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 enthusiasm 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, Chime-ra, 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.
1.3 Outline of this thesis

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.
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 an actor, 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, \url{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: myGRID. myGRID (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, focusing on interoperability.

One of the goals of the 3rd Provenance Challenge was to stimulate the development and use of concrete bindings and serializations 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 the 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 \textsc{used} and \textsc{was-generated-by}. 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


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 Irole, Orole and Prole are used for these relations respectively. Irole captures information about the context in which the artifact was used. By a similar argument, Orole captures information about the context in which the artifact was generated. Prole 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.

---

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
### 3.2 Improvement implementation details

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

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

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
<th>Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Tab2Multi-Sequence-FastaChunks</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>2a. BLAT</td>
<td>BLAT all sequences of a chunk against the transcriptome (UMT) and genome databases.</td>
<td></td>
</tr>
<tr>
<td>2b. BLAST</td>
<td>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.)</td>
<td>X</td>
</tr>
<tr>
<td>2c. ConcatenateFile</td>
<td>Concatenate the results of BLAT and BLAST (if a BLAST was performed).</td>
<td></td>
</tr>
<tr>
<td>3. Annotation-Analyser</td>
<td>Analyse the annotations of the previous results for the BLAT/BLAST results of both Genome and UMT.</td>
<td>X</td>
</tr>
<tr>
<td>4. OligoMergeXML</td>
<td>Merge the Genome and UMT results of the AnnotationAnalyser.</td>
<td></td>
</tr>
<tr>
<td>5. OligoQuality-Analyser</td>
<td>Perform a quality analyses over all merged OligoMergeXML results.</td>
<td>X</td>
</tr>
<tr>
<td>6. MakePieChart</td>
<td>The results of the QualityAnalyser can be visualised using a pie chart service.</td>
<td></td>
</tr>
</tbody>
</table>
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 con-
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 (bêta) 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.
3.4 Provenance implementation

Figure 3.2: OligoRAP in e-BioFlow. In the left navigation panel all sub-workflows are specified. The currently selected sub-workflow 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

3.5 Provenance Recording: an OPM-profile

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 \texttt{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 hierarchical 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 \texttt{Used} relation is created for each input port, between the process and artifact corresponding to the task and data item. The role of the \texttt{Used} relation is given the name of the input port. Likewise \texttt{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 \texttt{WasTriggeredBy} relation is created between the corresponding processes. For each service that executes a certain task, a \texttt{WasControlledBy} relation is created between the corresponding agent and process. The role of the task is assigned to the role of the \texttt{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 \textit{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.

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

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.
   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 OVERLAP(Ac\_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) Wtb\_1.
   Add Wtb\_1 to Ac\_main.
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
### 3.6 Running OligoRAP: results

<table>
<thead>
<tr>
<th>Property</th>
<th>Perl run</th>
<th>e-BioFlow run</th>
<th>Provenance run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>5:56:42</td>
<td>2:50:14</td>
<td>3:03:02</td>
</tr>
<tr>
<td>Compressed size</td>
<td>NA</td>
<td>345MB</td>
<td>392MB</td>
</tr>
<tr>
<td>Uncompressed size</td>
<td>3.3Gb</td>
<td>2.5Gb</td>
<td>2.9Gb</td>
</tr>
<tr>
<td>Files</td>
<td>994</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**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.

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

**Figure 3.6:**

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>
<thead>
<tr>
<th>Table</th>
<th>Rows</th>
<th>Size (in bytes)</th>
<th>% of total</th>
<th>∑ %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>64329</td>
<td>361324544</td>
<td>87,87</td>
<td></td>
</tr>
<tr>
<td>Artifact</td>
<td>64329</td>
<td>8994816</td>
<td>2,19</td>
<td></td>
</tr>
<tr>
<td>Task</td>
<td>36704</td>
<td>4325376</td>
<td>1,05</td>
<td></td>
</tr>
<tr>
<td>Agent</td>
<td>33835</td>
<td>3710976</td>
<td>0,90</td>
<td>4,14</td>
</tr>
<tr>
<td>WASGENERATEDBY</td>
<td>56063</td>
<td>6356992</td>
<td>1,55</td>
<td></td>
</tr>
<tr>
<td>WASTRIGGEREDBY</td>
<td>53080</td>
<td>5709824</td>
<td>1,39</td>
<td></td>
</tr>
<tr>
<td>USED</td>
<td>50710</td>
<td>5537792</td>
<td>1,35</td>
<td></td>
</tr>
<tr>
<td>WASCONTROLLEDBY</td>
<td>33835</td>
<td>3645440</td>
<td>0,89</td>
<td>5,18</td>
</tr>
<tr>
<td>Account</td>
<td>5739</td>
<td>696320</td>
<td>0,17</td>
<td></td>
</tr>
<tr>
<td>Overlap</td>
<td>5738</td>
<td>409600</td>
<td>0,10</td>
<td></td>
</tr>
<tr>
<td>Refines</td>
<td>2869</td>
<td>212992</td>
<td>0,05</td>
<td></td>
</tr>
<tr>
<td>Element_account</td>
<td>153745</td>
<td>10297344</td>
<td>2,50</td>
<td>2,82</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>(392MB) 411222016</td>
<td>100 12,13</td>
<td></td>
</tr>
</tbody>
</table>

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.

