Maturing and Versioning an Architectural Knowledge Repository

Hubert ten Hove
University of Groningen, the Netherlands

25th of May 2010

Abstract

This thesis is written as a master thesis in cooperation with the research group SEARCH. SEARCH does research into software architecture, the work in this thesis allows for more research by upgrading and extending the current repository. This thesis answers two problem statements, the first "*The current repository is not mature enough"* was needed to stabilize our repository, the second "*The repository is not able to store the evolution of a software architecture"*. First the repository needs to be stable before the current repository can be extended with functionality that allows us to store this data. Our approach was also two folded, first we analyzed the current server to get better insights into its workings, after writing unit tests to validate the process constantly we upgraded the server on several fronts. The second part also starts with a requirement analysis, to fulfill these requirements, existing solutions where searched and custom solutions were proposed. All those solutions are validated against the requirements using a custom framework; Performance was the main requirement therefore a thorough performance test is done for all the proposed solutions. The results in the framework presented the best solution and this was developed into a prototype. The first problem statement stabilized our repository, and improved the performance by 50 times. Then the results of the research and the build prototype proved that the solution works. And reevaluating the prototype against the requirements showed that it still fulfilled all of them. Not only is the prototype sufficient for the needs of the research group, but it is also build in such a generic way that it can be adopted into the Sesame architecture. First the prototype and the rest of the system need to be tweaked and polished, this however is future work.

Acknowledgements

This thesis is the result of a long period of hard work. It was hard sometimes, and distractions and side tracks are hard to leave alone. Nevertheless I am grateful that I got the opportunity to work on such a great project and with such devoted people. This work could not have been done without them.

First of all I would like to thank SEARCH group of the University of Groningen for allowing me into their midst and performing research in collaboration with them and supporting me in this process. In particular, I would like to thank my supervisor Anton Jansen for always being there when I needed him, giving guidance and having insightful discussions about my work. Also I would like to thank Paris Avgeriou for his input and support.

Secondly I would like to thank my parent for believing in me that I could finish this thesis and my friends who stood behind me if I needed them giving a layman's opinion if I needed one, bringing things into a different perspective.

Finally I would like to thank my girlfriend for being supportive throughout my thesis work, for who at some times my mood must have been hard. Thank you for keeping up with me when I got stuck with some problem and did not have much attention for other things in life.

Contents

1 Introduction

The main focus of this thesis is maturing and extending the GRIFFIN knowledge hub, a part of the Knowledge Architect, a tool suite for software architecture. Software architecture is part of the software development process, a process that consists of a set of activities that as a result produces a software product. One of these activities is the creating of the software architecture. Software architecture plays an important role in managing the complex interactions and dependencies between stakeholders and providing a central artifact that can be used for reference [12]. However it proves to be very difficult to understand software architectures through their documentation. The documentation is only a description of the design but it tells nothing about the rationale behind the design, e.g. which choices and decisions made up the final design. This leads to serious problems in traceability in the software architecture. Bosch [13] defines the following problems:

1. Lack of first-class representation;
2. Design decisions crosscutting and intertwined;
3. High cost of change;
4. Design rules and constraints are violated;
5. Obsolete artifacts are not removed from architecture;
6. Lack of stakeholder communication;
7. Limited reusability.

Problems 3 to 7 all follow from the first two problems, *Lack of first-class representation*, and *Design decisions crosscutting and intertwined*. Without proper documentation about the design decisions it is impossible to have a full understanding about the architecture, this leads to mistakes in the maintenance and evolution of the software. The consequences of not documenting those architectural decisions are well known in the software industry: expensive system evolution, poor stakeholder communication, limited reusability of architectural assets, and poor traceability between requirements, architecture, and implementation [12]. Architectural design decisions are part of the wider notion of architectural knowledge, which is an emerging field in software architecture.

1.1 Architectural Knowledge

The last few years there has been a shift from software architecture to architectural knowledge (AK). AK encompasses software architecture; it is not only the architecture but also the knowledge needed to create that architecture. Therefore AK not only improves the architecting process but also the architecture itself [14]. At the moment there is no agreed upon definition for AK [15]. Some define AK as, AK = design decisions + design [16], others as AK = drivers, decisions, analysis [17], and some also include the process and the people aspects [18]. In this thesis, we embrace the view of the GRIFFIN project on AK: "*Architectural Knowledge is defined as the integrated representation of the software architecture of a software-intensive system or a family of systems, the architectural design decisions, and the external context / environment*" [30].

1.1.1 Architectural Knowledge Management

Architectural Knowledge Management (AKM) is Knowledge management (KM) specialized for AK. KM comprises a range of practices used in an organization to identify, create, represent, distribute, and enable adoption of insights and experiences [35]. Knowledge and AK can come in many forms and dimensions; one of these dimensions is the type of

knowledge. In KM, there is a distinction between two types: implicit and explicit knowledge [19]. Implicit knowledge, also known as tacit knowledge, is the knowledge that is in the heads of people. Explicit knowledge is tangible knowledge; it is written down or modeled in some document or source code. One of the challenges of software architecture, at the moment, is to document this tacit knowledge to make it explicit instead of implicit knowledge.

AKM becomes more and more important as organizations want to improve their architectural capabilities. Already several tools exist that specialize in AKM, Peng *et al.* gives an overview of them [36]. In his paper, he gives an overview of the available tools for AKM and from these tools and trough research he defines use cases needed for developing AKM tools. Some of these use cases are implemented in the available tools while others are desired by researchers in the field of AKM.

1.1.2 Architectural Decisions

Software architecture is much more that a set of models to describe a system, to construct those models architects make decisions. Decisions have contexts, assumptions, drivers, consequences, and considered alternatives. This knowledge remains tacit knowledge in the head of the architect, but it is an essential part of the architecture and architecting process, that needs to be documented. A developer is not able to explicitly derive these decisions from the architecture models unless they are documented. Architectural Decisions (AD) have been proposed as the missing link that bridges the gap between requirements, rationale, and software architecture design by capturing their implicit relations [20, 21]. When tacit knowledge is not documented, it is forgotten over time, this is called knowledge vaporization. Awareness and standards for documenting AK would prevent knowledge vaporization [22] and improve the traceability between the architecture and the requirements. The AD describe how the requirements are addressed in the design, this allows the reader to understand the rationale behind the design. Understanding the rationale behind the design allows for better reuse, adaptation, maintenance, and evolution of the architecture. Looking at software architecture in this way makes software architecture the result of a set of decisions [23].

1.2 AK Repositories

AK repositories are a specialized form of knowledge repository. They store AK allowing users to store and retrieve the AK inside. The repository consists of an ontology, which describes the knowledge domain of the knowledge that is stored inside the repository. There are various formal languages to encode an ontology. The Web Ontology Language (OWL) is the most frequently used and is defined by the W3C [33]. OWL can be used to build ontology's; it can explicitly represent the meanings of terms in vocabularies and their relationships [34]. It has build in facilities for expressing meanings and semantics which go beyond the power of XML, RDF, and RDF-S. OWL does use the RDF (S) syntax but extends it when needed.

OWL has the power to build an ontology for a specific domain, and to inference additional data from the provided data. This inferencing system, also known as truth maintenance system, uses the domain model to inference the additional data. And fits perfectly into the vision of the Semantic web (web 2.0) in which information is given explicit meaning, making it easier for machines to automatically process and integrate information available on the Web.

1.3 GRIFFIN Project

The GRIFFIN (a GRId For inFormatIoN about architectural knowledge) research project focuses on the capturing, sharing, (re)using, and managing AK during the architecting process. To do so the GRIFFIN project develops notations, tools, and associated methods to extract, represent, and use architectural knowledge that currently is not documented or represented in the system. The project emphasizes sharing architectural knowledge in a distributed context. Consequently, the project will also devise tools and infrastructure to do so in an integrated way [32].

The Griffin Repository

As envisaged by Zhuge [24], "Modern communication facilities, like the Internet, provide people with unprecedented social opportunities for knowledge generation and sharing". Zhuge designed a knowledge grid to improve the generation and sharing of knowledge. The aim of the GRIFFIN project is to create a knowledge hub for professionals involved in the software architecture process. For this case, the GRIFFIN project created the GRIFFIN toolsuite. i.e. the Knowledge Architect.

The Knowledge Architect

The GRIFFIN project created the Knowledge Architect (KA) to support the architecting process [29]. The Knowledge Architect (KA) is an AKM tool suite for capturing, using, translating, sharing, and managing AK [36]. At the center of the tool suite is the Knowledge Hub, an AK repository, which is accessed by different clients [29]. Semantic Web technologies are extensively used throughout the tool suite to allow for formal AKM. The tool suite is created iteratively according to the needs and feedback of the case studies performed at Astron [31]. The KA supports the following use cases defined for AKM [36].

1. View AK and their relationship
2. Trace AK
3. Share AK
4. Elicit/Capture AK
5. Integrate AK
6. Translate AK
7. Add/Store AK
8. Edit AK
9. Search/Retrieve AK
10. Check completeness of AK

At the moment the KA consists of one server and five client programs [29]. The server is the knowledge hub; also known as the GRIFFIN repository. Here all AK is stored and interfaces are provided for the clients to store and retrieve data. The Document Knowledge Client is a MS Word client that allows capturing, storing and retrieving of AK inside MS Word documents. For analysis models, similar tools have been developed, the Excel and Python clients, both capture, store, and retrieve AK inside, respectively, excel or python quantitative analysis models. The Knowledge Explorer visualizes the AK and their relationships. It provides various visualizations to analyze the data inside the repository. The last client tool of the KA is the Knowledge Translator. It can translate formal AK from one domain model to another domain model. The tool manually or automatically makes a mapping between the two domain models and translates it.

2 Problem Statements

The first thing that was developed for the AK tool suite was the Knowledge Hub (repository). Following was the development of the Document Knowledge Client which was connected to the hub. Other case studies brought the need for further development of different tools for the KA tool suite. During these developments it became obvious that the current repository was not able to provide the required stability. The repository proved to be stable enough for the initial case studies, but due to increased usage it started to show problems. The repository seemed to possess some errors and crashed on some queries. Also initial signs of performance issues where emerging, through the growing amount of data inside the repository. From experience the first problem statement was formulated:

PS1: The current repository is not mature enough.

From this problem statement four research questions can be derived.

RQ1 What are the maturity problems and what is causing them?

RQ2 What are potential solutions to address the aforementioned causes?

RQ3 What potential solutions to use?

RQ4 Do the solutions solve the maturity problems?

Another case study in the GRIFFIN project revolved around tracking the evolution of software architecture documentation (SAD). The aim was to see how software architecture documentation changed and evolved over time. To facilitate this, the knowledge from and changes to the SAD needed to be stored somewhere inside the Knowledge Hub and the AK tool suite should be able to work with it. This lead to the following problem statement:

PS2: The repository is not able to store the evolution of a software architecture documentation.

From this problem statement six research questions can be derived.

RQ5 Why is the repository not able to store the evolution of a software architecture?

RQ6 How to store evolution of a software architecture in a knowledge repository?

RQ7 What are the requirements for tracking the evolution?

RQ8 What are the potential solutions for tracking the evolution?

RQ9 Which solution is the best one?

RQ10 How to validate whether the chosen solution meets the requirements?

In the following chapters the defined research questions will be addressed. The first three chapters address problem statement one: Chapter three addresses the problems that the current repository has and the requirements that follow from these problems (RQ1). The next chapter,

four, reviews the potential and the chosen solutions to the problems (RQ2 & RQ3). Chapter five gives a conclusion and answers RQ4, i.e. do the solutions solve the maturity problems. The next five chapters address problem statement two. The second part of this thesis addresses problem statement two. First chapter six describes why the current repository is not able to store architectural evolution data, and what is needed to store that data (RQ5, RQ6). Chapter seven presents the requirements for tracking the evolution of software architecture (RQ7). The potential solutions for this problem are in chapter eight (RQ8). Chapters nine presents a quantitative evaluation of the potential solutions and chapter ten does a qualitative evaluation using a framework (RQ9). And chapter eleven validates the chosen solution by building a prototype (RQ10). In chapter twelve the related work for this thesis is presented. Finally, chapter thirteen presents the conclusion of our work and future directions.

