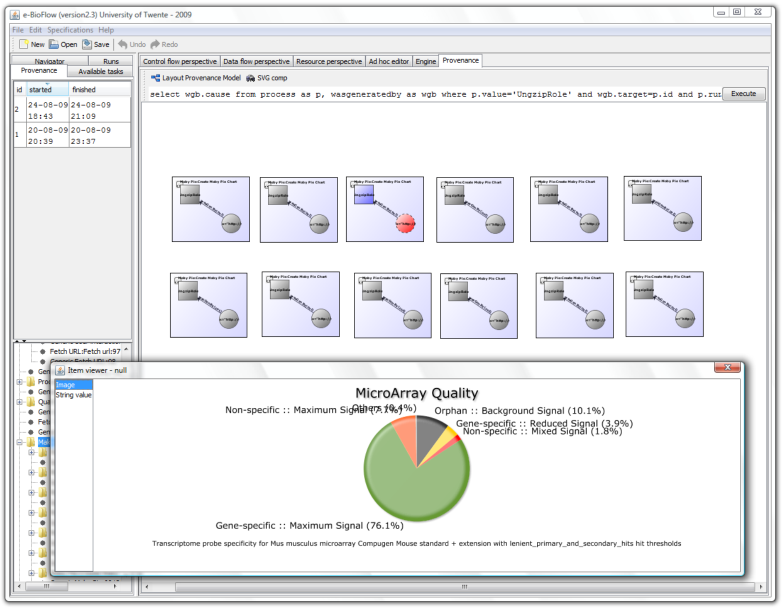
Query C, Transcriptome pie charts only:

```
SELECT a.id FROM process AS p, wasgeneratedby AS wgb,
    artifact AS a, data AS d WHERE
      p.value='UngzipRole' AND wgb.target=p.id AND
        p.run=1 AND a.id=wgb.source AND
          a.dataid=d.id AND d.itemvalue~'Transcriptome'
```

charts of the transcriptome, by matching the word 'Transcriptome' in the datavalue of the artifact of the pie chart. This works, because SVG data is stored as XML, and the word Transcriptome is contained in a string in the SVG picture. See Figure 5.5 for a screenshot of the partial provenance graphs in the provenance browser of e-BioFlow of both queries A (Figure 5.5(a)) and C (Figure 5.5(b).

Example 2. Recursive queries are implemented in PostgreSQL since version 8.4 and prove very valuable in the following example, for which the query in natural language would read: retrieve all elements from which artifact A was derived.

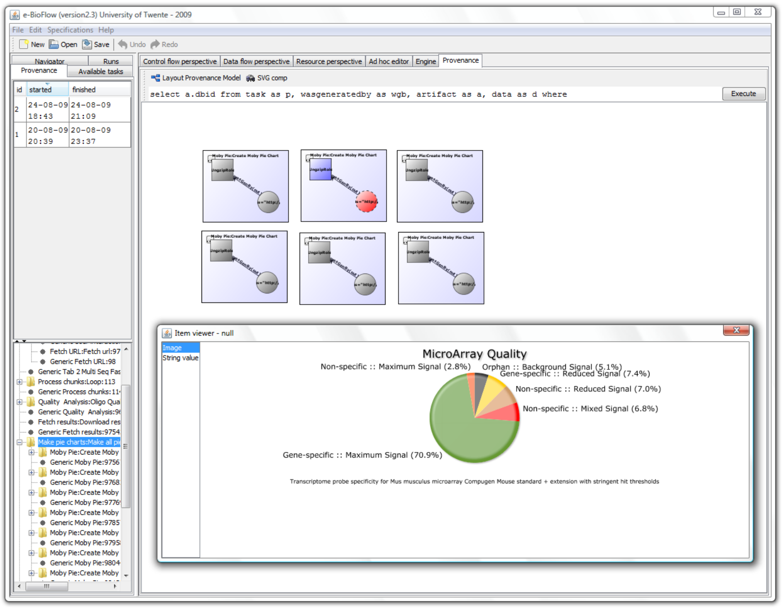
The result of such a query are all the elements leading from artifact A to the first task executed in the workflow. This query expressed in SQL can be found in Query 5.2. The recursive query can be easily adapted to limit the number of retrieved elements to the nth-order neighbour. This is expressed in query B of Query 5.2. A screenshot of the provenance browser of e-BioFlow showing a partial provenance graph retrieved using a recursive query is presented in Figure 5.4, where the partial graph is visualised in four perspectives. The leftmost task is the start task of the workflow, the rightmost task the task queried for (with id 45).


e-BioFlow (version2.3) University of Twente - 2009
File Edit Specifications Help
New Open Save Undo Redo
Navigator Runs
Provenance Available tasks
Control flow perspective
Data flow perspective
Resource perspective
Ad hoc editor
Engine
Provenance
Layout Provenance Model
SVG comp
Execute
Item viewer - null
Image
String value
MicroArray Quality
Non-specific :: Maximum Signal (7.1%)
Others (0.4%)
Orphan :: Background Signal (10.1%)
Gene-specific :: Reduced Signal (3.9%)
Non-specific :: Mixed Signal (1.8%)
Gene-specific :: Maximum Signal (76.1%)
Transcriptome probe specificity for Mus musculus microarray Compugen Mouse standard + extension with lenient_primary_and_secondary_hits hit thresholds

```
select wgb.cause from process as p, wasgeneratedby as wgb where p.value='UngzipRole' and wgb.target=p.id and p.run
```

id	started	finished
2	24-08-09 18:43	24-08-09 21:09
1	20-08-09 20:39	20-08-09 23:37

(a) All pie charts of run 1


e-BioFlow (version2.3) University of Twente - 2009
File Edit Specifications Help
New Open Save Undo Redo
Navigator Runs
Provenance Available tasks
Control flow perspective
Data flow perspective
Resource perspective
Ad hoc editor
Engine
Provenance
Layout Provenance Model
SVG comp
Execute
Item viewer - null
Image
String value
MicroArray Quality
Non-specific :: Maximum Signal (2.8%)
Orphan :: Background Signal (5.1%)
Gene-specific :: Reduced Signal (7.4%)
Non-specific :: Reduced Signal (7.0%)
Non-specific :: Mixed Signal (6.8%)
Gene-specific :: Maximum Signal (70.9%)
Transcriptome probe specificity for Mus musculus microarray Compugen Mouse standard + extension with stringent hit thresholds

```
select a.dbid from task as p, wasgeneratedby as wgb, artifact as a, data as d where
```

id	started	finished
2	24-08-09 18:43	24-08-09 21:09
1	20-08-09 20:39	20-08-09 23:37

(b) Only Transcriptome pie charts of run 1

Figure 5.5: Screenshots of pie charts in e-BioFlow retrieved with the query panel.

Query 5.2: Recursive query for retrieving a derivation trail to the start task of the workflow, from an element identified by ‘45’.

Query A:

```
WITH RECURSIVE subelement AS (

  SELECT 45 AS id

    UNION ALL

  SELECT m.id FROM element AS m, relation AS r, subelement AS sub
    WHERE r.effect = m.id AND r.cause = sub.id
) SELECT * FROM subelement
```

Query B, limited to the 6th-order neighbour:

```
WITH RECURSIVE subelement AS (

  SELECT 45 AS id,1 AS n

    UNION ALL

  SELECT m.id,n+1 AS n FROM element AS m, relation AS r, subelement AS sub
    WHERE r.effect = m.id AND r.cause = sub.id AND n<=6
) SELECT id FROM subelement
```

5.1.4 Loading neighbours

When elements are loaded by the provenance browser, either by selecting an account in the Account Navigator or using the query interface, not all first order neighbours are loaded. Neighbours can belong to an account at a higher level in the RT, or if the query panel was used to load elements using the query interface, only the elements queried for and their first order neighbours are loaded. Only a partial provenance graph is loaded in these cases. Especially the root elements of the partial graph loaded for a particular account of a subworkflow will have neighbours in higher level accounts. By a similar argument, the last tasks and generated artifacts are used by other processes, belonging to other accounts. For these elements, the provenance browser provides the possibility to extend the visual partial graph by loading the neighbours of a specific element.

If a user wants to determine which process generated some particular artifact, and this process is not yet loaded by the provenance browser, the provenance browser can be instructed to load the neighbours of this artifact. Similarly, this action can be performed for processes and agents as well. In the provenance browser, this action can be invoked by right clicking an element. A menu appears where the load neighbours action can be selected. The provenance browser queries the provenance archive for the neighbours of the selected task and adds the newly loaded neighbours to the partial graph. In an iterative approach where the neighbours of the newly added elements are loaded, this will lead eventually to the first task executed.

5.2 Browsing through an OligoRAP run

By starting at the root account and descending the Account Navigator, the user is able to select an interesting account, based on its name that follows from the workflow that was executed. Then, only the elements belonging to this account have to be retrieved and shown instead of the complete graph. This way very large graphs can be navigated, like the graph generated by an OligoRAP run, which consists of over 3×10^5 elements and relations.

If the OPM-profile is used, the hierarchy in the RT and thus the levels of detail provided in the provenance data follow from the hierarchy specified in the workflow. The hierarchy specified in the workflow specification of OligoRAP is intuitive, but designed having provenance navigation in mind: the use of composite tasks and specification of subworkflows was chosen strategically. Each service invocation, including the data transformation steps needed for the specific service, like constructing the XML data items, are all defined in a subworkflow. Further, a single subworkflow is used that provides all configuration items. In addition, the parallel invocation of the processing of all chunks is defined in a subworkflow and the main workflow consists of only three composite tasks.

In §3.6 it was shown that an OligoRAP run generates 5739 accounts and 2869 refinements. All accounts, except the root account, form a refinement pair ($5739 = 2869 \times 2 + 1$). 5738 of these accounts are child accounts (except the root account). 997 accounts are parent accounts. The average number of accounts per child is 5.7, with a standard deviation of 6.8. This high standard deviation is mainly due to the fact that the process chunk subworkflow invokes twelve composite tasks, which results in 24 child accounts for the parent account that executes the composite task.

When browsing and navigating through provenance data this way, it is

μ **childs**	**#instances**	**Composite task name**	**Subworkflow name**
10	1	Main	
6	1	Tab 2 Multi Seq Fasta Chunks	Tab2MultiSeqFastaChunks
24	1	Process chunks	Loop
24	108	Loop	Loop
4	109	Analyse Genome	OligoAnnotationAnalyser
4	109	Analyse UMT	OligoAnnotationAnalyser
2	109	BLAT BLAST GENOME	BLAT and BLAST
10	109	BLAT BLAST UMT	BLAT and BLAST
2	218	BLAT	BLAT
2	109	BLAST	BLAST
2	109	Merge Genome and UMT results	Oligo Merge XML
6	1	Quality Analysis	Oligo Quality Analyser
24	1	Make pie charts	Make all pie charts
2	12	Moby Pie	Create Moby Pie Chart

Table 5.1: Accounts created during an OligoRAP run

desirable for the RT to be deep, varying from a very coarse grained to a very fine grained granularity.