3.7 Discussion

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 \textit{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.

**Figure 3.7:** CPU load of e-BioFlow during an OligoRAP run.
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, \texttt{WAS GENERATED BY} and \texttt{USED} relations are shown.

(b) Resource perspective: only processes, agents \texttt{WAS CONTROLLED BY}, and \texttt{WAS TRIGGERED BY} relations are shown.

(c) Control flow perspective: only processes and \texttt{WAS TRIGGERED BY} relations are shown.

\textbf{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 $\text{WASGENERATEDBY}$, $\text{USED}$ and $\text{WASTRIiggeredBy}$. Of these only the $\text{WASGENERATEDBY}$ and $\text{USED}$ relations are shown in this perspective. The $\text{WASTRIiggeredBy}$ 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 $\text{WASCONTROLLEDBy}$ 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 $\text{WASTRIiggeredBy}$, 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, $\text{WASCONTROLLEDBy}$, is visible as well. Artifacts, $\text{USED}$ and $\text{WASGENERATEDBy}$ 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.
Figure 5.4: Screenshots of perspectives in e-BioFlow of a partial graph retrieved using a recursive query executed by the query panel.
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
5.1 Provenance Browser

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

<table>
<thead>
<tr>
<th>Query A:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT wgb.cause FROM process AS p, wasgeneratedby AS wgb WHERE</code></td>
</tr>
<tr>
<td><code>p.value = 'UngzipRole' AND wgb эффект=p.id AND p.run=1</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query B, neighbours included:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT * FROM element WHERE</code></td>
</tr>
<tr>
<td><code>id IN (Query 1) or</code></td>
</tr>
<tr>
<td><code>id IN (SELECT cause FROM relation WHERE effect IN (Query 1)) or</code></td>
</tr>
<tr>
<td><code>id IN (SELECT effect FROM relation WHERE cause IN (Query 1))</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query C, Transcriptome pie charts only:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT a.id FROM process AS p, wasgeneratedby AS wgb,</code></td>
</tr>
<tr>
<td><code>artifact AS a, data AS d WHERE</code></td>
</tr>
<tr>
<td><code>p.value = 'UngzipRole' AND wgb.target=p.id AND</code></td>
</tr>
<tr>
<td><code>p.run=1 AND a.id=wgb.source AND</code></td>
</tr>
<tr>
<td><code>a.dataid=d.id AND d.itemvalue = 'Transcriptome'</code></td>
</tr>
</tbody>
</table>

charts of the transcriptome, by matching the word ‘Transcriptome’ in the data value 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 \(n\)-th 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).
Figure 5.5: Screenshots of pie charts in e-BioFlow retrieved with the query panel.
5.1 Provenance Browser

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

```sql
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.effec = m.id AND r.cause = sub.id
) SELECT * FROM subelement
```

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

```sql
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.effec = 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 $3 \times 10^5$ 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 sub-workflows 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
Table 5.1: Accounts created during an OligoRAP run

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

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 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 encounter-
ing 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.
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:
6.1 Summary

- 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 hash value, 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.

```sql
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 = 'query' AND u2.effect=a.id AND
    a2.dataid=d2.d1id AND d2.d2id='48984' AND
    t.id=u3.cause AND u3.role = 'output' 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
```
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.

```sql
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 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.

```sql
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.
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 $\sqrt{\text{ }}$. 
### Query plan B.1

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

<table>
<thead>
<tr>
<th>Limit</th>
<th>(cost=44000.69..52096.96 rows=1 width=4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join Filter</td>
<td>((t.id=ul.cause) and (wcb.cause=ul.effect))</td>
</tr>
<tr>
<td>Nested Loop</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seg Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>index Scan using process_pkey on process t</td>
<td>(cost=0.00..3969.82 rows=1 width=4)</td>
</tr>
<tr>
<td>index Cond</td>
<td>(t.cause=ul.cause)</td>
</tr>
<tr>
<td>Filter</td>
<td>(not (t.finished is null)) and (t.value='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 2</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Hash Join</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 2</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
<tr>
<td>Sub Plan 1</td>
<td>(cost=1936.91..1936.92 rows=1 width=4)</td>
</tr>
<tr>
<td>Aggregate</td>
<td>(cost=0.00..5232.57 rows=1 width=12)</td>
</tr>
<tr>
<td>Seq Scan on wascontrolledby wcb</td>
<td>(cost=0.00..1262.74 rows=1 width=8)</td>
</tr>
<tr>
<td>Filter</td>
<td>(role='MobyBlat'::text)</td>
</tr>
</tbody>
</table>
Appendix B

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