3 The Knowledge Hub

At the start of the project the knowledge hub was in its early stages of development. The further developing of tools to add to the AK tool suite showed problems on different levels in the knowledge hub. The hub proved to be prone to errors, had badly written code, and slowed in performance due to increasing amounts of data and higher demands. The current repository was not deemed mature enough to handle all the needs. To identify the problems a good understanding of the current repository was needed.

3.1 The existing Knowledge Hub

The existing Knowledge Hub was developed incrementally as different case studies needed different kind of the tools and functionality. A the start of the project it was decided to use Sesame as the central repository of the knowledge hub. Sesame is an open source Java framework for storing, querying, and reasoning with RDF and RDF Schema [7]. It is a database for RDF or RDF Schema and provides a framework for applications to implement RDF internally. Different data stores can be connected to Sesame, initially the choice was made for MySQL due to its scalability and easy implementation. Figure 1 shows the existing knowledge hub as it was build supporting the first case studies. On the left we see Protégé, Protégé is a free, open-source platform that provides a suite of tools to construct domain models and knowledge-based applications with ontology's [37]. The GRIFFIN project used protégé to build their domain model in OWL format, which could be inserted into the Sesame repository. This OWL file is inserted through the Sesame web interface that Sesame provides for apache tomcat. The client allows the uploading of OWL files, explore, export and insert the data inside the repository. On the right side of Figure 1 we see a Sun Application Server; this server is used to host the Knowledge Server Implementation (KSI). The KSI is developed by the GRIFFIN project it was a layered java application that provided the connection with Sesame, the logic to handle the data for insertion and retrieval, and an interface that could be accessed by client programs. Jax-ws was used to make this possible, Jax-ws is a Java API for XML web services, it provides an XML interface which allows for connecting remote clients [38]. The choice for Jax-ws was needed to connect the Document Knowledge Client (DKC) to the knowledge hub. Since the DKC was programmed in .NET and Sesame is written in Java web services are ideal to connect different technologies and languages. The other client that was connected to the Knowledge Hub was the Knowledge Explorer.

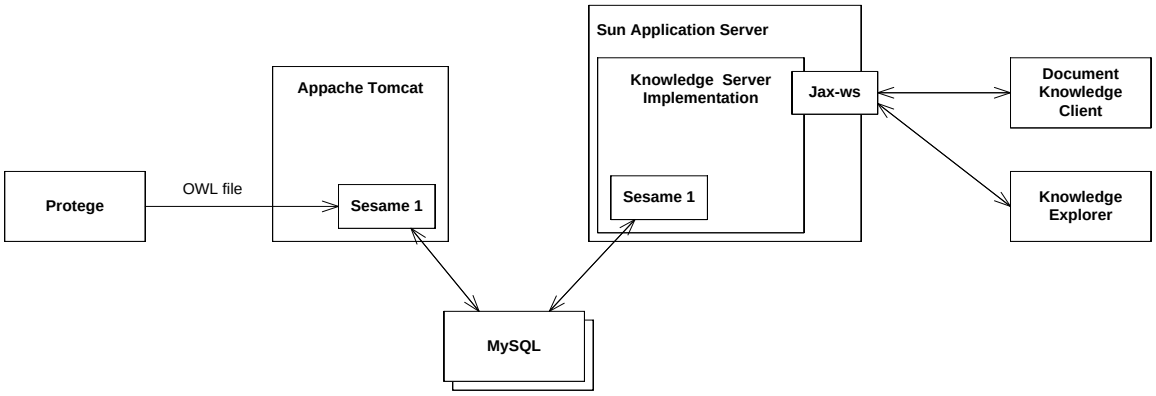


Figure 1 The existing knowledge hub

Internally Sesame is able perform RDF Schema inferencing. Inferencing is one of the new technologies that the Semantic Web offers, it is like automated deductive reasoning, it

deduces (inferres) new information from information that is already known. Sesame comes with its own basic inferencing engine, it infers new data from the data that is inserted into the repository and the data already present in there. The initial choice was to use this inferencing engine in the first system.

3.2 Requirements of the Knowledge Hub

The existing system, as described above, fulfills some of the use cases as seen in section 1.3.2, others are not supported because the tools, for those use cases, where not developed at that moment. At this point the only interest lies with the use cases that involve the Knowledge Hub itself and not the connecting client programs. These are the following use cases:

1. Share AK
2. Integrate AK
3. Add/Store AK
4. Search/Retrieve AK
5. Check completeness of AK

So what we expect from our Knowledge Hub is that it allows for storing and retrieving (sharing) AK, it integrates this knowledge based upon a common domain model. For these use cases the following prioritized requirements where defined:

1. Performance
2. Reliability
3. Transparency
4. Maintainability

Performance
Performance was identified first, without a good performance users would never use the system. While working with the system the users should have a good response time of the system. This is why performance has the highest priority, because if no one wants to work with the system, it is useless.

Reliability
Reliability ensures that the system handles errors correctly instead of crashing and data inserted or retrieved from the system may not be corrupted. A high reliability is needed. Corrupted data is of no use to anybody and a error prone system is never used. Therefore this property gets the second priority.

Transparency
Transparency had the third priority; it means that users do not need to have knowledge about the system to be able to work with it. The provided tools must be easy and intuitive to use. Secondly it means that the system must also be transparent for developers, it must be clear to see where to adapt the system and what parts of the system provide which functionality. If these requirements fail, users will not be able to work with the system correctly and developers will have a hard time developing or adapting the system.

Maintainability
Maintainability was identified, because the development for new tools for the KA tool suite is an ongoing process. It stands to reason that the Knowledge Hub also has to be adapted or evolved from time to time. Therefore a good maintainability is needed, it ensures that code is well written, structured and documented and the architecture of the Knowledge Hub allows for easy integration of new clients.

3.3 Maturity problems of knowledge grid

At the start of the project the requirements where defined. Later the first knowledge hub was build and it fulfilled those requirements in most ways, performance and reliability where good. The transparency was ok but could be improved. Maintainability was poor and implementation was good, since most technologies used where off the shelve components. Also the querying mechanism in Sesame was very good. All together the first knowledge hub fitted the demands needed from the system. The problems started emerging when more tools where developed for the KA tool suite and more complex queries where required. Three maturity problems (MP) where identified:

MP1: Error prone code
The first problem emerged after the need for some complex queries. Data was corrupted and the system was not able to handle errors. The indentified cause was the poorly written code in the knowledge server implementation. In total about 30 issues were identified. This violated the reliability and maintainability requirements and in the long run could also affect the performance.

MP2: Faulty data
Secondly after a thorough inspection of the inserted data, some of the expected inferenced data was not present. There were three possible causes identified; the OWL model build in protégé was not complete, The inserting code contained errors, and the Sesame inferencing engine was not good enough. This posed a serious problem to our reliability constraint.

MP3: Complex queries crash MySQL
With one of the new case studies there was a need for very complex queries. These queries degraded performance and some crashed the MySQL database. The where two possible causes identified; the first was that the queries where formulated wrongly, and secondly that MySQL just was not able to cope with such complex querying. This problem also relates to the reliability but it also affected the performance.

4 Maturing to the knowledge hub

In the previous section, the maturity problems and their possible causes to the existing server where identified. This section explores those causes, seeks, and implements the solutions to results in a more mature knowledge hub. The improvements to the hub where done in an iterative way following the identified problems. Figure 2 shows the iterative steps that were taken. In section 3.3, three problems were identified but there are four iterative steps; this is because the third step raised a problem that is addressed in the fourth iteration step.

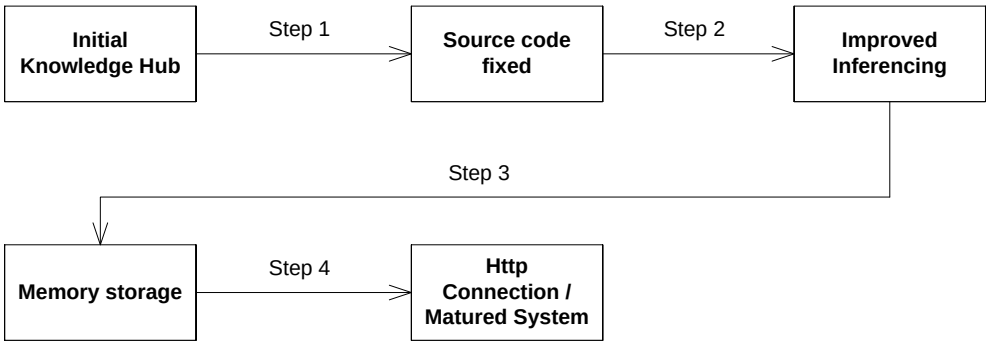


Figure 2 The Iterative Maturing Steps

4.1 Step one: Error Prone Code

The first identified problem of the Knowledge Hub was the error prone code. After a thorough review of the available code several problems were identified:

- Methods not always produced the expected result;
- The system was not able to handle errors and crashed;
- The code was devided into three layers; database connection, logic and interface, but the different roles of the layers where intertwined;
- There was almost no documentation in the code.

The first implemented solution was the development of unit tests. Trough these tests problems in the code could be identified and addressed. The unit tests constantly check if the methods produce the expected results, positive and negative unit tests were used. The code was refactored and rewritten until all unit tests succeeded. To enable the system to handle errors java exceptions were added to the code. Try and catch statements intercepted the errors and gave the user a proper notification of what went wrong. Through the catching of errors the system stayed in running order after an error instead of crashing. Next the code was restructured, all methods and source code was moved to the correct layer to improve the overall understanding of the code. During all the previous fixes to the code Javadoc was added to the fixed areas and the rest of the code. Figure 3 shows the Knowledge Hub, the structure did not change. The red highlighted area shows the part of the hub were all the problems were located and fixed.

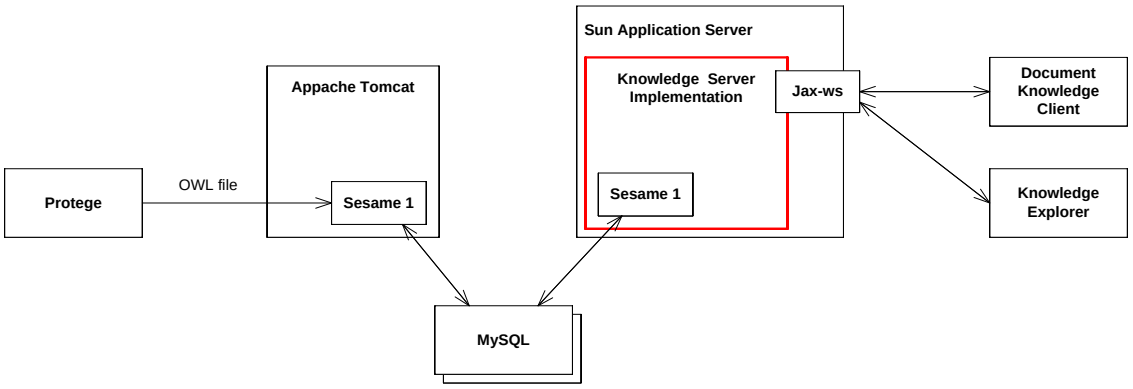


Figure 3 Result of Step One

4.2 Step two: Faulty Data

After inspecting the data inside the repository through the Sesame web interface provided by Tomcat, we found that some data was not inferenced correctly. Three possible causes where identified for this problem. The first was a possible fault in de code. The inserting queries where tested through the web interface, they seemed to be working correctly. The second possible cause was a fault in the OWL file. The OWL file was reviewed and using protégé some fine tuning was done, it made some improvements, but still faulty data emerged. The last cause that was considered was that the sesame inferencing engine was not robust enough. Sesames architecture allows for attaching different inferencing engines. The default Sesame inferencing engine only allows RDF Schema inferencing but cannot handle full OWL, but it was thought to be sufficient at the start of the project. After reviewing several inferencers, the best replacement seemed to be OWLIM an inferencing engine that can be connected to Sesame that allows for RDF Schema and OWL inferencing. Figure 4 shows where the OWLIM inferencing engine was placed.