If the RT has a small average branching factor (an account does not have too many child accounts) with little deviation (there does not exist a small set of parent accounts that overlap many others, while others do not overlap any), then the OPM entities are balanced over all accounts, keeping the number of entities per account equally divided.

5.3 Debugging with the Provenance Browser

Services can be (temporarily) unavailable for various reasons, reasons the workflow system unfortunately cannot control. Network problems can occur or high server load can cause time-outs and connection problems; services become of age and are not being properly maintained any more. See the PhD thesis of Wassink [49] for an analysis of the availability of webservices in Taverna workflows present at $^{\text{my}}$Experiment [36]. Although OligoRAP is properly maintained, running OligoRAP was not always a smooth ride. The BioMOBY services used in OligoRAP are designed to be robust, in terms of uptime and load balancing, using a GRID approach. However, when e-BioFlow was submitting more jobs simultaneously than the original Perl implementation has stressed the server with, unforeseen problems were caused. As was mentioned before, the synchronous (BLAT) jobs cannot be performed in parallel, causing a very high server load leading to time-outs. This was solved by not running these services in parallel. A problem encountered with asynchronous services was already shortly addressed in §3.3.2, but is explained here in more detail, showing how the provance browser can be used for debugging purposes.

During the runs with caching enabled, an error occurred: the OligoMergeXML could not parse XML data that was submitted to it. The input of the OligoMergeXML has two XML files as input. Which of these two XML files caused the problem? Was the XML malformed, and if so, which process produced this malformed XML?

Using the provenance browser, the specific OligoMergeXML task could be identified by browsing through the Account navigator, loading the OligoMerge account and navigate the graph until the OligoMergeXML process was found. The artifacts containing (the URL of the) XML data used as input for this task could then be directly examined, which were related to this process by a USED dependency. It turned out the XML of the Genome was incomplete, causing the parse error. But what was the cause for the incomplete XML data? Following the WASGENERATEDBY relations of the artifact that contained the incomplete XML the processes this artifact was generated by were indicated and inspected, and by loading their neighbours (since some of them occurred in higher level accounts) the specific asynchronous OligoAnnotationAnalyser process that created this malicious XML data was found. Although the last poll task indicated the service was finished (which is why the workflow continued), the artifact of the service notes of the Poll process contained an error message: the maximum connection limit to the database was reached, causing a connection error. Since this message is present in the service notes, the error occurred on server side. The asynchronous services, such as the Oligo Annotation Analyser, are designed to be executed all in parallel, so that jobs can be scheduled on a GRID. A GRID manager is responsible for scheduling jobs efficiently, preventing problems such as these.

Since the problem occurs on server side, it can not be solved by the workflow system and was therefore submitted to the maintainers. They responded quickly, and it turned out the GRID manager was ill configured, causing jobs to start at several nodes and appear in multiple queues, causing too many connections to their Ensembl database. After reconfiguration of the GRID manager, OligoRAP was running without encountering any further problems.

5.4 Discussion

The levels of detail that can be addressed in an OPM graph depend on the level of granularity, specified by the RT. Users intuitively use hierarchy whilst designing workflows. The OPM-profile captures the hierarchical structure of the workflow automatically in the provenance data, by defining both a coarse and fine grained account view towards the provenance data of each (sub)workflow.

The RT structure depends on the number of composite tasks and how they are used in the workflow. When a workflow uses loops or iterations, the provenance graph of a single account can still become very large. Good workflow design practice would be to divide loops and iterations in subworkflows, so that the provenance data of the processing of the items itself is captured at a deeper level.

How to use composite tasks is something to consider at workflow design time. Workflow designers use the hierarchy of subworkflows intuitively to keep the specification simple and arranged. This automatically contributes to a finer level of granularity of the generated provenance data.

The provenance browser works in theory with any OPM graph, also with provenance graphs generated by others, yet there is no import function present in e-BioFlow at the moment. The provenance graph visualised by the provenance browser does not necessarily have to be generated using the OPM-profile. To fully benefit of the Account Navigator, and being able to intuitively collapse and expand refinements, the provenance graph should make use of a similar account structure as presented in §5.1.1.

A conclusion is the place where you got tired of thinking.

Harold Fricklestein

Chapter 6

Conclusion

This chapter summarises this thesis by briefly discussing the conclusions of the previous chapters and the most important directions for future work.

6.1 Summary

Several improvements were made to the workflow tool e-BioFlow in order to run the case study OligoRAP, such as the storage of data values in a database back-end, pass data as reference in the workflow engine, the integration of BioMOBY and the data viewer component. All these advancements have made e-BioFlow a far more mature SWfMS. With its intuitive way of designing workflows and easy way of implementing new services, e-BioFlow has evolved from a proof-of-concept into a powerful research prototype. The case study proved that a large-scale data-intensive use case such as OligoRAP can successfully be casted as a workflow in e-BioFlow. By using parallelism already a huge performance improvement was achieved by a factor 2 and more than 3GB of data was generated and processed.

All four provenance challenges mentioned by Davidson and Freire [11] have been addressed: information management infrastructure and information overload, interoperability, analyse and visualise provenance data, connecting database and workflow provenance.

Information management and dealing with information overload is achieved by using e-BioFlow and by implementing the OPM into a relational database management system, persistently storing the provenance data. Provenance data was captured successfully using the OPM-profile and stored in this provenance archive. The stored provenance data can be queried efficiently using queries expressed in SQL. e-BioFlow is now a complete provenance aware scientific workflow management system.

To speed up workflow execution and benefit workflow design, a mechanism is proposed that employs captured provenance data as cache and the proposed caching scheme for deterministic atomic tasks was successfully applied in the case study. For the end-user, caching tasks is easy as marking them deterministic.

Queries are defined for the implemented provenance archive to determine whether a task execution already exists in cache and retrieve the output of such a task if found. The queries have been optimised by rewriting them, and by creating strategic indices in the database. This has resulted in a performance improvement of 19% for an OligoRAP run where BLAT and Download tasks were cached.

Davidson and Freire state that information management systems are notoriously hard to use, therefore the usability aspect is of paramount importance. Since the structure of the provenance archive follows directly from the structure of the workflow if the OPM-profile is used, the provenance browser forms an intuitive interface for end-users to analyse and manage large amounts of provenance data.

Interoperability is improved by defining an OPM-profile, a profile to use the OPM for hierarchical SWfMS. This profile preserves the hierarchy of composite tasks (subworkflows) in the generated provenance data by means of overlapping accounts and refinements. The proposed profile is used in the provenance implementation of e-BioFlow to collect provenance data during workflow execution.

Analysing and visualising large provenance graphs is achieved in e-BioFlow by introducing a provenance browser that uses the graph structure of the OPM as primary visualisation. In addition, instead of representing a complete provenance graph which is hard to navigate, four additional techniques are designed that helps the user navigate and browse through large provenance graphs. These techniques are:

- An account navigator that uses the hierarchy captured by the OPM-profile using composite tasks and subworkflows to visualise a tree structure of generic and detailed views towards the provenance data. By descending the RT, a tree of accounts that defines multiple levels of detail in the OPM, starting at the root account and expanding refinements when needed. The OPM-profile facilitates the generation of an RT that has a tree structure optimised for navigating a provenance graph.

- The provenance browser can use several perspectives towards provenance data, namely data flow, control flow and resource perspectives, similar to the perspectives used towards workflows in e-BioFlow. This enables the end-user to show detail on demand.

- A query panel that enables the end-user to specify a provenance query. The result of the query is directly visualised in the provenance browser, allowing the end-user to query for certain data items, tasks or even complete derivation trails.

- Retrieve tasks or data items that are not yet loaded by the provenance browser, but are neighbours of currently visible tasks or data items. This expands the current graph presented in the provenance browser, enabling the end-user to start searching at some point, and determine how some data item was derived.

The techniques proved to be very useful whilst debugging OligoRAP: error messages, and more interestingly, their cause, can be easily identified using the provenance browser, and the provenance archive can be queried using the query panel for all generated pie charts, presenting a clear overview of the result of the run.

Connecting database and workflow provenance is not tackled in the sense that workflow and databases are treated uniformly across several applications. However, by implementing the OPM into a relational database and using the OPM-profile, the way in which provenance data is stored in the database is known. This provenance data can be queried and reused by other database applications or workflow systems. The OPM implementation and OPM-profile can serve as the framework in which database operators and workflow modules can be treated uniformly.

6.2 Future Work

Provenance queries formulated for this OPM implementation and profile are unambiguous. It would be a good thing to have a common OPM-profile standard for SWfMS. Combining the advantages of such a standard OPM-profile, the OPM, cache and the provenance browser makes it possible to exchange and interpret provenance data of experiments executed by other SWfMS, and their results can be reused and analysed as well.

A query language for the OPM would benefit reasoning over provenance data and would greatly improve interoperability between provenance aware systems. The OPM implementation provided here can be used as such. The SQL query language is Turing-complete since the latest SQL standards that support recursion, and probably most provenance queries can be expressed with it. However, the queries can become very large and hard to interpret. Davidson et al. [10] argue for a high level query language for searching and visualising provenance data using concepts that are familiar to the end-user. The provenance browser already presents results in an intuitive manner, a mapping from a high level query language to the OPM implementation presented here can provide a solution for intuitive searching. In addition, if several provenance aware systems adopt the same query language, these systems can be compared to each other. One system might perform better than another. A provenance benchmark can be devised, defining a proper provenance data set and a diversity of queries.

A major interoperability challenge that remains for the OPM, is how to serialise a graph, and especially the values of artifacts, such that it can be interpreted by others. This became clear during my visit to the 3rd provenance challenge held in Amsterdam, and still after this challenge, it remains an open research issue. Teams were able to generate XML serialisations, but the values were represented differently, even though the same XML schema was used. Reasoning over imported data is still a challenge. An agreement about a set of standard common data types should be used, however, if (bio)informaticians are allowed to use their own data types, they will.

The OPM or the OPM-profile can be extended with a standard way of defining errors that occur during execution. Errors can be modelled as artifacts, or as tasks. The provenance browser can use errors as a starting point, which helps during the debugging of a workflow or service, such as the problem described in §5.3.

In this research, the structure of the OPM was used to efficiently and effectively query provenance data. By storing additional data the retrieval of previous tasks executions can probably be optimised even further, with the drawback of some redundancy. A simple example of such would be to store the number of input and output ports in the PROCESS table directly, instead of querying them. A more complex but probably more efficient way of querying would be to calculate a hash value over all the input and output ports, input data, task and actor values and store this hash in the PROCESS table. The current phase 1 query can then be extended using this hashvalue, improving its performance even more.

During the development of the OPM implementation and discussing its use, the idea for a new way of designing workflows was born, called ad-hoc workflow design [50]. Instead of the traditional workflow life cycle, where a workflow is first composed, then mapped to resources, then executed and finally results inspected using provenance, an iterative approach is taken. During design a single task is executed. Based on the output of this task, a next step or task in the workflow is selected. This task can then in turn be executed, creating a workflow specification on-the-fly. Intermediate results can be fetched directly from the provenance archive. This iterative design approach brings together the design, execution and provenance cycle of a workflow.