```
limit (cost=1453.79..2608246.24 rows=1 width=4)
  -> Nested Loop Semi Join (cost=1453.79..2608246.24 rows=1 width=4)
    Join Filter: (t.id=vcb.effect)
      -> Nested Loop Semi Join (cost=11046.22..2599448.99 rows=1 width=32)
        Join Filter: (t.id=u.cause)
          -> Nested Loop Semi Join (cost=11046.22..2597307.12 rows=1 width=28)
            Join Filter: (t.id=wcb.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: (u.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)
                                                                -> Seq Scan using process_pkey on process t (cost=0.00..3967.41 rows=1 width=4)
                                                                  index Cond: (t.id=u.cause)
                                                                    -> Aggregate (cost=1936.91..1936.92 rows=1 width=4)
                                                                      -> 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: (u.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)
                                                                                                     -> Seq Scan using process_pkey on process t (cost=0.00..3967.41 rows=1 width=4)
                                                                                                       index Cond: (t.id=u.cause)
                                                                                                         -> Aggregate (cost=1936.91..1936.92 rows=1 width=4)
                                                                                                           -> 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: (u.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)```
```
| Query plans | 103 |

```sql
-> 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_hit '::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)
-> 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=6)
      -> Nested Loop (cost=4.48..111.38 rows=3927 width=6)
        -> 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=6.48..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)
Query plans

1. 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=5448.17..5448.17 rows=830 width=4)
     - HashAggregate (cost=5443.87..5448.17 rows=830 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)
         Filter: (role='input'::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 = 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)
           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='input'::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)
       -> 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 1
     -> 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)
  − Hash Join (cost=5474.14..5905.17 rows=192 width=8)
  − 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)
  − Hash Aggregate (cost=5453.79..5460.28 rows=649 width=4)
    Hash Cond: (u.effect = a.id)
  − Seq Scan on used u (cost=0..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 = 48964)
  − 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)
    Hash Aggregate (cost=5429.47..5430.58 rows=267 width=4)
    Hash Semi Join (cost=3487.56..3487.56 rows=1564 width=4)
    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 = 48964)
  − 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=0)
    Index Cond: (u.cause = u.cause)
    Filter: (role = 'out ':: text)
  − Hash (cost=3443.48..3443.48 rows=3527 width=4)
    Hash Join (cost=3487.56..3487.56 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 wsgeneratedby 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 wsgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)
Query plan B.5: Query plan for query A.4, Download url task. data-compare view 2 (completeitemindex).
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).
Query plan B.7: Query plan for query A.3, BLAT task. Created indices:
$I_1$-I_6$. Query form 3, datacompare view 2.
Query plans

111

Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)
Filter: (wgb.eff ect = t.id)
Index Cond: (wgb.eff ect = t.id)

Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)
Filter: (wgb.role = 'output ':: text)

Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)
Filter: (wgb.role = 'result ':: text)
Index Cond: (wgb.eff ect = t.id)
Filter: (wgb.role = 'serviceNotes ':: text)

Index Scan using i2wgbeffect on wasgeneratedby wgb (cost=0.00..7.02 rows=1 width=4)
Filter: (wgb.eff ect = t.id)

Hash (cost=4484.73..4484.73 rows=430 width=4)
-> Hash Aggregate (cost =4480.43..4484.73 rows =430 width =4)
-> Hash Semi Join (cost=3515.85..4477.84 rows=1035 width=4)
Hash Cond : (u. eff ect = a.id)

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)
-> Nested Loop (cost=4.48..111.38 rows=3927 width=4)
-> Hash (cost=111.36 ..111.38 rows=3927 width=4)

Hash (cost=16.56..16.56 rows=1 width=4)
-> Nested Loop (cost=0.00..16.56 rows=1 width=4)
-> Index Scan using i1usedcause on used u (cost=0.00..8.31 rows=2 width=4)
Index Cond: (cause = $0)

Aggregate (cost=6.32..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)

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: $I_1 - I_6$. $I_6$, used role, 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 role on wascontrolledby wcb (cost=0.00..8.28 rows=1 width=8)
            index Cond: (role='url'::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))
          → 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=219)
                → 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.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 role on wasgeneratedby wgb (cost=0.00..8.60 rows=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