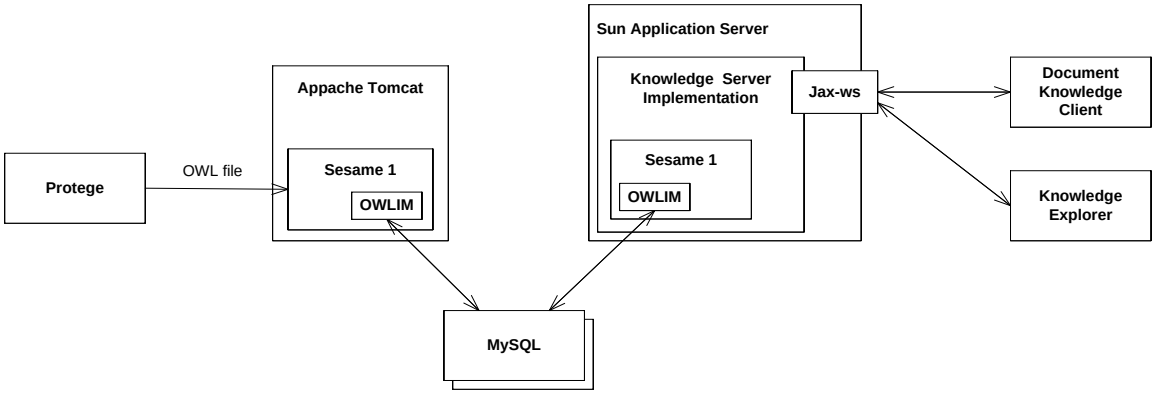


Figure 4 Upgrade With The OWLIM Inferencing Engine

4.3 Step three: Complex Queries Crash MySQL

The development of new tools for further case studies demanded complex queries on the data inside the repository. MySQL seemed to crash on these queries. There were two possible causes identified, the first assumption was that the involved queries where formulated incorrectly and the second was that MySQL was just not able to cope with such complex querying. After validating and optimizing the queries it was obvious that the fault lay with MySQL. Sesame offered the possibility to store the data in several ways, MySQL was chosen since it has good scalability. But it also proved to be a bit low on performance after inserting a few million triplets. We needed a replacement for MySQL with a faster repository that would

not crash on our queries, the choice was made to enable the Sesame Memory Store. Sesame has an integrated memory store that is optimized for triplet storage; it stores all the triplets into the memory, skipping slow mechanisms like hard drives. The memory store synchronizes to a local file storage to save the data on a shutdown. This implementation proved to work with our queries but meant a change in the system architecture. Figure 5 shows the new system architecture. Tomcat has been removed, since the memory store is not a shared repository and has to be initialized locally. It makes the architecture cleaner however it raised another problem. The sesame web interface is not accessible anymore so developers and users are no longer able to inspect the data inside.

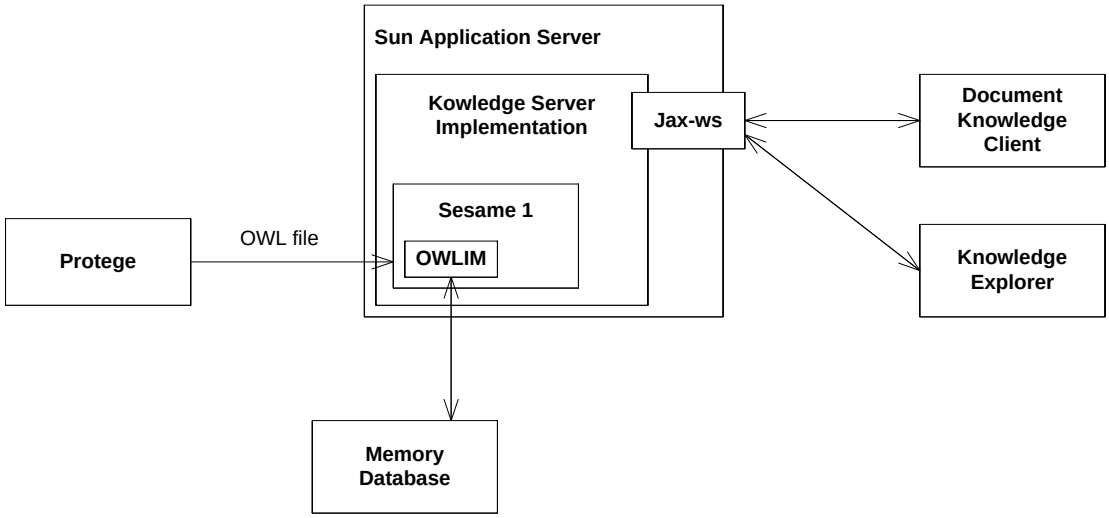


Figure 5 System With the Memory Store Upgrade

4.4 Step four: No Inspection Mechanism

The last step in addressing the maturity problems of the Knowledge Hub introduced a new problem in our Knowledge Hub; the ability to inspect data inside the Knowledge Hub through the Sesame web interface. For users this is no problem but it was for developers. To solve this problem the repository was moved back to Tomcat with the preservation of the previously introduced memory store. This was done using a HTTP connecting that Sesame offers. Figure 6 shows this final upgrade to the system. Sesame with an OWLIM inferencer and a memory store is loaded in Tomcat providing us with the web interface. The Application Server makes a HTTP connection to Sesame and forwards all requests to Tomcat.

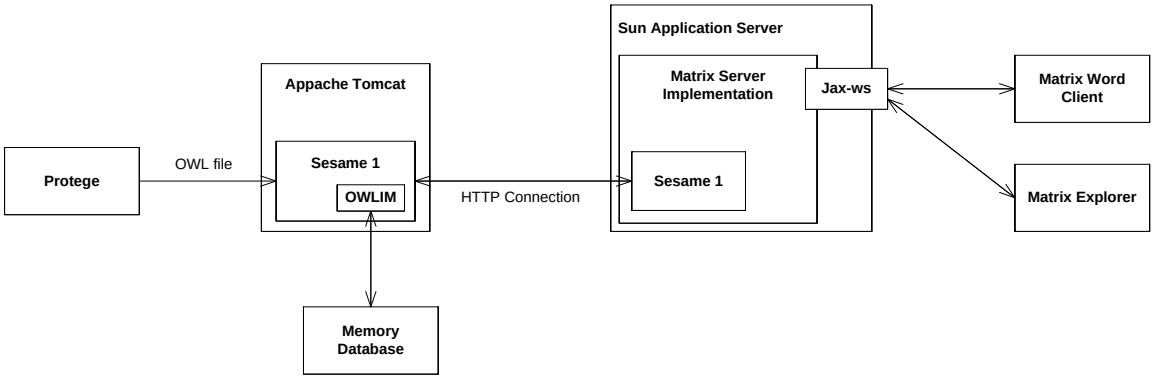


Figure 6 The Final Matured System

5 Maturity Evaluation

The final research question connected to the first problem statement was, did the upgrades solve our maturity problems. In this sections the implemented solutions, chapter four, and their impact on the requirements is evaluated.

The first problem, MP1, see chapter 3.3, was the error prone code which had a very negative impact on the maintainability, the transparence, and reliability. The upgrade of the code solved those problems. The implementation of the unit tests ensured a better maintainability since every test can be run again after changing the source code to check if everything is still in working order after the changes. Then the exceptions where implemented to improve reliability, good error handling and avoid the crashing of the system. The division into a three layered structure greatly improved the transparency for developers. To improve the performance some of the queries where combined, each time a query is run the inferencer was triggered and this slowed the repository. The combining of queries improved performance. And finally adding documentation throughout the code improved its maintainability and transparency.

The second problem, MP2, see chapter 3.3, in the Knowledge Hub was the faulty data after insertion. Several causes where identified but the final solution proved to be the implementation of the OWLIM inferencing engine. The implementation of OWLIM improved the reliability of the data and as a side effect also boosted the overall performance. The review and optimization of the existing queries also boosted the performance slightly.

The final identified problem, MP3, see chapter 3.3, was that complex queries seemed to crash MySQL. The cause seemed to be MySQL which could just not handle large amounts of data and those complex queries. MySQL was switched for the Sesame Memory store. The implementation of the memory store did not only improve the reliability but also made a significant boost in performance. The memory store is optimized for triplets, which MySQL is not. In the old system deleting took about 50 seconds, the memory store boosted that to 0.5 seconds, a factor 100 performance increase. On the downside the ability to inspect the data present in the Knowledge Hub through the web interface was lost, sacrificing maintainability. To counter this problem the HTTP connection was implemented. It sacrificed a bit in transparency for the developers but the improved maintainability was deemed more important.

Looking at the overall picture the implemented solutions solved the maturity problems and the Knowledge Hub is much more mature in its final form. Through the upgrades not only the identified problems with their related requirements where solved but also a significant gain in performance was achieved.

When looking at the future of the system we can conclude that it has become much more transparent, through the HTTP connection multiple clients can connect directly to the Knowledge Hub skipping the resource consuming Application Server. The Document Knowledge Client is the only one that needs the Application Server due to interoperability between .NET and Java. The Knowledge Explorer at the moment is connected through the Application Server but can be adapted to work with a HTTP connection.

6 Storing Architectural Evolution

The knowledge architect tool suite is in ongoing development, it is driven by needed functionality of different case studies. One of the desired case studies was tracing of the evolution of software architecture documentation. To see how an software architecture documentation evolves over time could give insights into the architecting process. The idea was to store the evolutionary data inside the Knowledge Hub. However the existing repository was not able to store this evolutionary data, this lead to the formulation of the second problem statement: "*The repository is not able to store the evolution of a software architecture*". This chapter goes into the first two research questions that followed from that problem statement: RQ5, "*Why is the repository not able to store this data?"* and, RQ6, "*how to store this kind of data*?". Before we can answer these questions we first need a better understanding what evolutionary data is.

Architectural evolution can be seen as the evolution of the software document. At the start of the architecting effort the architect makes an initial document which is refined (evolves) over time. This initial document changes: parts are added, rewritten, changed, or even removed. To store the evolution of the document all those changes have to be stored somewhere. So why is the Knowledge Hub not able to store this evolutionary data (RQ5)? First let's take a look at the inner workings of the repository i.e. Sesame.

6.1 RDF

The core of the Sesame repository is RDF. RDF stands for Resource Description Framework and it is the basis for the semantic web. RDF was created in 1999 as a standard, which extended XML for encoding metadata. A later released version of RDF provided the means to create relations between things in the real world: People, places, concepts, etc. An RDF statement is composed of three parts, Subject, Predicate, and Object, also called a triple. Every sentence describing things can be decomposed into a triple,. For example if we take the sentence, "Hubert is studying computer science" it translates to the following triples: The subject is Hubert, the predicate is studies, and the object is Computer science. Everything in an RDF repository is modeled using statements. RDF ontology's describe these relations between high level things so applications know how to interpret them. Later RDF Schema (S) was introduced it extended RDF with classes and allowed the user to specify a domain or range for predicates. Another extension to RDFS was OWL, this allowed the user to specify, symmetric, transitive, functional and inverse properties to predicates. These properties allow computers to interpret the available data even better and with the use of inferencing generate new data.