Appendices

Appendix A

Queries

Query A.1: Cache query for MobyBlat in query form 1. Ineffective for caching purposes due to long execution time.

```
SELECT
  t.id FROM process AS t ,
  used AS u1, artifact AS a1, dcompare AS d1,
  used AS u2, artifact AS a2, dcompare AS d2,
  used AS u3, artifact AS a3, dcompare AS d3,
  used AS u4, artifact AS a4, dcompare AS d4,

  wasgeneratedby AS wgb1,
  wasgeneratedby AS wgb2,
  wasgeneratedby AS wgb3,
  wasgeneratedby AS wgb4,
  wasgeneratedby AS wgb5,

  wascontrolledby AS wcb,
  agent AS a
WHERE t.value='MobyBlat' AND
  t.id=u1.cause AND u1.role='user' AND u1.effect=a.id AND
    a1.dataid=d1.d1id AND d1.d2id='48966' AND
  t.id=u2.cause AND u2.role='q' AND u2.effect=a.id AND
    a2.dataid=d2.d1id AND d2.d2id='48984' AND
  t.id=u3.cause AND u3.role='out' AND u3.effect=a.id AND
    a3.dataid=d3.d1id AND d3.d2id='48985' AND
  t.id=u4.cause AND u4.role='input' AND u4.effect=a.id AND
    a4.dataid=d4.d1id AND d4.d2id='48951' AND

  t.id=wgb1.effect AND wgb1.role='user' AND
  t.id=wgb2.effect AND wgb2.role='serviceNotes' AND
  t.id=wgb3.effect AND wgb3.role='result' AND
  t.id=wgb4.effect AND wgb4.role='output' AND
  t.id=wgb5.effect AND wgb5.role='no_hits' AND
  t.id=wcb.effect AND wcb.cause=a.id AND

  wcb.role='MobyBlat' AND a.value='MobyBlat'
  AND NOT t.finished is NULL AND
  (SELECT count(u.id) FROM used AS u WHERE u.cause=t.id)=4 AND
  (SELECT count(wgb.id) FROM wasgeneratedby AS wgb WHERE wgb.effect=t.id)=5
LIMIT 1
```

Query A.2: Cache query for MobyBlat in query form 2

```
SELECT t.id
FROM process AS t WHERE
  t.value='MobyBlat' AND
  t.id IN (SELECT u.cause FROM used AS u WHERE
    u.role='user' AND u.effect IN (
      SELECT a.id FROM artifact AS a,dcompare AS d WHERE
        a.dataid=d.d1id AND d.d2id='48966') ) AND
  t.id IN (SELECT u.cause FROM used AS u WHERE
    u.role='q' AND u.effect IN (
      SELECT a.id FROM artifact AS a,dcompare AS d WHERE
        a.dataid=d.d1id AND d.d2id='48984') ) AND
  t.id IN (SELECT u.cause FROM used AS u WHERE
    u.role='out' AND u.effect IN (
      SELECT a.id FROM artifact AS a,dcompare AS d WHERE
        a.dataid=d.d1id AND d.d2id='48985') ) AND
  t.id IN (SELECT u.cause FROM used AS u WHERE
    u.role='input' AND u.effect IN (
      SELECT a.id FROM artifact AS a,dcompare AS d WHERE
        a.dataid=d.d1id AND d.d2id='48951') ) AND

  t.id IN (
    SELECT wgb.effect FROM wasgeneratedby AS wgb WHERE
      wgb.role='user') AND
  t.id IN (
    SELECT wgb.effect FROM wasgeneratedby AS wgb WHERE
      wgb.role='serviceNotes') AND
  t.id IN (
    SELECT wgb.effect FROM wasgeneratedby AS wgb WHERE
      wgb.role='result') AND
  t.id IN (
    SELECT wgb.effect FROM wasgeneratedby AS wgb WHERE
      wgb.role='output') AND
  t.id IN (
    SELECT wgb.effect FROM wasgeneratedby AS wgb WHERE
      wgb.role='no_hits') AND

  t.id IN (
    SELECT wcb.effect FROM wascontrolledby AS wcb, agent AS a WHERE
      wcb.cause=a.id AND wcb.role='MobyBlat' AND a.value='MobyBlat') AND

  NOT t.finished is NULL AND
  (SELECT count(u.id) FROM used AS u WHERE u.cause=t.id)=4 AND
  (SELECT count(wgb.id) FROM wasgeneratedby AS wgb WHERE wgb.effect=t.id)=5

LIMIT 1
```

Query A.3: Cache query for MobyBlat in query form 3.

```
SELECT t.id,
  (SELECT count(u.id) FROM used AS u WHERE
    u.cause=t.id) AS inputport,
  (SELECT count(wgb.id) FROM wasgeneratedby AS wgb WHERE
    wgb.effect=t.id) AS outputport
WHERE t.value='MobyBlat' AND

  t.id IN (SELECT u.cause FROM used AS u WHERE
    u.role='user' AND u.effect IN (
      SELECT a.id FROM artifact AS a,dcompare AS d WHERE
        a.dataid=d.d1id AND d.d2id='48966') ) AND
  t.id IN (SELECT u.cause FROM used AS u WHERE
    u.role='q' AND u.effect IN (
      SELECT a.id FROM artifact AS a,dcompare AS d WHERE
        a.dataid=d.d1id AND d.d2id='48984') ) AND
  t.id IN (SELECT u.cause FROM used AS u WHERE
    u.role='out' AND u.effect IN (
      SELECT a.id FROM artifact AS a,dcompare AS d WHERE
        a.dataid=d.d1id AND d.d2id='48985') ) AND
  t.id IN (SELECT u.cause FROM used AS u WHERE
    u.role='input' AND u.effect IN (
      SELECT a.id FROM artifact AS a,dcompare AS d WHERE
        a.dataid=d.d1id AND d.d2id='48951') ) AND

  t.id IN (
    SELECT wgb.effect FROM wasgeneratedby AS wgb WHERE
      wgb.role='user') AND
  t.id IN (
    SELECT wgb.effect FROM wasgeneratedby AS wgb WHERE
      wgb.role='serviceNotes') AND
  t.id IN (
    SELECT wgb.effect FROM wasgeneratedby AS wgb WHERE
      wgb.role='result') AND
  t.id IN (
    SELECT wgb.effect FROM wasgeneratedby AS wgb WHERE
      wgb.role='output') AND
  t.id IN (
    SELECT wgb.effect FROM wasgeneratedby AS wgb WHERE
      wgb.role='no_hits') AND

  t.id IN (
    SELECT wcb.effect FROM wascontrolledby AS wcb, agent AS a WHERE
      wcb.cause=a.id AND wcb.role='MobyBlat' AND a.value='MobyBlat') AND

  NOT t.finished is NULL
```

Query A.4: Cache query for Download url process.

```
SELECT
  (SELECT count(u.id) FROM used AS u WHERE
    u.cause=t.id) AS inputport,
  (SELECT count(wgb.id) FROM wasgeneratedby AS wgb WHERE
    wgb.effect=t.id) AS outputport,t.id
FROM process AS t
WHERE t.value='Download url' AND
  t.id IN (SELECT u.cause FROM used AS u WHERE
    u.role='url' AND u.effect IN (
      SELECT a.id FROM artifact AS a,dcompare AS d WHERE
        a.dataid=d.d1id AND d.d2id='82971') ) AND

  t.id IN (
    SELECT wgb.effect FROM wasgeneratedby AS wgb WHERE
      wgb.role='content') AND

  t.id IN (
    SELECT wcb.effect FROM wascontrolledby AS wcb, agent AS a WHERE
      wcb.cause=a.id AND wcb.role='ScriptINg Role' AND
        a.value='Perl agent localhost') AND

NOT t.finished is NULL
```

Appendix B

Query plans

- Parts in the query plans that are costly or inefficient are colored red and in bold font (like sequential scans). These lines are marked using the symbol $\times$.
- Parts that are improved in respect to a previous query plan are colored green and underlined (like index scans that increase performance). These lines are marked using the symbol $\surd$.

Query plan B.1: Query plan for BLAT cache query A.1 in query form 1, datacompare view 2 (completeitemindex).

```
limit (cost=44000.69..52096.96 rows=1 width=4)
 -> Nested Loop (cost=44000.69..52096.96 rows=1 width=4)
  Join Filter: ((t.id=u1.cause) and (wcb.cause=u1.effect))
  -> Nested Loop (cost=0.00..5232.57 rows=1 width=12)
   ->  Seq Scan on wascontrolledby wcb (cost=0.00..1262.74 rows=1 width=8)          ×
    Filter: (role='MobyBlat'::text)
   -> index Scan using process_pkey on process t (cost=0.00..3969.82 rows=1 width=4)
    index Cond: (t.id=wcb.effect)
    Filter: ((not (t.finished is null)) and (t.value='MobyBlat'::text)
      and ((SubPlan 1)=4) and ((SubPlan 2)=5))
    SubPlan 1
     -> Aggregate (cost=1936.91..1936.92 rows=1 width=4)
      ->  Seq Scan on used u (cost=0.00..1936.90 rows=2 width=4)          ×
       Filter: (cause=$0)
    SubPlan 2
     -> Aggregate (cost=2024.61..2024.62 rows=1 width=4)
      ->  Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=2 width=4)          ×
       Filter: (effect=$0)
  -> Hash Join (cost=44000.69..46864.20 rows=13 width=56)
   Hash Cond: (a4.dataid=d1.id)
   ->  Seq Scan on artifact a4 (cost=0.00..2479.00 rows=102500 width=4)          ×
   -> Hash (cost=44000.51..44000.51 rows=15 width=60)
    -> Hash Join (cost=37024.37..44000.51 rows=15 width=60)
     Hash Cond: (a2.dataid=d1.id)
     -> Hash Join (cost=36863.90..43836.70 rows=426 width=64)
      Hash Cond: (a3.dataid=d1.id)
      -> Hash Join (cost=36703.44..43579.07 rows=12387 width=68)
       Hash Cond: (u1.effect=a.id)
       -> Nested Loop (cost=35449.34..41816.72 rows=102500 width=64)
        -> Hash Join (cost=35449.34..38312.72 rows=1 width=60)
         Hash Cond: (a1.dataid=d1.id)
         ->  Seq Scan on artifact a1 (cost=0.00..2479.00 rows=102500 width=4)          ×
         -> Hash (cost=35449.32..35449.32 rows=1 width=64)
          -> Merge Join (cost=18592.56..35449.32 rows=1 width=64)
           Merge Cond: (u1.cause=u2.cause)
           Join Filter: (u2.effect=u1.effect)
           -> Nested Loop (cost=10510.62..27111.05 rows=102500 width=44)
            -> Merge Join (cost=10510.62..23607.05 rows=1 width=40)
             Merge Cond: (u1.cause=u4.cause)
             Join Filter: (u4.effect=u1.effect)
             -> Nested Loop (cost=6337.27..19422.25 rows=3927 width=28)
              -> Merge Join (cost=6332.79..19271.60 rows=1 width=24)
               Merge Cond: (u1.cause=u3.cause)
               Join Filter: (u3.effect=u1.effect)
               -> Nested Loop (cost=4385.13..15801.65 rows=569415 width=16)
                -> Merge Join (cost=4269.82..4298.05 rows=145 width=12)
                 Merge Cond: (wgb1.effect=u1.cause)
                 -> Sort (cost=2249.94..2259.42 rows=3791 width=4)
                  Sort Key: wgb1.effect
                  ->  Seq Scan on wasgeneratedby wgb1 (cost=0.00..2024.60 rows=3791 width=4)          ×
                   Filter: (role='user'::text)
                 -> Sort (cost=2019.88..2023.79 rows=1564 width=8)
                  Sort Key: u1.cause
                  ->  Seq Scan on used u1 (cost=0.00..1936.90 rows=1564 width=8)          ×
                   Filter: (role='user'::text)
                -> Materialize (cost=115.31..154.58 rows=3927 width=4)
                 -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
                  -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
                   index Cond: (id=48966)
                  -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
                   Recheck Cond: (d1.itemvalue=d2.itemvalue)
                   ->  Bitmap index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)          √
                    index Cond: (d1.itemvalue=d2.itemvalue)
               -> Sort (cost=1947.66..1948.33 rows=267 width=8)
                Sort Key: u3.cause
                ->  Seq Scan on used u3 (cost=0.00..1936.90 rows=267 width=8)          ×
                 Filter: (role='out'::text)
              -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
               -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
                index Cond: (id=48951)
               -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
                Recheck Cond: (d1.itemvalue=d2.itemvalue)
                ->  Bitmap index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)          √
                 index Cond: (d1.itemvalue=d2.itemvalue)
             -> Sort (cost=4173.35..4173.94 rows=238 width=12)
              Sort Key: u4.cause
              -> Hash Join (cost=2141.88..4163.95 rows=238 width=12)
               Hash Cond: (u4.cause=wgb2.effect)
               ->  Seq Scan on used u4 (cost=0.00..1936.90 rows=1035 width=8)          ×
                Filter: (role='input'::text)
              -> Hash (cost=2024.60..2024.60 rows=9382 width=4)
               ->  Seq Scan on wasgeneratedby wgb2 (cost=0.00..2024.60 rows=9382 width=4)      ×
                Filter: (role='servicenotes'::text)
           ->  Seq Scan on artifact a3 (cost=0.00..2479.00 rows=102500 width=4)               ×
          -> Sort (cost=8081.95..8081.95 rows=1 width=20)
           Sort Key: u2.cause
           -> Nested Loop (cost=4004.03..8081.94 rows=1 width=20)
            Join Filter: (u2.cause=wgb4.effect)
            -> Hash Join (cost=4004.03..6035.27 rows=1 width=16)
             Hash Cond: (wgb3.effect=u2.cause)
             ->  Seq Scan on wasgeneratedby wgb3 (cost=0.00..2024.60 rows=1768 width=4)       ×
              Filter: (role='result'::text)
             -> Hash (cost=4003.96..4003.96 rows=6 width=12)
              -> Hash Join (cost=1940.24..4003.96 rows=6 width=12)
               Hash Cond: (wgb5.effect=u2.cause)
               ->  Seq Scan on wasgeneratedby wgb5 (cost=0.00..2024.60 rows=868 width=4)      ×
                Filter: (role='no_hits'::text)
               -> Hash (cost=1936.90..1936.90 rows=267 width=8)
                ->  Seq Scan on used u2 (cost=0.00..1936.90 rows=267 width=8)                 ×
                 Filter: (role='q'::text)
            ->  Seq Scan on wasgeneratedby wgb4 (cost=0.00..2024.60 rows=1765 width=4)        ×
             Filter: (wgb4.role='output'::text)
       ->  Seq Scan on artifact a2 (cost=0.00..2479.00 rows=102500 width=4)                   ×
      -> Hash (cost=1251.74..1251.74 rows=189 width=4)
       ->  Seq Scan on agent a (cost=0.00..1251.74 rows=189 width=4)                          ×
        Filter: (role='MobyBlat'::text)
     -> Hash (cost=111.38..111.38 rows=3927 width=4)
      -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
       -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
        index Cond: (id=48985)
       -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
        Recheck Cond: (d1.itemvalue=d2.itemvalue)
        ->  Bitmap index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)          √
         index Cond: (d1.itemvalue=d2.itemvalue)
    -> Hash (cost=111.38..111.38 rows=3927 width=4)
     -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
      -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
       index Cond: (id=48984)
      -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
       Recheck Cond: (d1.itemvalue=d2.itemvalue)
       ->  Bitmap index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)           √
        index Cond: (d1.itemvalue=d2.itemvalue)
```

Query plan B.2: Query plan for BLAT cache query A.2 in query form 2, datacompare view 2 (completeitemindex)

```
limit (cost=14533.79..2608246.24 rows=1 width=4)
 -> Nested Loop Semi Join (cost=14533.79..2608246.24 rows=1 width=4)
  Join Filter: (t.id=wcb.effect)
  -> Nested Loop Semi Join (cost=14533.79..2606975.20 rows=1 width=40)
   Join Filter: (t.id=u.cause)
   -> Nested Loop Semi Join (cost=11046.22..2601520.98 rows=1 width=36)
    Join Filter: (t.id=wgb.effect)
    -> Nested Loop Semi Join (cost=11046.22..2599448.99 rows=1 width=32)
     Join Filter: (t.id=wgb.effect)
     -> Nested Loop Semi Join (cost=11046.22..2597307.12 rows=1 width=28)
      Join Filter: (t.id=wgb.effect)
      -> Nested Loop Semi Join (cost=11046.22..2595260.42 rows=1 width=24)
       Join Filter: (t.id=wgb.effect)
       -> Nested Loop Semi Join (cost=11046.22..2593213.75 rows=1 width=20)
        Join Filter: (t.id=wgb.effect)
        -> Nested Loop Semi Join (cost=11046.22..2591178.30 rows=1 width=16)
         Join Filter: (u.effect=a.id)
         -> Nested Loop Semi Join (cost=10885.76..2587690.74 rows=1 width=20)
          Join Filter: (t.id=u.cause)
          -> Nested Loop (cost=5453.79..2580317.50 rows=1 width=8)
           -> HashAggregate (cost=5453.79..5460.28 rows=649 width=4)
            -> Hash Semi Join (cost=3487.56..5449.88 rows=1564 width=4)
             Hash Cond: (u.effect=a.id)
             -> Seq Scan on used u (cost=0.00..1936.90 rows=1564 width=8)
              Filter: (role='user'::text)
             -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
              -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
               Hash Cond: (a.dataid=d1.id)
               -> Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)
               -> Hash (cost=111.38..111.38 rows=3927 width=4)
                -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
                 -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
                  index Cond: (id=48966)
                 -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
                  Recheck Cond: (d1.itemvalue=d2.itemvalue)
                  ->  Bitmap index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)      ✓
                   index Cond: (d1.itemvalue=d2.itemvalue)
           -> index Scan using process_pkey on process t (cost=0.00..3967.41 rows=1 width=4)
            index Cond: (t.id=u.cause)
            Filter: ((not (t.finished is null)) and (t.value='MobyBlat'::text) and
                               ((SubPlan 1)=4) and ((SubPlan 2)=5))                                    ×
            SubPlan 1
             -> Aggregate (cost=1936.91..1936.92 rows=1 width=4)
              ->  Seq Scan on used u (cost=0.00..1936.90 rows=2 width=4)                               ×
               Filter: (cause=$0)
            SubPlan 2
             -> Aggregate (cost=2024.61..2024.62 rows=1 width=4)
              ->  Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=2 width=4)                  ×
               Filter: (effect=$0)
          -> Hash Join (cost=5431.97..7369.90 rows=267 width=12)
           Hash Cond: (u.cause=u.cause)
           -> Seq Scan on used u (cost=0.00..1936.90 rows=267 width=8)
            Filter: (role='out'::text)
           -> Hash (cost=5430.58..5430.58 rows=111 width=4)
            -> HashAggregate (cost=5429.47..5430.58 rows=111 width=4)
             -> Hash Semi Join (cost=3487.56..5428.80 rows=267 width=4)
              Hash Cond: (u.effect=a.id)
              -> Seq Scan on used u (cost=0.00..1936.90 rows=267 width=8)
               Filter: (role='q'::text)
              -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
               -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
                Hash Cond: (a.dataid=d1.id)
                -> Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)
                -> Hash (cost=111.38..111.38 rows=3927 width=4)
                 -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
                  -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
                   index Cond: (id=48984)
                  -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
                   Recheck Cond: (d1.itemvalue=d2.itemvalue)
                   ->  Bitmap index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)     ✓
                    index Cond: (d1.itemvalue=d2.itemvalue)
         -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
          Hash Cond: (a.dataid=d1.id)
          -> Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)
          -> Hash (cost=111.38..111.38 rows=3927 width=4)
           -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
            -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
             index Cond: (id=48985)
            -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
             Recheck Cond: (d1.itemvalue=d2.itemvalue)
            ->  Bitmap index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)      √
             index Cond: (d1.itemvalue=d2.itemvalue)
       -> Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=868 width=4)
        Filter: (wgb.role='no_hits'::text)
      -> Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=1765 width=4)
       Filter: (wgb.role='output'::text)
     -> Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=1768 width=4)
      Filter: (wgb.role='result'::text)
    -> Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=9382 width=4)
     Filter: (wgb.role='servicenotes'::text)
   -> Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=3791 width=4)
    Filter: (wgb.role='user'::text)
  -> Hash Semi Join (cost=3487.56..5441.28 rows=1035 width=4)
   Hash Cond: (u.effect=a.id)
   -> Seq Scan on used u (cost=0.00..1936.90 rows=1035 width=8)
    Filter: (role='input'::text)
   -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
    -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
     Hash Cond: (a.dataid=d1.id)
     -> Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)
     -> Hash (cost=111.38..111.38 rows=3927 width=4)
      -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
       -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
        index Cond: (id=48951)
       -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
        Recheck Cond: (d1.itemvalue=d2.itemvalue)
        ->  Bitmap index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)          √
         index Cond: (d1.itemvalue=d2.itemvalue)
 -> Nested Loop (cost=0.00..1271.03 rows=1 width=4)
  -> Seq Scan on wascontrolledby wcb (cost=0.00..1262.74 rows=1 width=8)
   Filter: (role='MobyBlat'::text)
  -> index Scan using agent_pkey on agent a (cost=0.00..8.28 rows=1 width=4)
   index Cond: (a.id=wcb.cause)
   Filter: (a.value='MobyBlat'::text)
```