A software architecture document, also called an Artifact, consists of many separate pieces of knowledge we call Knowledge Entities. In RDF, this oversimplified domain would be described by the following triples:

Subject	**Predicate**	**Object**
Artifact	owl	Thing
Artifact	documentName	String
Artifact	consistsOf	KnowledgeEntity
Artifact	ID	integer
KnowledgeEntity	owl	Thing
KnowledgeEntity	ID	integer
KnowledgeEntity	partOf	Artifact

The table shows that an artifact consists of Knowledge Entities and has three other properties one of which is owl Thing, which is a default property for OWL. It has an ID and a documentName. The KE has the same reverse relation in that is part of an Artifact and has also a owl Thing property and it has an ID. Of course in a real scenario there would be much more properties and relations between more classes than just Art and KE. But for this example this is enough. When this is loaded into the repository we can insert the following triples.

Subject	**Predicate**	**Object**
Art_123	*type*	*Artifact*
Art_123	*ID*	*123*
Art_123	*documentName*	*Example.xls*

After committing the previous triple, the repository knows, though the first triple that Subject Art_123 is of the type Artifact. Then we will insert the following triples

Subject	**Predicate**	**Object**
KE_432	*type*	*KnowledgeEntity*
KE_432	*ID*	*432*
KE_432	*partOf*	*Art_123*

The first triple again makes from Subject KE_432 a Knowledge Entity. The second triple sets its ID property and the final triple connects the KE to the object Art_123 which the inferenecer knows is an artifact. And through the rules described in the table above the inferencer creates the following triple in the repository.

Subject	**Predicate**	**Object**
Art_123	*consistsOf*	*KnowledgeEntity_432*

Of course all sorts of triples can be inserted into the repository, but the inferencer cannot do anything with them since it has no knowledge of their domain. We can now query the repository and ask it, which knowledge entities are connected with Art_123. Through the inferenced triple the repository knows that it is KE_432.

6.1.1 Inferencing

Inferencing is also called truth maintenance; it is a process that draws conclusion through applying rules. The conclusion that the system draws is called an inference. There are two types of inferences that can be made, deductive and inductive, Sesame's inferencer is deductive. So for the scope of this thesis inferencing can be seen as deductive reasoning. A simple example suffices to see the working of our inferencer. Say we have the following two statements:

- All knowledge entities are objects
- KE_432 is a knowledge entity

We know now through deductive reasoning that KE_432 is an object. The first statement "All knowledge entities are objects" is one of the rules that is contained in our OWL file. The second one is a typical statement that can be inserted into the repository. When the inferencer is triggered it creates a new statement that says KE_432 is an object and stores that into the

repository. This is a very simple example but the workings of the inferencing engine are clear. All statements that are inserted are validated against the rules of the model and are defined in our OWL file.

6.2 Unable to store architectural evolution

Looking at the basics of RDF, it is composed of triples, namely: Subject, Predicate, and Object, it must be clear that we cannot assign a versioned property to a single statement; there is just no room for it. The other problem is that our OWL is also build out of triples, we could assign a version property to the classes in our OWL model, like the artifact or knowledge entity class, but we would be unable to query for them since everything is still stored in triples. Example: The Artifact class has a predicate version that contains its version and it has a predicate hasText that contains some text. The following statements are then added to the Artifact model.

Subject	**Predicate**	**Object**
Artifact	*version*	*Integer*
Artifact	*hasText*	*String*

Lets assume there is an Art_123 is inside the repository and it's version is three. The repository would then contain the following triples.

Subject	**Predicate**	**Object**
Art_123	*type*	*Artifact*
Art_123	*ID*	*123*
Art_123	*documentName*	*Example.xls*
Art_123	*version*	*3*
Art_123	*hasText*	*"original text"*
Art_123	*hasText*	*"changed text"*
Art_123	*hasText*	*"rechanged text"*

It can be clearly seen that the artifact is in version three, but which triple is assigned to this version can never be determined. Single triples have the same problem since it is a triple it cannot be assigned an extra property like a version number. A query requesting the text of Art_123 will have three results instead of the desired one result, which is the latest change of Art_123. We could remove the older versions of the text, but then they could never be retrieved. Therefore the current repository is not able to store versioning information.

6.3 What is needed to store the architectural evolution

The need of the GRIFFIN project and the solution to the second problem statement is versioning. So how can the Knowledge Hub store evolutionary data (RQ6), by adding functionality to store versioning data or integrating an existing versioning system. In the following chapters the search for such a system to integrate into our knowledge hub is described. First, in chapter seven, we address the requirements needed for such a system (RQ7). Chapter eight then reviews the potential solutions for such a system (RQ8). In chapter nine and ten, a quantitative and qualitative evaluation is performed to see which solution is the best (RQ9). Chapter eleven evaluates the chosen solution with respect to our requirements

from chapter eight (RQ10) chapter twelve discusses the related work and we conclude our work and elaborate on future work in chapter thirteen.

7 Requirements for storing evolutionary data

In the previous chapter, it was made clear that the current repository is not able to store the evolutionary data. To enable the repository to store the data there was the need for some sort of a versioning system. To select the needed system, a good understanding of the requirements is needed (RQ7). The basic system at least supports storing and retrieving of (versioned) data. Branching, merging and locking are optional and not required for the initial system. In section 3.2 the use cases for a Knowledge Hub where stated, some of these original use cases have to be adapted somehow to fit the versioning system. Sharing, Integrating and the Checking of the completeness of AK stay the same. The ones that change are the storing and retrieving of AK. From these use cases the following functional requirements of the system are defined:

- Storing
- Retrieving
- Implementation
- Querying

Storing
The original use cases already provided this requirement, but the versioning part of the system needs some more specifics in the storing functionality. Not only must the Knowledge Hub be able to store this information, but it must also provide means to automatically version the data that is stored inside.

Retrieving
As was the case with storing, also retrieving is seen in a different way when there is a versioning system in place. The standard way to retrieve data stayed the same, it retrieved the data of the current version. But functionality must be implemented to allow for the retrieval of older revisions. The user must be able to retrieve older versions of complete artifacts and of single knowledge entities.

Implementation
Implementation was identified as a property to keep in mind the available tools instead of developing to much own code. Implementation refers to the effort it will take the developer to implement the solution. Is there an off the shelf piece of software that can be used? Is there one place in the system that has to be adapted, or do we need to change and add code all over the system?

Querying
The final property was querying, it is the means of communication with the collected data. Is there a query language available to communicate with the repository and access our data? Or does the developer needs to program his own code? Does the query language allow for a simple query procedure or is there a need for a difficult and adapted process? A good querying mechanism ensures that queries are validated, and is powerful enough to retrieve only the desired data instead of having to parse the retrieved data afterwards.

In section 4.2, the non functional requirements for the Knowledge Hub where defined. These requirements, of course, still apply to the entire system. The final system should also fulfill these requirements. The potential solutions must then confirm to the following set of requirements:

1. Storing
2. Retrieving
3. Performance
4. Reliability
5. Implementation
6. Querying
7. Transparency
8. Maintainability

The first two properties; storing, and retrieving are a must for the system for without them the system will never function. After that, performance is the most significant property and on the second place is reliability. A system that has a bad performance and is not reliable will never be used! The other properties are listed in order of importance.

8 Potential Solutions

Now that we know the requirements, potential solutions can be sought that fulfill this set of requirements. The first part of this chapter gives an overview of those potential solutions. The first one is OMM, an extension for Sesame that tracks changes. Secondly, DBMS's are considered, Sesame by default can use a few of them. Finally two custom solutions are introduced, one adapts existing versioning systems like SVN the other is a custom build into the Sesame architecture. The second part of the chapter reviews the potential solutions in depth.

8.1 The Solutions

Since ontology's and RDF(S)\OWL are a rather new and emerging field within software engineering there is not much software written for them at the moment. Most software is still in alpha or beta stage, and the available software is still at the early stages of development with still lots of features to add. The ideal solution is a system that integrates versioning for us with Sesame or extends Sesame. OMM is such a tool, which stands for Ontology Middleware Management and extends the Sesame repository with the ability to track changes. Next to OMM Sesame can store it's data inside DBMS's, namely MySQL PostgreSQL and Oracle, each of them can be equipped with versioning capabilities. Another solution that would require some customization are existing versioning systems like SVN, are they adaptable to work with Sesame. Finally a fully custom solution is proposed that extends the Sesame architecture with versioning.

8.1.1 OMM

OMM is an extension for the Sesame RDF repository; it supports tracking of changes, meta-information, fine grained access control and multi-protocol client access [10]. OMM's ability to track changes is a form of versioning, OMM stores the changes made to the triplets inside the meta-information, to be able to do this it upgrades the default Sesame triples with two extra properties. OMM creates an update ID inside the meta-information each time an insert or remove is performed on the repository. To retrieve older versions, OMM branches the current repository and calculates backwards to the selected update ID. Also there is the possibility to assign a version number to an update ID through the Sesame web interface!.

OMM seems like the perfect solution for our problem, it integrates with Sesame, it provides versioning functionality and is as fast as Sesame itself. But a closer look at the workings of OMM shows that it cannot be used. OMM is an extension to the Sesame triple store, in which it stores its adapted triples. But instead of versioning on triple or even object level OMM versions on the entire repository. Meaning that for a user it is virtually impossible to work with versioning, since for each version of an object or triple the state of the entire repository has to be calculated at that version. It is not only a very time consuming process, but the repository will contain more than one model. And OMM is not capable of distinguishing between models since it versions the entire repository. Adapting OMM is not possible since the idea behind the system, versioning on the entire repository, is fundamentally different from what we need, triple of object based versioning.

8.1.2 DBMS

Sesame can connect to various DBMS systems: MySQL, PostgreSQL and Oracle, these DBMS systems can be equipped with versioning systems. MySQL has the PBXT storage engine [42], PostgreSQL has Post Facto [43], and Oracle also has several version control

systems [44]. Looking at different performance test available online we can conclude that the performance of MySQL, Postgres, and Oracle is not far apart [45][46][47]. Some test conclude that MySQL is faster some say Postgres is faster, Oracle is always considered fast but only if you have serious hardware. The difference in performance between the three DBMS systems is at maximum factor two. In chapter 4.3, the MySQL DBMS was removed from the knowledge hub to solve some problems and improve the performance of the overall system, the result was that the default Sesame memory store was 100 times faster than MySQL. Thus integrating an DBMS with versioning capabilities would imply at drop in performance of 5000%. The conclusion was that hard disk based DBMS systems simply would not provide us with enough performance, which is our main requirement, thus ruling them out as potential solutions.

8.1.3 Sesame with version control system

A custom solution that uses a version control system to handle versioning, while Sesame is used for the most current version. The triples for each object are put into a separate file, those files will be inserted into the version control system, letting it handle the versioning.

8.1.4 Extending Sesame

The idea is to extend Sesame itself with versioning capabilities. To do that a new implementation of the Sesame Memory Store is developed that will handle the versioning. The versioning information is then stored inside the context, a fourth property added to the default Sesame triple that is introduced in Sesame two.

8.2 Scoping Solutions

The previous section started out with four possible solutions, OMM and RDBMS were ruled out. This section focuses on the remaining two solutions and provides in-depth views of both solutions. First a closer look at integrating Sesame with a version control system like SVN, secondly, extending Sesame with versioning capabilities.

8.2.1 Sesame with a version control system

Version control systems manage changes to documents, source code, and other files. Some well knows systems are CVS, SVN and GIT, but much more exist [40]. Most of these systems have a similar inner working; they calculate deltas for each updated or changed file that is inserted. A delta contains the information about the changes made to the file and through this information the older states of the file (version) can be calculated. All of these systems provide an interface to access and retrieve those stored versions of a file.