Query plan B.3: Query plan for BLAT cache query A.3 in query form 3, datacompare view 2 (completeitemindex)

```
Hash Semi Join (cost=33411.50..38846.73 rows=1 width=4)
 Hash Cond: (t.id=wcb.effect)
 -> Hash Join (cost=32140.46..33613.64 rows=192 width=40)
  Hash Cond: (t.id=wgb.effect)
  -> Hash Join (cost=30065.90..31536.21 rows=192 width=36)
   Hash Cond: (t.id=u.cause)
   -> Hash Join (cost=24612.36..26080.02 rows=192 width=32)
    Hash Cond: (t.id=wgb.effect)
    -> Hash Join (cost=22464.13..23928.44 rows=192 width=28)
     Hash Cond: (t.id=wgb.effect)
     -> Hash Join (cost=20416.24..21877.90 rows=192 width=24)
      Hash Cond: (t.id=wgb.effect)
      -> Hash Join (cost=18368.37..19827.39 rows=192 width=20)
       Hash Cond: (t.id=wgb.effect)
       -> Hash Join (cost=16332.33..17788.71 rows=192 width=16)
        Hash Cond: (t.id=u.cause)
        -> Hash Join (cost=10900.36..12354.10 rows=192 width=12)
         Hash Cond: (t.id=u.cause)
         -> Hash Join (cost=5468.39..6919.49 rows=192 width=8)
          Hash Cond: (t.id=u.cause)
          ->  Seq Scan on process t (cost=0.00..1448.46 rows=192 width=4)          ×
           Filter: ((not (finished is null)) and (role='MobyBlat'::text))
          -> Hash (cost=5460.28..5460.28 rows=649 width=4)
           -> HashAggregate (cost=5453.79..5460.28 rows=649 width=4)
            -> Hash Semi Join (cost=3487.56..5449.88 rows=1564 width=4)
             Hash Cond: (u.effect=a.id)
             ->  Seq Scan on used u (cost=0.00..1936.90 rows=1564 width=8)          ×
              Filter: (role='user'::text)
             -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
              -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
               Hash Cond: (a.dataid=d1.id)
               ->  Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)          ×
               -> Hash (cost=111.38..111.38 rows=3927 width=4)
                -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
                 -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
                  index Cond: (id=48966)
                 -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
                  Recheck Cond: (d1.itemvalue=d2.itemvalue)
                  ->  Bitmap index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)          √
                   index Cond: (d1.itemvalue=d2.itemvalue)
         -> Hash (cost=5430.58..5430.58 rows=111 width=4)
          -> HashAggregate (cost=5429.47..5430.58 rows=111 width=4)
           -> Hash Semi Join (cost=3487.56..5428.80 rows=267 width=4)
            Hash Cond: (u.effect=a.id)
            ->  Seq Scan on used u (cost=0.00..1936.90 rows=267 width=8)          ×
             Filter: (role='out'::text)
            -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
             -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
              Hash Cond: (a.dataid=d1.id)
              ->  Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)          ×
              -> Hash (cost=111.38..111.38 rows=3927 width=4)
               -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
                -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
                 index Cond: (id=48985)
                -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
                 Recheck Cond: (d1.itemvalue=d2.itemvalue)
                 ->  Bitmap index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)          √
                  index Cond: (d1.itemvalue=d2.itemvalue)
        -> Hash (cost=5430.58..5430.58 rows=111 width=4)
         -> HashAggregate (cost=5429.47..5430.58 rows=111 width=4)
          -> Hash Semi Join (cost=3487.56..5428.80 rows=267 width=4)
           Hash Cond: (u.effect=a.id)
           ->  Seq Scan on used u (cost=0.00..1936.90 rows=267 width=8)          ×
            Filter: (role='q'::text)
           -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
            -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
             Hash Cond: (a.dataid=d1.id)
             ->  Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)          ×
             -> Hash (cost=111.38..111.38 rows=3927 width=4)
              -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
               -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
                index Cond: (id=48984)
               -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
                Recheck Cond: (d1.itemvalue=d2.itemvalue)
                ->  Bitmap index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)          √
                 index Cond: (d1.itemvalue=d2.itemvalue)
       -> Hash (cost=2030.89..2030.89 rows=412 width=4)
        -> HashAggregate (cost=2026.77..2030.89 rows=412 width=4)
         ->  Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=868 width=4)          ×
          Filter: (role='no_hits'::text)
     -> Hash (cost=2037.39..2037.39 rows=838 width=4)
      -> HashAggregate (cost=2029.01..2037.39 rows=838 width=4)
       ->  Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=1765 width=4)     ✗
        Filter: (role='output'::text)
    -> Hash (cost=2037.41..2037.41 rows=839 width=4)
     -> HashAggregate (cost=2029.02..2037.41 rows=839 width=4)
      ->  Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=1768 width=4)      ✗
       Filter: (role='result'::text)
   -> Hash (cost=2092.58..2092.58 rows=4452 width=4)
    -> HashAggregate (cost=2048.06..2092.58 rows=4452 width=4)
     ->  Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=9382 width=4)       ✗
      Filter: (role='servicenotes'::text)
  -> Hash (cost=5448.17..5448.17 rows=430 width=4)
   -> HashAggregate (cost=5443.87..5448.17 rows=430 width=4)
    -> Hash Semi Join (cost=3487.56..5441.28 rows=1035 width=4)
     Hash Cond: (u.effect=a.id)
     ->  Seq Scan on used u (cost=0.00..1936.90 rows=1035 width=8)                    ✗
      Filter: (role='input'::text)
     -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
      -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
       Hash Cond: (a.dataid=d1.id)
       ->  Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)           ✗
       -> Hash (cost=111.38..111.38 rows=3927 width=4)
        -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
         -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
          index Cond: (id=48951)
         -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
          Recheck Cond: (d1.itemvalue=d2.itemvalue)
          ->  Bitmap index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)  ✓
           index Cond: (d1.itemvalue=d2.itemvalue)
 -> Hash (cost=2052.07..2052.07 rows=1799 width=4)
  -> HashAggregate (cost=2034.08..2052.07 rows=1799 width=4)
   ->  Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=3791 width=4)         ✗
    Filter: (role='user'::text)
-> Hash (cost=1271.03..1271.03 rows=1 width=4)
 -> Nested Loop (cost=0.00..1271.03 rows=1 width=4)
  ->  Seq Scan on wascontrolledby wcb (cost=0.00..1262.74 rows=1 width=8)            ✗
   Filter: (role='MobyBlat'::text)
  -> index Scan using agent_pkey on agent a (cost=0.00..8.28 rows=1 width=4)
   index Cond: (a.id=wcb.cause)
   Filter: (a.value='MobyBlat'::text)
 SubPlan 1                                                                              ✓
 -> Aggregate (cost=1936.91..1936.92 rows=1 width=4)
  ->  Seq Scan on used u (cost=0.00..1936.90 rows=2 width=4)                         ✗
   Filter: (cause=$0)
SubPlan 2                                                                               ✓
 -> Aggregate (cost=2024.61..2024.62 rows=1 width=4)
  ->  Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=2 width=4)             ✗
   Filter: (effect=$0)
```

Query plan B.4: Query plan for query A.3 with all effective indices. Created indices: i1usedcause, i2wgbeffect, i3artdataid, i4wcbrole, i7processvaluefinished. Query form 3, datacompare view 2 (completeitemindex).

```
Hash Semi Join (cost=20615.04..27831.32 rows=1 width=4)
 Hash Cond: (t.id = wcb.effect)
 -> Nested Loop Semi Join (cost=20598.47..27797.52 rows=192 width=40)
  -> Nested Loop Semi Join (cost=20598.47..26447.18 rows=192 width=36)
   -> Nested Loop Semi Join (cost=20598.47..25096.84 rows=192 width=32)
    -> Hash Join (cost=20598.47..23746.50 rows=192 width=28)
     Hash Cond: (t.id = u.cause)
     -> Nested Loop Semi Join (cost=15144.92..18290.31 rows=192 width=24)
      -> Nested Loop Semi Join (cost=15144.92..16939.97 rows=192 width=20)
       -> Hash Semi Join (cost=15144.92..15589.63 rows=192 width=16)
        Hash Cond: (u.effect = a.id)
        -> Hash Semi Join (cost=11657.36..12098.95 rows=192 width=20)
         Hash Cond: (t.id = u.cause)
         -> Hash Join (cost=5474.14..5905.17 rows=192 width=8)
          Hash Cond: (t.id = u.cause)
          -> Bitmap Heap Scan on task t (cost=5.75..434.14 rows=192 width=4)
           Recheck Cond: (value = 'MobyBlat'::text)
           Filter: (NOT (finished IS NULL))
           -> Bitmap Index Scan on i7processvaluefinished (cost=0.00..5.70 rows=192 width=0)      ✓
            Index Cond: (role = 'MobyBlat'::text)
          -> Hash (cost=5460.28..5460.28 rows=649 width=4)
           -> HashAggregate (cost=5453.79..5460.28 rows=649 width=4)
            -> Hash Semi Join (cost=3487.56..5449.88 rows=1564 width=4)
             Hash Cond: (u.effect = a.id)
             -> Seq Scan on used u (cost=0.00..1936.90 rows=1564 width=8)      ×
              Filter: (role = 'user'::text)
             -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
              -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
               Hash Cond: (a.dataid = d1.id)
               -> Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)      ×
               -> Hash (cost=111.38..111.38 rows=3927 width=4)
                -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
                 -> Index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
                  Index Cond: (id = 48966)
                 -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
                  Recheck Cond: (d1.itemvalue = d2.itemvalue)
                  -> Bitmap Index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)      ✓
                   Index Cond: (d1.itemvalue = d2.itemvalue)
         -> Hash (cost=6179.88..6179.88 rows=267 width=12)
          -> Nested Loop (cost=5429.47..6179.88 rows=267 width=12)
           -> HashAggregate (cost=5429.47..5430.58 rows=111 width=4)
            -> Hash Semi Join (cost=3487.56..5428.80 rows=267 width=4)
             Hash Cond: (u.effect = a.id)
             -> Seq Scan on used u (cost=0.00..1936.90 rows=267 width=8)      ×
              Filter: (role = 'q'::text)
             -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
              -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
               Hash Cond: (a.dataid = d1.id)
               -> Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)      ×
               -> Hash (cost=111.38..111.38 rows=3927 width=4)
                -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
                 -> Index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
                  Index Cond: (id = 48984)
                 -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
                  Recheck Cond: (d1.itemvalue = d2.itemvalue)
                  -> Bitmap Index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)      ✓
                   Index Cond: (d1.itemvalue = d2.itemvalue)
           -> Index Scan using i1usedcause on used u (cost=0.00..6.74 rows=1 width=8)      ✓
            Index Cond: (u.cause = u.cause)
            Filter: (u.role = 'out'::text)
        -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
         -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
          Hash Cond: (a.dataid = d1.id)
          -> Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)      ×
          -> Hash (cost=111.38..111.38 rows=3927 width=4)
           -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
            -> Index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
             Index Cond: (id = 48985)
            -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
             Recheck Cond: (d1.itemvalue = d2.itemvalue)
             -> Bitmap Index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)      ✓
              Index Cond: (d1.itemvalue = d2.itemvalue)
       -> Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)      ✓
        Index Cond: (wgb.effect = t.id)
        Filter: (wgb.role = 'no_hits'::text)
      -> Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)      ✓
      Index Cond: (wgb.effect = t.id)
      Filter: (wgb.role = 'output'::text)
    -> Hash (cost=5448.17..5448.17 rows=430 width=4)
     -> HashAggregate (cost=5443.87..5448.17 rows=430 width=4)
      -> Hash Semi Join (cost=3487.56..5441.28 rows=1035 width=4)
       Hash Cond: (u.effect = a.id)
       -> Seq Scan on used u (cost=0.00..1936.90 rows=1035 width=8)                ×
        Filter: (role = 'input'::text)
       -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
        -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
         Hash Cond: (a.dataid = d1.id)
         -> Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)         ×
         -> Hash (cost=111.38..111.38 rows=3927 width=4)
          -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
           -> Index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
            Index Cond: (id = 48951)
           -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
            Recheck Cond: (d1.itemvalue = d2.itemvalue)
            -> Bitmap Index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)  √
             Index Cond: (d1.itemvalue = d2.itemvalue)
   -> Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)   √
    Index Cond: (wgb.effect = t.id)
    Filter: (wgb.role = 'result'::text)
  -> Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)    √
   Index Cond: (wgb.effect = t.id)
   Filter: (wgb.role = 'serviceNotes'::text)
 -> Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)     √
  Index Cond: (wgb.effect = t.id)
  Filter: (wgb.role = 'user'::text)
-> Hash (cost=16.56..16.56 rows=1 width=4)
 -> Nested Loop (cost=0.00..16.56 rows=1 width=4)
  -> Index Scan using i4wcbrole on wascontrolledby wcb (cost=0.00..8.28 rows=1 width=8)     √
   Index Cond: (role = 'MobyBlat'::text)
  -> Index Scan using agent_pkey on agent a (cost=0.00..8.28 rows=1 width=4)
   Index Cond: (a.id = wcb.cause)
   Filter: (a.value = 'MobyBlat'::text)
SubPlan 1
 -> Aggregate (cost=8.32..8.33 rows=1 width=4)
  -> Index Scan using i1usedcause on used u (cost=0.00..8.31 rows=2 width=4)               √
   Index Cond: (cause = $0)
SubPlan 2
 -> Aggregate (cost=8.37..8.38 rows=1 width=4)
  -> Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..8.37 rows=2 width=4)    √
   Index Cond: (effect = $0)
```

Query plan B.5: Query plan for query A.4, Download url task. data-compare view 2 (completeitemindex).

```
Nested Loop Semi Join (cost=4758.59..12839.72 rows=1 width=4)
 Join Filter: (t.id=wgb.effect)
 -> Nested Loop Semi Join (cost=4758.59..6811.86 rows=1 width=12)
  Join Filter: (t.id=u.cause)
  -> Nested Loop (cost=1271.03..1279.33 rows=1 width=8)
   -> HashAggregate (cost=1271.03..1271.04 rows=1 width=4)
    -> Nested Loop (cost=0.00..1271.03 rows=1 width=4)
     ->  Seq Scan on wascontrolledby wcb (cost=0.00..1262.74 rows=1 width=8)        ×
      Filter: (role='Scripting Role'::text)
     -> index Scan using agent_pkey on agent a (cost=0.00..8.28 rows=1 width=4)
      index Cond: (a.id=wcb.cause)
      Filter: (a.value='Perl agent localhost'::text)
   -> index Scan using process_pkey on process t (cost=0.00..8.28 rows=1 width=4)
    index Cond: (t.id=wcb.effect)
    Filter: ((not (t.finished is null)) and (t.value='Download url'::text))
  -> Hash Semi Join (cost=3487.56..5485.55 rows=3759 width=4)
   Hash Cond: (u.effect=a.id)
   ->  Seq Scan on used u (cost=0.00..1936.90 rows=3759 width=8)        ×
    Filter: (role='url'::text)
   -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
    -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
     Hash Cond: (a.dataid=d1.id)
     ->  Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)        ×
     -> Hash (cost=111.38..111.38 rows=3927 width=4)
      -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
       -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
        index Cond: (id=82971)
       -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
        Recheck Cond: (d1.itemvalue=d2.itemvalue)
        ->  Bitmap index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)        ✓
         index Cond: (d1.itemvalue=d2.itemvalue)
 ->  Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=3338 width=4)        ×
  Filter: (wgb.role='content'::text)
 SubPlan 1
  -> Aggregate (cost=1936.91..1936.92 rows=1 width=4)
   ->  Seq Scan on used u (cost=0.00..1936.90 rows=2 width=4)        ×
    Filter: (cause=$0)
 SubPlan 2
  -> Aggregate (cost=2024.61..2024.62 rows=1 width=4)
   ->  Seq Scan on wasgeneratedby wgb (cost=0.00..2024.60 rows=2 width=4)        ×
    Filter: (effect=$0)
```

Query plan B.6: Query plan for query A.4 with all effective indices. Created indices: i1usedcause, i2wgbeffect, i3artdataid, i4wcbrole, i7processvaluefinished. Query form 3, datacompare view 2 (completeitemindex).

```
Nested Loop Semi Join (cost=1618.72..2381.00 rows=1 width=4)
 -> Hash Semi Join (cost=1618.72..2359.83 rows=1 width=12)
  Hash Cond: (u.effect=a.id)
  -> Hash Semi Join (cost=74.23..803.09 rows=754 width=16)
   Hash Cond: (t.id=u.cause)
   -> Bitmap Heap Scan on process t (cost=49.31..765.36 rows=1684 width=4)
    Recheck Cond: (role='Download url'::text)
    Filter: (not (finished is null))
    -> Bitmap Index Scan on i7processvaluefinished (cost=0.00..48.89 rows=1684 width=0)          √
     index Cond: (role='Download url'::text)
   -> Hash (cost=24.91..24.91 rows=1 width=12)
    -> Nested Loop (cost=16.57..24.91 rows=1 width=12)
     -> HashAggregate (cost=16.57..16.58 rows=1 width=4)
      -> Nested Loop (cost=0.00..16.56 rows=1 width=4)
       -> Index Scan using i4wcbrole on wascontrolledby wcb (cost=0.00..8.28 rows=1 width=8)     √
        index Cond: (role='Scripting Role'::text)
       -> Index Scan using agent_pkey on agent a (cost=0.00..8.28 rows=1 width=4)
        index Cond: (a.id=wcb.cause)
        Filter: (a.value='Perl agent localhost'::text)
     -> Index Scan using i1usedcause on used u (cost=0.00..8.32 rows=1 width=8)                  √
      index Cond: (u.cause=wcb.effect)
      Filter: (u.role='url'::text)
  -> Hash (cost=1500.40..1500.40 rows=3527 width=4)
   -> Nested Loop (cost=0.00..1500.40 rows=3527 width=4)
    -> Nested Loop (cost=0.00..117.34 rows=3927 width=4)
     -> Index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
      index Cond: (id=82971)
     -> Index Scan using completeitemindex on data d1 (cost=0.00..108.73 rows=26 width=223)      √
      index Cond: (d1.itemvalue=d2.itemvalue)
    -> Index Scan using i3artdataid on artifact a (cost=0.00..0.34 rows=1 width=8)               √
     index Cond: (a.dataid=d1.id)
 -> Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..4.45 rows=1 width=4)          √
  index Cond: (wgb.effect=t.id)
  Filter: (wgb.role='content'::text)
 SubPlan 1
  -> Aggregate (cost=8.32..8.33 rows=1 width=4)
   -> Index Scan using i1usedcause on used u (cost=0.00..8.31 rows=2 width=4)                    √
    index Cond: (cause=$0)
 SubPlan 2
  -> Aggregate (cost=8.37..8.38 rows=1 width=4)
   -> Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..8.37 rows=2 width=4)        √
    index Cond: (effect=$0)
```