8.2.1.1 Using SVN as a versioning system

The idea was to equip Sesame with some form of versioning layer that directly connects to a version control system. The choice for SVN as a version control system was made because it is well know, developed, stable and has a Java API. The basic idea was to export the data into files and insert them into SVN letting SVN deal with the versioning. This is modeled in Figure 7, here the Knowledge EXtractor (KEX) parses all knowledge from the inserted models and creates a collection of files from them, next to that it inserts the knowledge into Sesame, updating the current version that is kept inside the repository. The collection of files is inserted into SVN, letting SVN deal with the versioning of those files. Other programs of the KA tool suite, like the Matrix Explorer (MEX), can then connect to the SVN to work with the versioned data. The data for the current version the MEX, and the other clients, retrieve Sesame, for the versioned data they need a connection with the SVN, through a custom layer.

The retrieved data is stored in a small local Sesame memory repository on which the MEX is able to perform the same queries as the remote repository. The MEX is then able to present the user with a visual representation of the current and versioned data. When working with the current version the user is able to update certain relationships that KE's have. This information is stored locally during the session of the user and when he commits these changes, they are stored within Sesame and the SVN versioning information is updated.

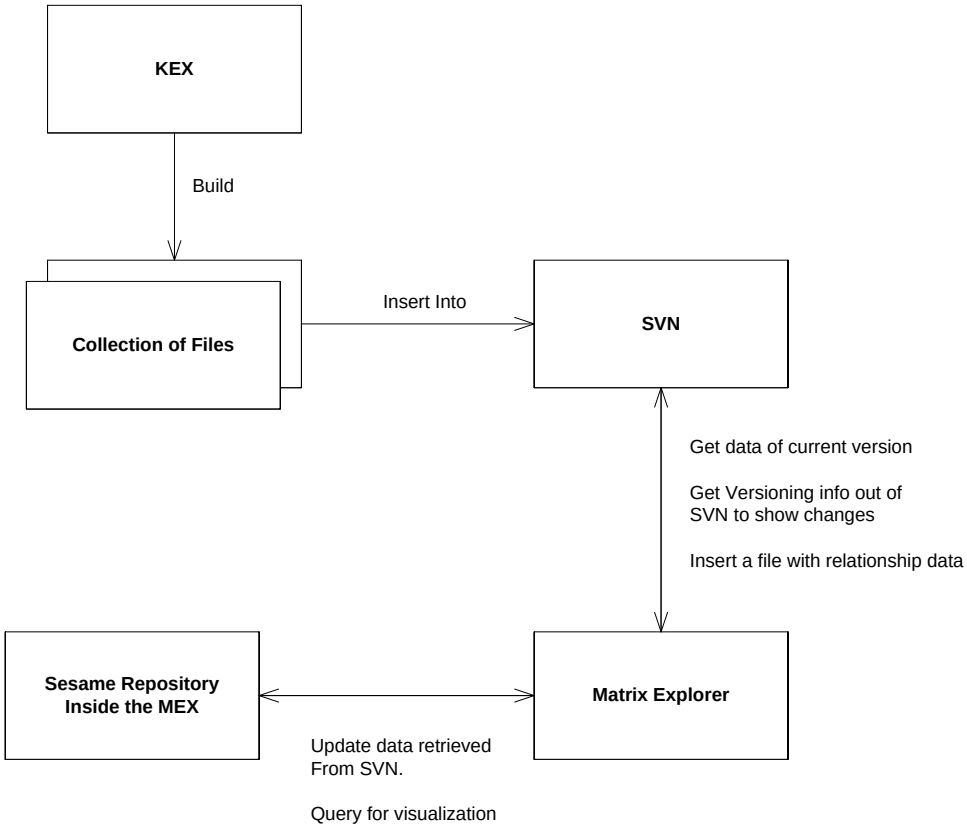


Figure 7 The SVN solution

8.2.1.2 Potential Pitfalls

Although the basic idea is good, and provides an easy way to handle versioning, it does have some potential pitfalls that have to be taken into account:

- When generating files from sesame we do not know in which order the triples are written to the files, it must be check if SVN can handle this instead of marking a file changed when it is only ordered in a different way.
- The generation of files is an IO process; IO considerably degrades performance.
- There is no query language for SVN. A custom versioning layer has to be developed for all clients.

8.2.2 Extending Sesame

The second custom solution was based on the current Sesame repository. Since the repository is already the center of the knowledge hub a direct integration of a versioning system would be the logical choice. The existing knowledge hub works on Sesame 1 which is a triple store, the solution proposed here works with Sesame 2 a quad store, it adds an extra property to the triples named context. This extra property is used to build a versioning system based on

triples within Sesame 2. To be able to explain this properly let's first take a look at the inner workings of Sesame 2.

8.2.2.1 The inner workings of Sesame

In Figure 8, the Sesame 2 Architecture is shown [41]. On the lowest layer we see the RDF Model; this model is the foundation of the Sesame framework. Because Sesame is an RDF-oriented framework, all of its parts are in some way dependent on this RDF model. The model defines the interfaces and contains implementation for all basic RDF entities.

On top of the RDF Model we find RIO, RIO stands for RDF I/O. It consists of a set of parsers and writers for the RDF file formats. These parsers are used to translate RDF files to sets of statements, the writers perform the reverse.

Next to RIO we find the Sail API, which stands for Storage And Inferencing Layer. This is a low level System API that is used for RDF stores and inferencers. The memory store that is used in the current system is an implementation of the Sail API, as is OWLIM.

The HTTPClient layer is used to communicate with a HTTP Server which is found in the upper layer. This allows for the remote HTTP connection that is used in our current Knowledge Hub.

The third layer is the Repository API, it is a high level API that offers developers a number of methods for handling RDF data. It offers methods for uploading data files, querying, extracting and manipulating data. This layer has various implementations, the two most common are the SailRepository and the HTTPRepository. Sail translates all calls to access the Sail implementation; the HTTP translates and transports those calls over HTTP.

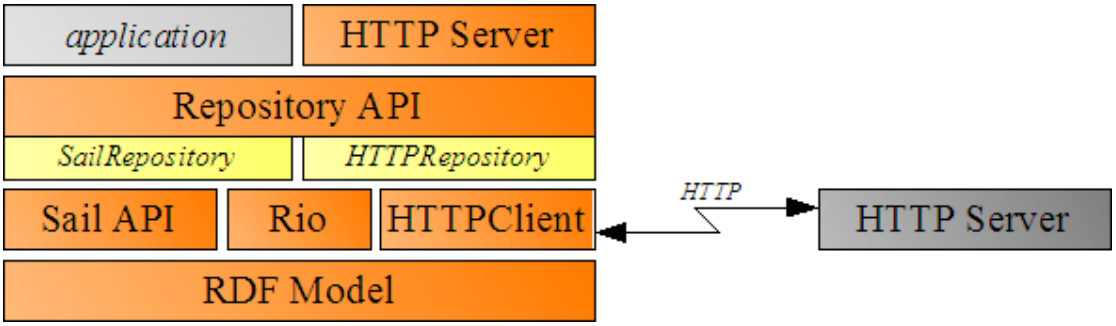


Figure 8 The Sesame 2 Architecture

8.2.2.2 Our Solution

In the current knowledge hub clients, like the Knowledge EXtractor (KEX), connect to the Knowledge Hub through a HTTP connection. The idea was to implement version control inside Sesame on the server side making it directly available to all clients. The challenge is not to harm the original working of the repository. The version control system will be a new implementation of the Sail API. A versioned remake of the existing "Sesame Memory store". The original SailRepository is left untouched hiding the implementation of the versioning systems for the clients. Figure 9 shows the proposed solution, the client application implements a HTTP connection to the Knowledge Hub where the data passes through the internal layers of Sesame.

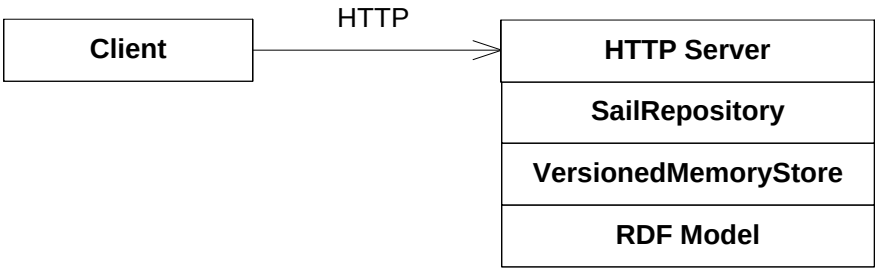


Figure 9 The Layers of the Proposed Versioning System

The SailRepository will give us the proper means to insert, update and delete (versioned) data. This layer will then forward these requests to the VersionedMemoryStore where the versioning system will be located. This versioned memory store will intercept all Insert and Delete operations that are sent to the repository and will use that data to construct *insert* and *delete* deltas in a similar way that versioning systems like SVN do. To store these deltas inside Sesame, Sesame was upgraded to Sesame version 2.

Sesame 2 uses quads instead of the triples to store the data. The original triple, containing, subject, predicate, and object, is replaced with a quad that adds a fourth property called context. The context part is what we are going to use to enable the repository to store our versioning information. Like SVN, Sesame needs to store the constructed deltas somewhere, with the use of contexts the repository is segmented into "four" parts, see Figure 10. The current context contains the current status of all the triplets inside the repository. The versioning context contains the administration part of the versioning system; it contains data that describes the version of all the objects and where the deltas of those previous versions can be found. These deltas are stored in the last two segments, divided into a delete and insert contexts, one context for each delta created.

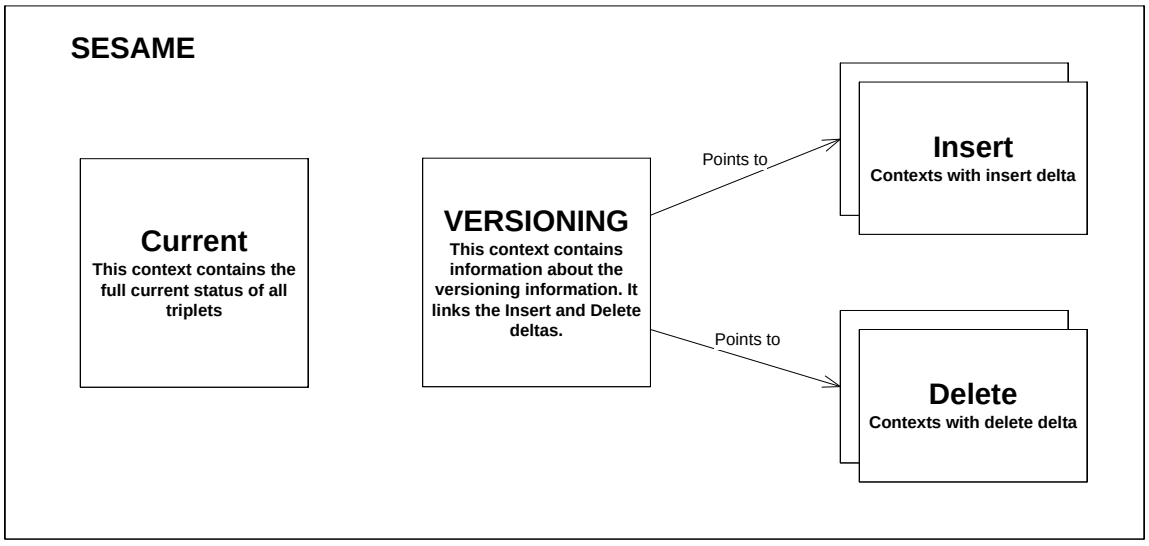


Figure 10 The context segments in Sesame

The impact client side is as small as possible, the client can access all data inside the repository through the default Sesame querying languages, SeRQL and SpaRQL. The client will probably get a layer that handles the connecting with Sesame and here extra queries have to be implemented that enable the client to query for versioned data. The final system will look something like Figure 11. The client has a layer which will handle all data, this layer communicates with the Sesame HTTP layer for accessing the Knowledge Hub. In the

knowledge Hub, the HTTP layer accepts the calls from the client application and passes them through Sesame's layers into the versioned memory store. These layers update the current context and the versioning context and create contexts for the insert and delete deltas.