Query plan B.7: Query plan for query A.3, BLAT task. Created indices: I_1-I_6. Query form 3, datacompare view 2.

```
Hash Semi Join (cost=17175.46..25403.41 rows=1 width=4)
 Hash Cond: (t.id = wcb.effect)
 -> Hash Join (cost=17158.88..25369.61 rows=192 width=40)
  Hash Cond: (t.id = u.cause)
  -> Nested Loop Semi Join (cost=12668.78..20876.87 rows=192 width=36)
   -> Nested Loop Semi Join (cost=12668.78..19526.52 rows=192 width=32)
    -> Nested Loop Semi Join (cost=12668.78..18176.18 rows=192 width=28)
     -> Nested Loop Semi Join (cost=12668.78..16825.84 rows=192 width=24)
      -> Nested Loop Semi Join (cost=12668.78..15475.50 rows=192 width=20)
       -> Hash Join (cost=12668.78..14125.16 rows=192 width=16)
        Hash Cond: (t.id = u.cause)
        -> Hash Join (cost=8586.10..10039.84 rows=192 width=12)
         Hash Cond: (t.id = u.cause)
         -> Hash Join (cost=4503.41..5954.51 rows=192 width=8)
          Hash Cond: (t.id = u.cause)
          ->  Seq Scan on process t (cost=0.00..1448.46 rows=192 width=4)                ×
           Filter: ((NOT (finished IS NULL)) AND (value = ’MobyBlat’::text))
          -> Hash (cost=4495.30..4495.30 rows=649 width=4)
           -> HashAggregate (cost=4488.81..4495.30 rows=649 width=4)
            -> Hash Semi Join (cost=3527.95..4484.90 rows=1564 width=4)
             Hash Cond: (u.effect = a.id)
             -> Bitmap Heap Scan on used u (cost=40.38..971.92 rows=1564 width=8)
              Recheck Cond: (role = ’user’::text)
              ->  Bitmap Index Scan on i6usedrole (cost=0.00..39.99 rows=1564 width=0)      ✓
               Index Cond: (role = ’user’::text)
             -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
              -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
               Hash Cond: (a.dataid = d1.id)
               ->  Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)           ×
               -> Hash (cost=111.38..111.38 rows=3927 width=4)
                -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
                 -> Index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
                  Index Cond: (id = 48966)
                 -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
                  Recheck Cond: (d1.itemvalue = d2.itemvalue)
                  ->  Bitmap Index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)  ✓
                   Index Cond: (d1.itemvalue = d2.itemvalue)
         -> Hash (cost=4081.30..4081.30 rows=111 width=4)
          -> HashAggregate (cost=4080.19..4081.30 rows=111 width=4)
           -> Hash Semi Join (cost=3497.90..4079.52 rows=267 width=4)
            Hash Cond: (u.effect = a.id)
            -> Bitmap Heap Scan on used u (cost=10.33..587.62 rows=267 width=8)
             Recheck Cond: (role = ’out’::text)
             ->  Bitmap Index Scan on i6usedrole (cost=0.00..10.27 rows=267 width=0)         ✓
              Index Cond: (role = ’out’::text)
            -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
             -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
              Hash Cond: (a.dataid = d1.id)
              ->  Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)            ×
              -> Hash (cost=111.38..111.38 rows=3927 width=4)
               -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
                -> Index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
                 Index Cond: (id = 48985)
                -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
                 Recheck Cond: (d1.itemvalue = d2.itemvalue)
                 ->  Bitmap Index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)   ✓
                  Index Cond: (d1.itemvalue = d2.itemvalue)
        -> Hash (cost=4081.30..4081.30 rows=111 width=4)
         -> HashAggregate (cost=4080.19..4081.30 rows=111 width=4)
          -> Hash Semi Join (cost=3497.90..4079.52 rows=267 width=4)
           Hash Cond: (u.effect = a.id)
           -> Bitmap Heap Scan on used u (cost=10.33..587.62 rows=267 width=8)
            Recheck Cond: (role = ’q’::text)
            ->  Bitmap Index Scan on i6usedrole (cost=0.00..10.27 rows=267 width=0)          ✓
             Index Cond: (role = ’q’::text)
           -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
            -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
             Hash Cond: (a.dataid = d1.id)
             ->  Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)             ×
             -> Hash (cost=111.38..111.38 rows=3927 width=4)
              -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
               -> Index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
                Index Cond: (id = 48984)
               -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
                Recheck Cond: (d1.itemvalue = d2.itemvalue)
                ->  Bitmap Index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)    ✓
                 Index Cond: (d1.itemvalue = d2.itemvalue)
       ->  Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)    ✓
        Index Cond: (wgb.effect = t.id)
        Filter: (wgb.role = ’no_hits’::text)
     ->  Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)     √
      Index Cond: (wgb.effect = t.id)
      Filter: (wgb.role = 'output'::text)
    ->  Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)      √
     Index Cond: (wgb.effect = t.id)
     Filter: (wgb.role = 'result'::text)
   ->  Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)       √
    Index Cond: (wgb.effect = t.id)
    Filter: (wgb.role = 'serviceNotes'::text)
  ->  Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)        √
   Index Cond: (wgb.effect = t.id)
   Filter: (wgb.role = 'user'::text)
 -> Hash (cost=4484.73..4484.73 rows=430 width=4)
  -> HashAggregate (cost=4480.43..4484.73 rows=430 width=4)
   -> Hash Semi Join (cost=3515.85..4477.84 rows=1035 width=4)
    Hash Cond: (u.effect = a.id)
    -> Bitmap Heap Scan on used u (cost=28.29..973.46 rows=1035 width=8)
     Recheck Cond: (role = 'input'::text)
     ->  Bitmap Index Scan on i6usedrole (cost=0.00..28.03 rows=1035 width=0)                    √
      Index Cond: (role = 'input'::text)
    -> Hash (cost=3443.48..3443.48 rows=3527 width=4)
     -> Hash Join (cost=160.47..3443.48 rows=3527 width=4)
      Hash Cond: (a.dataid = d1.id)
      ->  Seq Scan on artifact a (cost=0.00..2479.00 rows=102500 width=8)                        ×
      -> Hash (cost=111.38..111.38 rows=3927 width=4)
       -> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
        -> Index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
         Index Cond: (id = 48951)
        -> Bitmap Heap Scan on data d1 (cost=4.48..102.77 rows=26 width=223)
         Recheck Cond: (d1.itemvalue = d2.itemvalue)
         ->  Bitmap Index Scan on completeitemindex (cost=0.00..4.47 rows=26 width=0)            √
          Index Cond: (d1.itemvalue = d2.itemvalue)
-> Hash (cost=16.56..16.56 rows=1 width=4)
 -> Nested Loop (cost=0.00..16.56 rows=1 width=4)
  ->  Index Scan using i4wcbrole on wascontrolledby wcb (cost=0.00..8.28 rows=1 width=8)         √
   Index Cond: (value = 'MobyBlat'::text)
  -> Index Scan using agent_pkey on agent a (cost=0.00..8.28 rows=1 width=4)
   Index Cond: (a.id = wcb.cause)
   Filter: (a.value = 'MobyBlat'::text)
SubPlan 1
 -> Aggregate (cost=8.32..8.33 rows=1 width=4)
  ->  Index Scan using i1usedcause on used u (cost=0.00..8.31 rows=2 width=4)                    √
   Index Cond: (cause = $0)
SubPlan 2
 -> Aggregate (cost=8.37..8.38 rows=1 width=4)
  ->  Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..8.37 rows=2 width=4)        √
   Index Cond: (effect = $0)
```

Query plan B.8: Query plan for query A.4, Download task. Created indices: I_1-I_6. I_6, usedrole, reduces performance for Download queries.

```
Nested Loop Semi Join (cost=1654.45..2726.98 rows=1 width=4)
 -> Nested Loop Semi Join (cost=1654.45..2705.81 rows=1 width=12)
  Join Filter: (t.id=u.cause)
  -> Nested Loop (cost=16.57..24.87 rows=1 width=8)
   -> HashAggregate (cost=16.57..16.58 rows=1 width=4)
    -> Nested Loop (cost=0.00..16.56 rows=1 width=4)
     ->  index Scan using i4wcbrole on wascontrolledby wcb (cost=0.00..8.28 rows=1 width=8)        ✓
      index Cond: (role='Scripting Role'::text)
     -> index Scan using agent_pkey on agent a (cost=0.00..8.28 rows=1 width=4)
      index Cond: (a.id=wcb.cause)
      Filter: (a.value='Perl agent localhost'::text)
   -> index Scan using process_pkey on process t (cost=0.00..8.28 rows=1 width=4)
    index Cond: (t.id=wcb.effect)
    Filter: ((not (t.finished is null)) and (t.value='Download url'::text))
  -> Hash Semi Join (cost=1637.88..2633.95 rows=3759 width=4)
   Hash Cond: (u.effect=a.id)
   -> Bitmap Heap Scan on used u (cost=93.40..1028.38 rows=3759 width=8)
    Recheck Cond: (role='url'::text)
    ->  Bitmap index Scan on i6usedrole (cost=0.00..92.46 rows=3759 width=0)                        ×
     index Cond: (role='url'::text)
   -> Hash (cost=1500.40..1500.40 rows=3527 width=4)
    -> Nested Loop (cost=0.00..1500.40 rows=3527 width=4)
     -> Nested Loop (cost=0.00..117.34 rows=3927 width=4)
      -> index Scan using data_pkey on data d2 (cost=0.00..8.28 rows=1 width=219)
       index Cond: (id=82971)
      ->  index Scan using completeitemindex on data d1 (cost=0.00..108.73 rows=26 width=223)      ✓
       index Cond: (d1.itemvalue=d2.itemvalue)
     ->  index Scan using i3artdataid on artifact a (cost=0.00..0.34 rows=1 width=8)              ✓
      index Cond: (a.dataid=d1.id)
 ->  index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..4.45 rows=1 width=4)            ✓
  index Cond: (wgb.effect=t.id)
  Filter: (wgb.role='content'::text)
 SubPlan 1
  -> Aggregate (cost=8.32..8.33 rows=1 width=4)
   ->  index Scan using i1usedcause on used u (cost=0.00..8.31 rows=2 width=4)                    ✓
    index Cond: (cause=$0)
 SubPlan 2
  -> Aggregate (cost=8.37..8.38 rows=1 width=4)
   ->  index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..8.37 rows=2 width=4)        ✓
    index Cond: (effect=$0)
```

Query plan B.9: Query plan for a phase 2 query.

```
Nested Loop (cost=0.00..25.37 rows=2 width=11)
  ->   Index Scan using wgbeffect on wasgeneratedby wgb (cost=0.00..8.80rows=2 width=11)           ✓
        Index Cond: (effect = 94076)
  ->   Index Scan using artifact_pkey on artifact a (cost=0.00..8.27 rows=1 width=8)              ✓
        Index Cond: (a.id = wgb.cause)
```

Bibliography

[1] Altintas, I., Barney, O., and Jaeger-Frank, E. Provenance collection support in the Kepler scientific workflow system. Moreau, L., and Foster, I., Eds., vol. 4145 of *Lecture Notes in Computer Science*, Springer, pp. 118–132, 2006. doi: 10.1007/11890850_14.

[2] Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludäscher, B., and Mock, S. Kepler: an extensible system for design and execution of scientific workflows. Hatzopoulos, M., and Manolopoulos, Y., Eds., *SSDBM'04: 16th International Conference on Scientific and Statistical Database Management*, pp. 423–424, 2004. doi: 10.1109/ssdm.2004.1311241.

[3] Altschul, S. F., Madden, T. L., Schäffer, A. A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. J. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997. doi: 10.1093/nar/25.17.3389.

[4] Barga, R. S., and Digiampietri, L. A. Automatic generation of workflow provenance. Moreau, L., and Foster, I., Eds., vol. 4145 of *Lecure Notes in Computer Science*, pp. 1–9, 2006. doi: 10.1007/11890850_1.

[5] Barga, R. S., Jackson, J., Araujo, N., Guo, D., Gautam, N., Grochow, K., and Lazowska, E. Trident: Scientific workflow workbench for oceanography. *SERVICES '08: Congress on Services - Part I*, IEEE Computer Society, pp. 465–466, 2008. doi: 10.1109/services-1.2008.101.

[6] Bowers, S., McPhillips, T., Riddle, S., Anand, M., and Ludäscher, B. Kepler/pPOD: scientific workflow and provenance support for assembling the tree of life. vol. 5272 of *Lecture Notes in Computer Science*, pp. 70–77, 2008. doi: 10.1007/978-3-540-89965-5_9.

[7] Brazma, A., Hingamp, P., Quackenbush, J., Sherlock, G., Spellman, P., Stoeckert, C., Aach, J., Ansorge, W., Ball, C. A., Causton, H. C., Gaasterland, T., Glenisson, P., Holstege, F. C. P., Kim, I. F., Markowitz, V., Matese, J. C., Parkinson, H., Robinson, A., Sarkans, U., Schulze-Kremer, S., Stewart, J., Taylor, R., Vilo, J., and Vingron, M. Minimum information about a microarray experiment (MIAME)-toward standards for microarray data. *Nature genetics*, 29(4):365–371, 2001. doi: 10.1038/ng1201-365.

[8] Callahan, S. P., Freire, J., Santos, E., Scheidegger, C. E., Silva, C. T., and Vo, H. T. Vistrails: visualization meets data management. Hristidis, V., and Polyzotis, N., Eds., *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ACM, pp. 745–747, 2006. doi: 10.1145/1142473.1142574.

[9] da Cruz, S. M. S., Campos, M. L. M., and Mattoso, M. Towards a taxonomy of provenance in scientific workflow management systems. *IEEE Congress on SERVICES - I*, 0:259–266, 2009.

[10] Davidson, S. B., Boulakia, S. C., Eyal, A., Ludäscher, B., McPhillips, T. M., Bowers, S., Anand, M. K., and Freire, J. Provenance in scientific workflow systems. *IEEE Data Engineering Bulletin*, 30(4):44–50, 2007.

[11] Davidson, S. B., and Freire, J. Provenance and scientific workflows: challenges and opportunities. *SIGMOD'08: SIGMOD International conference on Management of data*, ACM, pp. 1345–1350, 2008. doi: 10.1145/1376616.1376772.

[12] Deelman, E., Gannon, D., Shields, M., and Taylor, I. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25:528–540, 2008. doi: 10.1016/j.future.2008.06.012.

[13] Dolgert, A., Gibbons, L., Jones, C. D., Kuznetsov, V., Riedewald, M., Riley, D., Sharp, G. J., and Wittich, P. Provenance in high-energy physics workflows. *Computing in Science and Engineering*, 10(3):22-29, 2008. doi: 10.1109/mcse.2008.81.