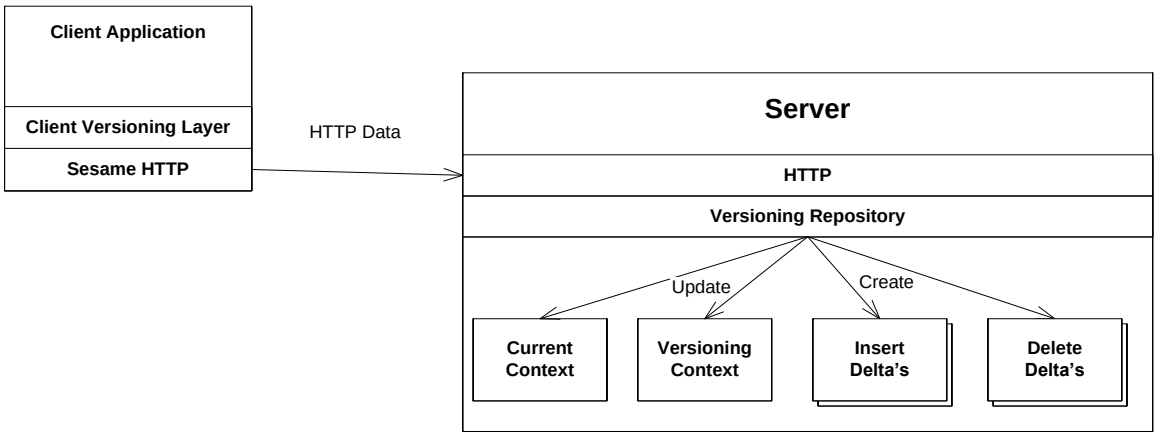


Figure 11 Final Proposed Versioning System

8.2.2.3 The required system

The implementation of this all requires also a change in the current Knowledge Hub which can be seen in chapter 4.4 were the result of the maturing of the existing system is concluded. This is also the starting point of the needed system for our versioning proposal. Figure 12 shows the required system, which in a lot of ways still looks like the system in chapter 4.4 with two exceptions. Sesame 1 is replaced with Sesame 2 through the need of the quadruplets. And more important the application server is removed. It is not needed since it was only required for the DKC. This leaves us with a system with the same specifications but with a better performance and an improved transparency since the slowing application server was removed.

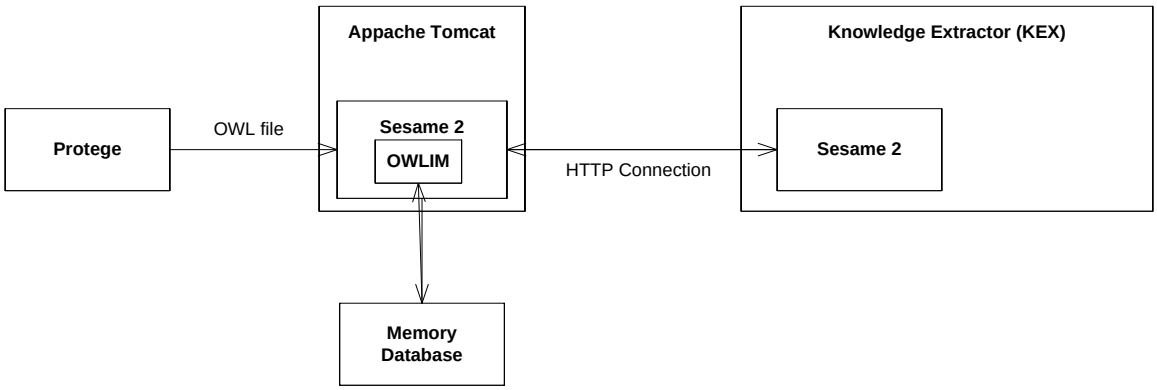


Figure 12 Final system for the start of the versioning project

8.2.2.4 Potential Pitfalls

The following pitfalls where identified and need to be taken into account:

- Duplicate Triplets
- Querying
- Locking Of Data
- Inferencing

Duplicate Triplets
When inserting new triplets into Sesame, the checks that no duplicate triplets are versioned have to be thorough. Sesame checks somewhere that duplicate triplets are not inserted, so here we must extend it and make sure that these triplets are not versioned. If this happens there will be a considerable amount of wrong deltas created.

Querying
The querying of the repository is very powerful and gives us lots of control over the versioned data. Since we can query over different contexts and construct sub queries, there is a lot of information that we can retrieve. Do we need to adapt the complete Sesame querying mechanism in order to let this solution work?

Locking of Data
How do we handle locking of the data? What is done when for example the following situation, Sam checks out some data from the repository, Lisa also checks out the same data. Lisa makes some changes and commits, then Sam makes some changes and commits. This is a classic problem that has two strategies to solve it. The first strategy is a lock-modify-unlock mechanism. The second strategy is the cope-modify-merge, both solutions are good and can be implemented fairly easy using transaction level locking.

Inferencing
How will the inferencer handle all the versioned data that is inside the repository? Since this data does not comply anymore to the original OWL model. Some checks need to be made how the repository deals with these things.

9 Quantitative Evaluation

In quantitative research the central process is measurement; the quantitative evaluation performed for this thesis focuses on measuring the performance of the proposed solutions. Performance was identified as the most important, measurable, requirement in chapter seven. Therefore it was decided to do a performance analysis of the proposed solutions. A thorough method for performance testing [11] was adapted to suit the needs of the testing effort. It provided a structured test plan, which consisted of the following steps:

1. Identify Test Environment
2. Identify Performance Acceptance Criteria
3. Plan and Design Tests
4. Configure Test Environment
5. Implement Test Design
6. Execute Tests
7. Analyze, Report and Retest

These steps follow a logical order, step one provides a better understanding of the environment that the test needs model, the focus lies with the user interaction, the non user initiated processes and the architecture of the system. In the second step, the metrics needed to measure the performance are captured. The third step, "*plan and design tests*", identifies the needed use cases based on the system interactions found in step one, from those use cases the tests are designed to mimic the production system as closely as possible. Step four describes how to configure the test environment, the closer the test environment matches the production environment the better the results. The fifth step focuses on the implementation of those designs; it describes the choices made needed to implement the tests. In step six the tests are executed, it describes how the metrics from step two are measured and collected. The final step, seven, analyses and presents the results found in step six. Retesting of all the tests is also done to validate the findings.

The chapter is divided according to these steps. Each step has its own sub chapter, and some sub chapters are then again divided into sub steps. This chapter gives an overview of the complete performance report which can be found in appendix I. The chapter ends with a conclusion on the found results.

9.1 Identify the test environment

Identifying the test environment explores and evaluates the system, to support and guide performance testing. According to [11]; "The intent of system evaluation is to collect information about the project as a whole, the functions of the system, the expected user activities, the system architecture, and any other details that are helpful in guiding performance testing to achieve the specific needs of the project". So evaluating the system on forehand gives a better understanding of the system as a whole and results in a better test design. It is essential for good performance testing. Evaluating the system is composed of the following activities:

1. Identify the user-facing functionality of the system.
2. Identify non–user-initiated (batch) processes and functions.
3. Determine expected user activity.
4. Develop an exact model of both the test and production architecture.
5. Description of the test system

The first steps, 1, 2 and 3 provide answers to the following questions: How do the users work with the system, what other processes are there and what is the frequency of these interactions with the system? With these answers better estimates of the load of the system and the impact of the accompanying use cases can be made. From these simple questions the basic architecture of the system was derived, Figure 13 shows the models of the test and production architecture, step 4. Users interact with the system from two different points, through interaction with the SVN and through the MEX. However the core of the system, marked with the red dotted line, shows us that the KEX and MEX are the only ones with direct contact with the knowledge hub. The test environment will then only model the retrieval and insertion of data from the knowledge hub automating all user input and SVN triggers. Step five describes the test environment; this can be found in appendix I

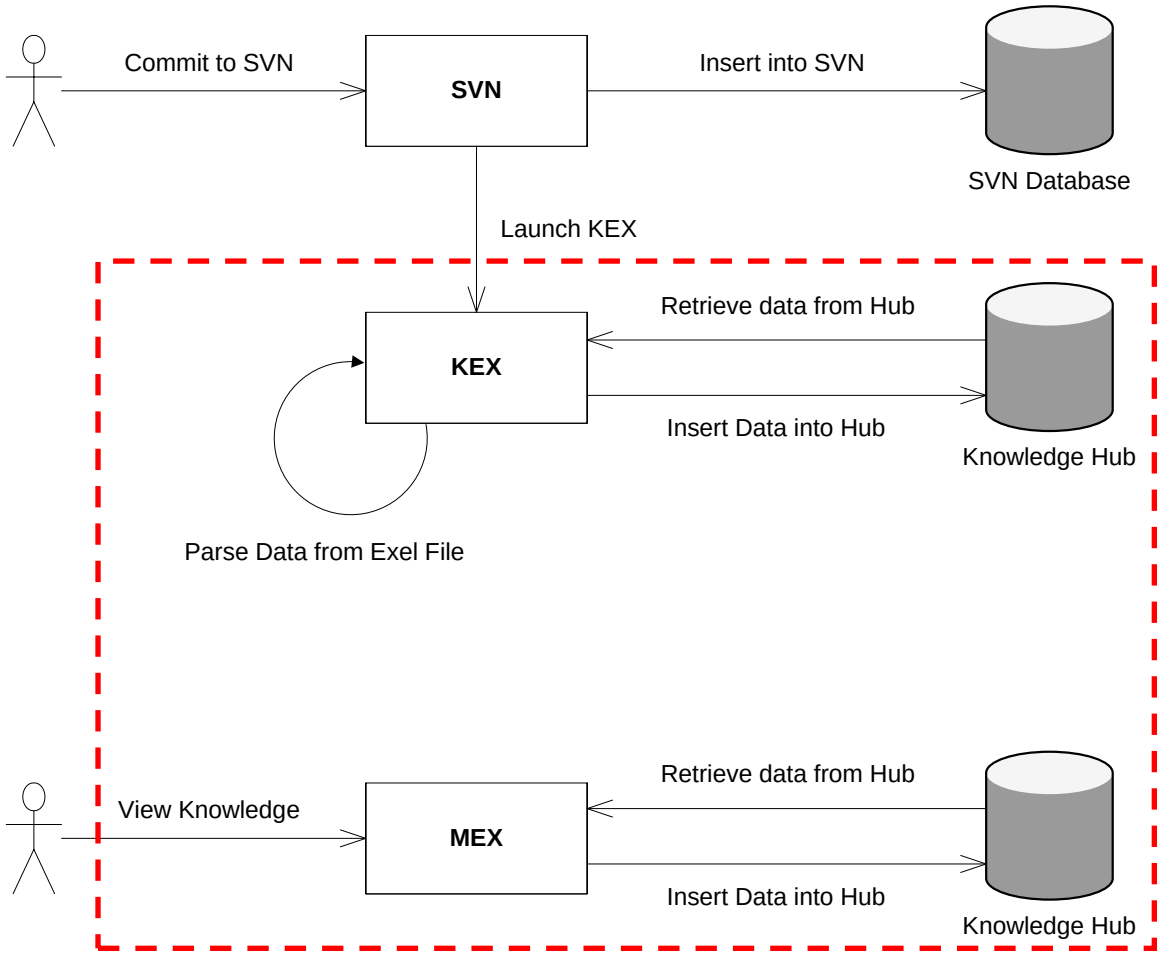


Figure 13 The System Architecture

9.2 Identify performance acceptance criteria

Identifying the performance acceptance criteria is necessary to get a better understanding of the critical areas in an application and to what criteria these areas must comply. It consists of the following tasks:

1. where do we want to get performance measurements from?
2. what criteria do apply to these measurements?
3. which metrics do we use for them?