[14] Eisenberg, A., and Melton, J. SQL:1999, formerly known as SQL3. *ACM SIGMOD Record*, 28(1):131–138, 1999. doi: 10.1145/309844.310075.

[15] Flicek, P., Aken, B. L., Beal, K., Ballester, B., Caccamo, M., Chen, Y., Clarke, L., Coates, G., Cunningham, F., Cutts, T., Down, T., Dyer, S. C., Eyre, T., Fitzgerald, S., Fernandez-Banet, J., Gräf, S., Haider, S., Hammond, M., Holland, R., Howe, K. L., Howe, K., Johnson, N., Jenkinson, A., Kähäri, A., Keefe, D., Kokocinski, F., Kulesha, E., Lawson, D., Longden, I., Megy, K., Meidl, P., Overduin, B., Parker, A., Pritchard, B., Prlic, A., Rice, S., Rios, D., Schuster, M., Sealy, I., Slater, G., Smedley, D., Spudich, G., Trevanion, S., Vilella, A. J., Vogel, J., White, S., Wood, M., Birney, E., Cox, T., Curwen, V., Durbin, R., Fernandez-Suarez, X. M., Herrero, J., Hubbard, T. J., Kasprzyk, A., Proctor, G., Smith, J., Ureta-Vidal, A., and Searle, S. Ensembl 2008. *Nucleic acids research*, 36(Database issue):D707-D714, 2008. doi: 10.1093/nar/gkm988.

[16] Freire, J., Koop, D., and Moreau, L., Eds. Second International Provenance and Annotation Workshop, IPAW'08, vol. 5272 of *Lecture Notes in Computer Science*, Springer. 2008.

[17] Gil, Y., Deelman, E., Ellisman, M., Fahringer, T., Fox, G., Gannon, D., Goble, C. A., Livny, M., Moreau, L., and Myers, J. Examining the challenges of scientific workflows. *Computer*, 40(12):24–32, 2007. doi: 10.1109/MC.2007.421.

[18] Goble, C. A. Position statement: Musings on provenance, workflow and (semantic web) annotations for bioinformatics. Zhao, Y., Ed., *Workshop on Data Derivation and Provenance*, 2002.

[19] Goble, C. A., and De Roure, D. C. $^{\text{my}}$Experiment: social networking for workflow-using e-scientists. Deelman, E., and Taylor, I., Eds., *WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science*, ACM, pp. 1–2, 2007. doi: 10.1145/1273360.1273361.

[20] Greenwood, M., Goble, C. A., Stevens, R., Zhao, J., Addis, M., Marvin, D., Moreau, L., and Oinn, T. Provenance of e-science experiments - experience from bioinformatics. Cox, S., Ed., *Proceedings of UK e-Science All Hands Meeting 2003*, pp. 223–226, 2003.

[21] Groth, P. *The Origin of Data: Enabling the Determination of Provenance in Multi-institutional Scientific Systems through the Documentation of Processes*. PhD thesis, University of Southampton, 2007.

[22] Kawas, E., Senger, M., and Wilkinson, M. D. BioMoby extensions to the Taverna workflow management and enactment software. *BMC bioinformatics*, 7(523):1–13, 2006. doi: 10.1186/1471-2105-7-523.

[23] Kent, W. J. BLAT: The BLAST-like alignment tool. *Genome Research*, 12(4):656–664, 2002. doi: 10.1101/gr.229202.

[24] Kwasnikowska, N., and Van den Bussche, J. Mapping the NRC Dataflow Model to the Open Provenance Model. vol. 5272 of *Lecture Notes in Computer Science*, pp. 3–16, 2008. doi: 10.1007/978-3-540-89965-5_3.

[25] Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E. A., Tao, J., and Zhao, Y. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006. doi: 10.1002/cpe.994.

[26] Maglott, D., Ostell, J., Pruitt, K. D., and Tatusova, T. Entrez gene: gene-centered information at ncbi. *Nucleic Acids Research*, 33:54–58, 2005. doi: doi:10.1093/nar/gki031.

[27] Moreau, L., and Foster, I., Eds. International Provenance and Annotation Workshop, IPAW 2006, vol. 4145 of *Lecture Notes in Computer Science*, Springer. 2007.

[28] Moreau, L., Freire, J., Futrelle, J., McGrath, R., Myers, J., and Paulson, P. The Open Provenance Model: An overview. vol. 5272 of *Lecture Notes in Computer Science*, pp. 323–326, 2008. doi: 10.1007/978-3-540-89965-5_31.

[29] Moreau, L., Groth, P., Miles, S., Vazquez, J., Ibbotson, J., Jiang, S., Munroe, S., Rana, O., Schreiber, A., Tan, V., and Varga, L. The Provenance of Electronic Data. *Communications of the ACM*, 51(4):52–58, 2008.

[30] Moreau, L., and Ludäscher, B., Eds. Special Issue: The First Provenance Challenge, vol. 20(5) of *Concurrency and Computation: Practice and Experience*, Wiley. 2008.

[31] Moreau, L., Plale, B., Miles, S., Goble, C. A., Missier, P., Barga, R. S., Simmhan, Y. L., Futrelle, J., McGrath, R., Myers, J., Paulson, P., Bowers, S., Ludäscher, B., Kwasnikowska, N., van den Bussche, J., Ellkvist, T., Freire, J., and Groth, P. The Open Provenance Model (v1.01). Technical report, University of Southampton, 2008.

[32] Neerincx, P. B. T., Rauwerda, H., Nie, H., Groenen, M. A. M., Breit, T. M., and Leunissen, J. A. M. OligoRAP - An Oligo Re-Annotation Pipeline to improve annotation and estimate target specificity. *BMC Proceedings*, 3(Suppl 4):S4, 2009. doi: 10.1186/1753-6561-3-S4-S4.

[33] Newcomer, E. *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Addison-Wesley, 2002.

[34] Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M. R., Wipat, A., and Li, P. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004. doi: 10.1093/bioinformatics/bth361.

[35] Oinn, T., Greenwood, M., Addis, M., Alpdemir, M. N., Ferris, J., Glover, K., Goble, C. A., Goderis, A., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M. R., Senger, M., Stevens, R., Wipat, A., and Wroe, C. Taverna: lessons in creating a workflow environment for the life sciences: Research articles. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006. doi: 10.1002/cpe.993.

[36] Roure, D. D., Goble, C., and Stevens, R. Designing the $^{\text{my}}$experiment virtual research environment for the social sharing of workflows. Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007):603-610, 2007. doi: 10.1109/e-science.2007.29.

[37] Sahoo, S. S., Barga, R. S., Goldstein, J., and Sheth, A. P. Provenance algebra and materialized view-based provenance management. Technical report, Microsoft Research, 2008.

[38] Seltzsam, S., Holzhauser, R., and Kemper, A. Semantic caching for web services. Benatallah, B., Casati, F., and Traverso, P., Eds., vol. 3826 of *Lecture Notes in Computer Science*, Springer, pp. 324–340, 2005. doi: 10.1007/11596141_25.

[39] Senger, M., Rice, P., and Oinn, T. Soaplab - a unified sesame door to analysis tools. *UK e-Science All Hands Meeting*, pp. 509–513, 2003.

[40] Simmhan, Y. L., Plale, B., and Gannon, D. A survey of data provenance in e-science. *ACM SIGMOD Record*, 34(3):31–36, 2005. doi: 10.1145/1084805.1084812.

[41] Stevens, R., Zhao, J., and Goble, C. Using provenance to manage knowledge of In Silico experiments. *Briefings in Bioinformatics*, 8(3):183–194, 2007. doi: 10.1093/bib/bbm015.

[42] Taylor, I. Triana generations. *E-SCIENCE '06: Second IEEE International Conference on e-Science and Grid Computing*, IEEE Computer Society, p. 143, 2006. doi: 10.1109/e-science.2006.146.

[43] Taylor, I., Harrison, A., Mastroianni, C., and Shields, M. Cache for workflows. *WORKS'07: 2nd workshop on Workflows in support of large-scale science*, ACM, pp. 13–20, 2007. doi: 10.1145/1273360.1273363.

[44] Taylor, I., Shields, M., Wang, I., and Harrison, A. Visual grid workflow in Triana. *Grid Computing*, 3(3):153–169, 2005. doi: 10.1007/s10723-005-9007-3.

[45] Taylor, I., Shields, M., Wang, I., and Harrison, A. The Triana workflow environment: Architecture and applications. Taylor, I., Deelman, E., Gannon, D., and Shields, M., Eds., *Workflows for e-Science*, pp. 320–339, 2007. doi: 10.1007/978-1-84628-757-2_20.

[46] van der Aalst, W. M. P., Aldred, L., Dumas, M., and ter Hofstede, A. H. M. Design and implementation of the YAWL system. Goos, G., Hartmanis, J., and van Leeuwen, J., Eds., *CAiSE'04: 16th International Conference on Advanced Information Systems Engineering*, pp. 142–159, 2004.

[47] van der Aalst, W. M. P., and ter Hofstede, A. H. M. Yawl: yet another workflow language. *Information Systems*, 30(4):245 - 275, 2005. doi: doi:10.1016/j.is.2004.02.002.

[48] van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5-51, 2003. doi: 10.1023/A:1022883727209.

[49] Wassink, I. *Work flows in life science*. PhD thesis, Group of Human Media Interaction, Department of Electrical Engineering, Mathematics and Computer Science (EEMCS), University of Twente, 2009. In press.

[50] Wassink, I., Ooms, M. J., and van der Vet, P. E. Designing workflows on the fly using e-BioFlow. *Joint ICSOC&ServiceWave 2009 Conference*, p. 15, 2009, accepted.

[51] Wassink, I., Rauwerda, H., Neerincx, P. B. T., van der Vet, P. E., Breit, T. M., Leunissen, J. A. M., and Nijholt, A. Using R in Taverna: RShell v1.2. *BMC Research Notes*, 2(138):1–8, 2009. doi: 10.1186/1756-0500-2-138.

[52] Wassink, I., Rauwerda, H., van der Vet, P. E., Breit, T. M., and Nijholt, A. e-BioFlow: Different perspectives on scientific workflows. Elloumi, M., Küng, J., Linial, M., Murphy, R. F., Schneider, K., and Toma, C., Eds., vol. 13 of *Communications in Computer and Information Science*, Springer, pp. 243–257, 2008. doi: 10.1007/978-3-540-70600-7_19.

[53] Wilkinson, M. D., and Links, M. BioMOBY: an open source biological web services proposal. *Briefings in Bioinformatics*, 3(4):331–341, 2002. doi: 10.1093/bib/3.4.331.

[54] Wong, S. C., Miles, S., Fang, W., Groth, P., Zauner, K.-P., and Moreau, L. Provenance-based validation of e-science experiments. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(1):28–38, 2007. DOI: 10.1016/j.websem.2006.11.003.

[55] Zhao, J., Goble, C. A., and Stevens, R. Semantically linking and browsing provenance logs for e-science. Bouzeghoub, M., Goble, C. A., Kashyap, V., and Spaccapietra, S., Eds., Springer, pp. 158–176, 2004.