Meier and Fare [11] describe them as following:

1. Determine the objectives of the performance-testing effort.
2. Capture or estimate resource usage targets and thresholds.
3. Identify metrics.

The objective, step one, of this performance testing effort, is to get a better view of the performance of the proposed solutions. These results will support the decision making process that chooses the solution that will be implemented.

Targets and thresholds, step two, are needed to check if the system operates within the given parameters and if the tests are executed within optimal circumstances. The idea is that a target identifies the optimal resource usage of a system, and a threshold indicates where the performance of the system is affected if the system reaches or crosses a threshold. For SVN the targets and thresholds for CPU usage will be set to respectively 70 and 80%. The I/O aspect of SVN is expected to be the bottleneck of the application. 70% of CPU usage or below is when a CPU functions optimal and will not affect the I/O bottleneck, thus the target is set to 70%. Above 80% the performance of a CPU will play a role in the overall system performance, therefore the threshold is set to 80%. Sesame does not have the I/O bottleneck that SVN has since most operations are in memory. Therefore we expect the CPU to be at 100% all the time. However available memory does affect Sesame's performance, the available memory is measured using the size of the data inside the repository minus the total available memory for Sesame. If the available memory for sesame is below 50% the performance degrades, below 75% it degrades significantly. The target for Sesame is to keep its memory footprint below 50% and a threshold is set at 75%.

To be able to quantify the performance these metrics need to be defined, step three. These metrics give us means to measure and compare. The above defined targets and threshold will be measured using the "Windows Task Manager". The memory usage of Sesame cannot be simply checked using the task manager, to measure it the entire contents of the repository is exported after each test, the file size of the exported file represents the actual size of the contents inside the repository. The most important requirement was performance; performance can be measured in time. The performance of the solutions will be measured in the total time it takes to complete one test.

9.3 Plan and Design Tests

With the metrics identified the planning and designing of the needed tests can start. The tests have to simulate the behavior that the system will have during normal operation. To let the test results reflect the final production system as closely as possible. If the test environment deviates from the production environment to much the results are useless. The process that is used to identify the usage profiles that will be used in the performance testing is known as *workload modeling* [11]. Workload modeling is done using the following steps:

1. Identify the objectives.
2. Identify key usage scenarios.
3. Determine individual user data and variances.
4. Determine the relative distribution of scenarios.
5. Identify application load distribution.

The first step is identifying the objectives of the testing process; it is always good practice to know what we want out of the performance testing effort. The global objective of the performance tests is to compare the different proposed solutions based on their performance.

To be able to do that, accurate measurements of inserting, updating and retrieving models inside the repository are needed. A secondary objective is looking at the impact of the data inside the repository and its effect on the performance, so estimates can be made about the scalability of the system.

There were six usage scenarios identified for the given objectives, step two. The first two are for the SVN solution and the others address the Sesame solution. Sesame supports three different methods that can be used for inserting of retrieval. Use cases 3, 4 and 5 test those different methods. The fastest of the methods is then used for use case 6. The usage scenarios identified where the following:

1. Committing a new model into the SVN
2. Updating an existing model inside the SVN
3. Committing a new model into Sesame using ELMO
4. Committing a new model into Sesame using a RDF-File
5. Committing a new model into Sesame using SeRQL
6. Updating an existing model inside Sesame using the fastest method

Individual user data and variances, step three, are not taken into account during the performance testing effort. Of course in the real production environment there is variation in model size. But for the testing effort it was decided to test using an average sized model. Examining the models produced at Astron we found that the average model consists of 1000 elements and that a change in the model leads to 20% of change inside the total model. The tests are implemented using this average model of 1000 elements and change is set to 20%.

Step four, determines the scenario distribution. There are six use cases identified, but these fulfill only two scenarios, committing and updating a new model. Updating is identified as the most used scenario; a model is only created once but updated more.

Finally there has to be an estimate of the load levels of the application, step six. The trigger is the commit into the repository. Every user commits their work at different points during the day, however, expected is that there will be peak levels around office breaks, most users commit their work before the break and the highest peak is expected at the end of the working day.

9.4 Configure test environment

To be able to run the tests correctly not only the test design needs to be good, also the environment must be configured correctly. All unnecessary services where shutdown for all tests; it was assured that for all tests the environment was the same. The JVM was set to use 500MB of memory instead of the default 300, higher would be better, but the test machine was not able to handle this so it was kept at 500. SVN needed the 500MB to prevent an "*Out of memory error*" and for the Sesame testing more available memory would be better for testing due to the memory targets and threshold identified in chapter 9.2 which affect performance. However allocating more memory to the JVM than 500MB produced undesired results, probably due to the use of swapping since windows did not have enough available memory.

9.5 Implement test design

The implementation of the test design is done in Java using eclipse. Java was the most obvious choice for Sesame, since it is written in Java, as was the existing Knowledge Hub.

For SVN the Java SVNKit was used. When implementing the model the following points need to be taken into account [11]:

1. Do not change the model without serious consideration simply because the model is difficult to implement.
2. Implement the model as designed, if there have to be changes make sure you record them in detail.
3. Think about the metrics that have to be recorded and how to implement the recording of those metrics.

Java classes will be created for each of the defined use cases, see 9.3. When looking at the detailed description of use cases in appendix I we see that the first two steps are similar for all use cases and therefore will not be implemented into the tests. The time for step one and two will always be the same for each model and use case, leaving them out will not affect the overall performance. Normally, SVN triggers the KEX instance. The start of the tests now simulates this behavior. The metrics that the tests will gather also have been defined in 9.3. A log class is made which the test notifies at certain points during the testing, collecting data needed to satisfy the pre defined metrics. At the end of the test, the performance class outputs a performance log generating data collected during the tests.

All of the classes are configurable so we have some control over the testing, we can adjust variables as the amount of data present in the repository, how much data needs to be inserted, what connection type is needed and some more. This gives some flexibility when testing and may help to specify some tests on crucial points.

Tests are always run more than once. A parameter can be set that defines how many times a test should run. This is done to get an average time to counter the problem of an erratic sample.

Sesame allows the use of three different connection types, default, augur, and read ahead, these connection types are optimized for storing and retrieving scenarios. The default connection type processes each request and does not cache anything; an augur connection type tracks the requests and anticipates related information. It is best used when the results are expected to fit into memory and not all properties will be read [7]. The read ahead connection type reduces the number of hits to the repository for the same subject. It is used when the complete results may not fit into memory and most of the concept properties will be retrieved [7]. The different connection types are used in the tests for the use cases of ELMO and SeRQL. When inserting files nothing is or can be cached so testing the different connection types is useless the default connection will be used for the file tests.

9.6 Execute Tests

Before executing the tests, checks were made to see if the test environment mirrored the execution environment. Concluded was that for testing purposes the environment was a good. Then all tests where run for the first time and each tests was monitored step by step for validation and was checked for unexpected behaviour. Not only where the tests checked, also random samples of data where checked to see if the tests did not transform them in any way. The tests and the environment where accepted as valid for the performance testing effort.

The SVN tests and the Sesame tests where all run and repeated 10 times. The average values of those 10 runs were taken. All test where done with an increment of 50 models inside the Knowledge Hub, starting with zero models and ending at 300 models. One model consists of

approximately 9500 triples or 1000 files. So after the final test there would be 2.850.000 triples inside the Sesame repository and 1000 files inside the SVN.

A special constructed log class, was used to log the time of all the different operations during the tests, and at the end of the testing plotted the results.

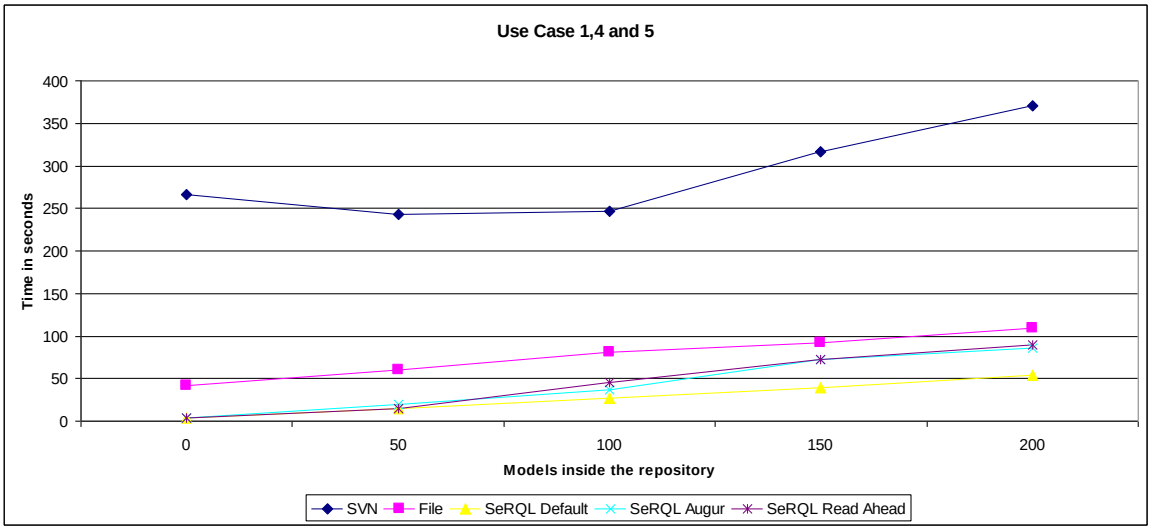
The other metrics that had to be recorded where done so using the data presented in the "Windows Task Manager". This could just be monitored while the test was running. System outs indicated what part of the testing process was running.

9.7 The Results

In appendix I, the complete results per use case can be found, as can the results of all the independent tests. Here only the combined results of the tests are presented. The results are split into the two different scenarios, the committing and updating use cases.

9.7.1 The committing use cases

The first graph below here shows the result of all the use cases that commit models into the repository. Use case three, ELMO, was left out of this graph for scalability reasons. The first performance test of ELMO with zero models inside the repository took almost two hours of time, see appendix I for further explanation and test results for ELMO. Testing was done from 0 – 300 models, the graph below shows only the result until 200 models for scalability and comparability reasons. After 200 models SVN became much slower. Instead of three lines for the two remaining use cases, five lines can be seen. A line is plotted for each Sesame connection type using the SeRQL use case.



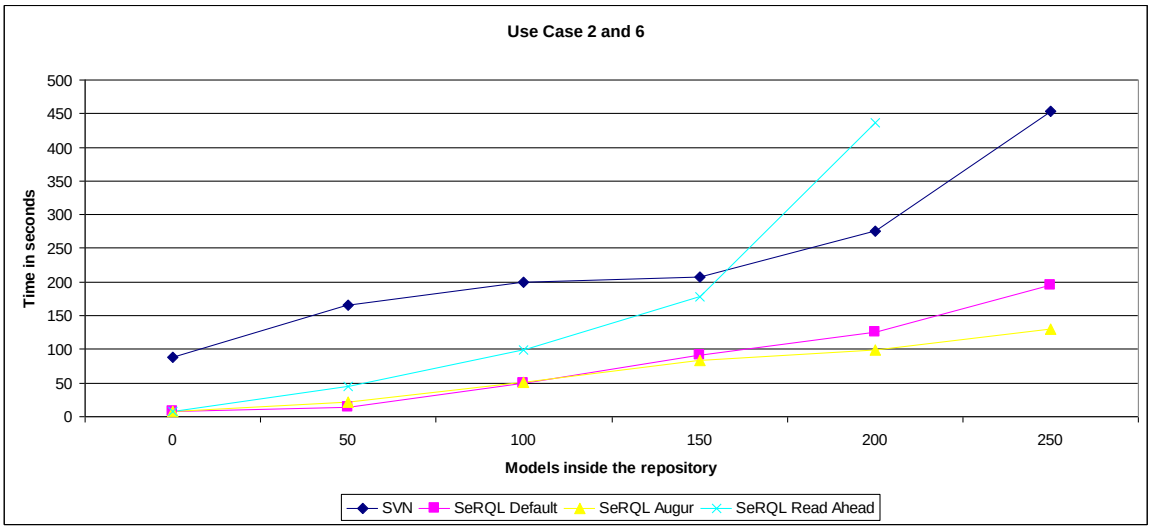
9.7.1.1 Analysis

As can be clearly seen, Sesame outperforms SVN by far. Looking at the Sesame results we see that File insertion is the slowest, which was expected since the file needs to be generated locally which takes extra time. The default repository connection type is the fastest, augur, and read ahead make no real difference.

9.7.2 The updating use cases

The second graph shows the test results of the updating use cases. Again SeRQL is split over the three different repository connection types. Here we scale until 250 models, with the

exception of the SeRQL Read Ahead connection type, which goes only to 200, for scalability reasons.



9.7.2.1 Analysis

As was the case with inserting, Sesame also outperforms SVN on updating, with the exception of the read ahead connection type. The augur connection type is faster here than the default which can be explained since the augur connection type is optimized for retrieving data which is part of the updating process.

9.7.3 Other metrics

The above sections focus on the performance with respect to the time it takes to complete a test. Next to the time metric other metrics where measured. Targets and thresholds were measured using the windows task manager. The memory usage of Sesame was measured using through the export of the data inside the repository.

The average CPU usage of SVN during the tests was between 60 and 70%, with a target of 70% this is good. The average CPU usage of Sesame was 100% this was also the expected result.

Sesame seemed to use all the available memory when around 200 models where inserted into the repository. An export revealed that at that moment only 300MB of actual data was present, the rest was used for inferencing purposes. After the 200 models we inserted and tested with 250 models, the available memory stayed the same, due to Java VM constraints (500MB) and the resulting performance drop was not really noticeable. When scaling up to 300 models the performance dropped fast, after an export it showed that there was almost 500MB of actual data inside the repository leaving almost no memory left for inferencing and other operations, which must lead to swapping.

9.8 Retesting and validating

After all test where run and the results plotted, all tests where run again to see if the results could be reproduced. The tests were already accepted as valid; reproducing the results cancels out test failure and is a form of result validation.

9.9 Conclusion

Looking at the combined results it can clearly be seen that Sesame has the upper hand when it comes to performance. SVN does outperform Sesame when Sesame runs low on available memory; however this can be countered by adding more memory to the system. The results came from the limitations of the testing system, and where expected by the defined targets and thresholds. So if a system can be outfitted with enough memory, which is easy to do with modern servers, Sesame is definitely the best solution.

The secondary concern was the scalability of the system; it is obvious that SVN can scale very high since disk space nowadays is cheap and very large. Scalability for Sesame on a desktop is not so good due to memory limitations; however modern servers can be equipped with large amounts of memory improving Sesame's performance. Sesame also has a built in functionality to create a federation; this combines multiple sesame stores through the SAIL layer acting as one large data store [51] boosting performance and creating huge scalability.

10 Qualitative Evaluation

The quantitative evaluation in chapter nine tested the performance of the different solutions. In this chapter, performance and the remaining requirements are graded aiding the selection between the proposed solutions. First an evaluation framework is presented; the framework provides a basis for the comparison of the solutions. The second part of this chapter focuses on the grading process, each requirement is graded and a rationale for the grade is given. The final part of this chapter inserts the grades into the framework and based on that the optimal solution is chosen.

10.1 Evaluation Framework

The evaluation of the proposed solutions was done using a grading framework. In this framework the solutions are graded on the requirements, indentified in chapter seven. Here an overview of those requirements:

1. Storing
2. Retrieving
3. Performance
4. Reliability
5. Implementation
6. Querying
7. Transparency
8. Maintainability

The framework will look as tTable 1 The Evaluation Framework, where Rqx equals requirement X and A, B are the proposed solutions. The first two requirements from the above list will not receive a grading, as they are only required to be present, an OK is assigned. The remaining requirements will be graded using the following table:

- Very Good = ++
- Good = +
- Average = +/-
- Bad = -
- Very Bad = --

	A	B
Rq1		
Rq…		
Rqn		

Table 1 The Evaluation Framework

10.2 Qualification of the requirements

In this section, the requirements for the SVN and the Sesame solution are graded. Each solution is grades the identified requirements one by one. Advantages and disadvantages of each requirement are identified, combined with a rationale this lead to the grade the requirement received.

10.2.1 Sesame with SVN

Storing & Retrieving: OK

Storing and retrieving was not a requirement that had to be graded, it was only required of the application. In respect to SVN we can say that is has storing and retrieving capabilities conform the requirements so this was checked as ok.

Performance: --

Looking at the previous chapter, the performance of SVN was pretty bad. A single commit with no files present in the repository took around five minutes; this does not fall within the acceptable performance range. Updating models using SVN was faster but still took 1.5 minutes with an empty repository; with 50 models inside the time also dropped to three minutes. The proposed performance upgrades, will not boost the performance enough to get SVN performing within acceptable speeds. Therefore performance gets graded very bad for the SVN solution.

Reliability: ++

Our definition of reliability, chapter 3.2, stated that a system must have correct handling of errors and that it must prevent data corruption.

Advantages

- SVN is a well developed solution that, in industry, has proven to be very reliable when it comes to data corruption.
- SVN has a rollback functionality that is triggered upon failed insert. This prevents the crashing on errors and ensures reliability.

SVN is a proven solution that is very reliable in both aspects. It is a mature and well tested system in business solutions. It has no disadvantages in respect to reliability. SVN as a solution received a grade of very good on reliability.

Implementation: +

The requirement of implementation, chapter seven, refers to the effort it will take the developer to implement the solution.

Advantages

- SVN is an existing and stable solution that handles versioning very good
- The versioning is taken care of by SVN the developer simply calls the SVN API to store the files there.

Disadvantages

- In order to implement SVN there has to be a Sesame repository inside the client programs.
- An SVN layer has to be developed that understands the versioning and needs to be implemented on the client side, communicating with SVN and the local Sesame repository.

- The clients still need SeRQL/ SpaRQL queries to access the remote Sesame repository that contains the current version.

SVN has some clear advantages and disadvantages in respect to implementation. It is an existing and stable solution that is proven in industry. The developer gets a clean interface and needs no programming effort to implement versioning, but has to build a custom backend for each client to be able to work with the system. The system also makes use of both SVN and Sesame and offers no integrated solution. Nevertheless, the effort to implement this is reasonably low and therefore it is graded good.

Querying: +
Querying was a requirement (see chapter seven) that checks if there is an available query language / mechanism for the solution or that the developer needs to implement his own querying mechanism.

Advantages
- Java provides an API for SVN that allows for easy retrieval of the versioned files, it is publically available and easy to use.

Disadvantages
- Next to SVN, there is also the need to query Sesame using SeRQL, thus making use of two querying mechanisms instead of one.
- The retrieval of the SVN files cannot be directly used within the complete system, the data still needs to be parsed from the files in some way. This must be done client side, adding to the querying mechanism.

Although SVN, through the use of the Java API allows for easy querying and retrieval of the versioned files, there still is the need for SeRQL to query the current data from Sesame. Also the retrieved files from the SVN need to be parsed to get the data out of them in a usable format. This can be implemented fairly easy by importing the data into a local Sesame repository. Therefore querying from SVN is graded good.

Transparency: +/-
Transparency was a twofold requirement (see chapter 3.2), looking from the perspective of the users as well as for the developers. For the users it meant that they should be able to work with the system without in-depth knowledge of the system. For the developers transparency meant that for them it should be clear which parts of the system provide which functionality, making it easier to understand the system as a whole.

Advantages
- SVN is clear and understandable for developers. It comes with a well defined API and is widely known in the industry.

Disadvantages
- The SVN is another component in the overall system architecture, adding to the complexity of the architecture.
- The Roles of Sesame and SVN are intertwined, Sesame hold the current version and SVN the older versions, which is confusing when trying to understand the system.

Although SVN is an off the shelve component. The integration of SVN with the Knowledge Hub adds an extra component to the architecture. Another consequence is that it distributes the data over two different data repositories. This is very confusing for developers wanting to understand the workings of the system. Taking this into account we grade SVN average on transparency.

Maintainability: +/-
Maintainability refers to the ease of maintaining the application, chapter 3.2. How easy is it to connect new clients that have to use the versioning system, to the Knowledge Hub? How difficult is it to implement new functionality without altering too much throughout the system. A maintainable system is needed to easy the effort of implementation of these new clients or functionality.

Advantages
- The Java SVN API is well documented and new releases can be easily integrated with the system.

Disadvantages
- Through the use of SVN there is more code needed in the clients for retrieving data from the SVN and parsing it into understandable data.
- Adaptation of the system must be done in all clients.

SVN is easily maintained, however SVN puts some effort into maintaining the rest of the system. When a developer wants to add new functionality to the Knowledge Hub or even when the data changes in the hub, all clients have to be adapted. Because SVN always functions in the same way, the parsing and handling of the data is programmed client side, this is bad for maintainability. Therefore SVN is graded average for maintainability.

10.2.2 Extending Sesame

Storing & Retrieving: ok
As was the case with SVN, Sesame also has storing and retrieving capabilities, conform the requirements. Therefore Sesame also gets an OK.

Performance: ++
Looking at the performance of Sesame in the previous chapter we can see that it outperforms SVN easily. Inserting and updating with 0 – 50 models inside the repository is very fast and stays under 30 seconds per model. With the proposed upgrades to the memory it can only be assumed that the Sesame solution will become even faster, therefore Sesame is graded very good on performance.

Reliability: +
Our definition of reliability, chapter 3.2, stated that a system must have correct handling of errors and that it must prevent data corruption.

Advantages
- The available rollback mechanism ensures a reliable insertion of data.
- The maturing in chapter four, made the current system much more reliable, the final system at the ends of that chapter is the basis of the proposed system.

Disadvantages
- The existing system also had some reliability problems, using sesame and developing a custom solution can make these problems resurface or introduce new problems.
- Sesame and RDF are still in early development and have no proven track record.

Sesame has some advantages, and the rollback ensures some reliability. But a lot depends on the implementation of the solution, and since Sesame is still not widely used in the industry problems can still emerge. However the testing and stabilizing done in chapter four made the overall system pretty stable thus we grade Sesame with a good on reliability.

Implementation: +
The requirement of implementation, chapter seven, refers to the effort it will take the developer to implement the solution.

Advantages
- The solution integrates with the existing system
- The Sesame core is very extendible
- The implementation is in the back end, the implementation client side is minimal.

Disadvantages
- It is a custom solution that needs to be developed.
- A good understanding of the complete Sesame architecture is needed in order to implement the solution correctly; this implies a lot of research.

Although the Sesame architecture allows for easy extension, the implementation is a difficult one. The developer needs in-depth knowledge of the architecture of Sesame and must implement one or two extra layers. The plus side is that the client side implementation is almost effortless, only some more advanced queries are needed to retrieve the data from the repository. So Sesame is graded good for implementation.

Querying: ++
Querying was a requirement (see chapter seven) that checks if there is an available query language / mechanism for the solution or that the developer needs to implement his own querying mechanism.

Advantages
- The querying is the same as in the current knowledge hub, only queries to retrieve the versioned data must be written.
- The querying mechanism in Sesame allows the retrieval of multiple versions in one query.

Disadvantages
- The queries can become very complex.

The querying mechanism of Sesame can be used without an adaptation. It even allows for querying over multiple versions, although these queries can become very complex. Sesame was graded very good for querying.

Transparency: +
Transparency was a twofold requirement (see chapter 3.2), looking from the perspective of the users as well as for the developers. For the users it meant that they should be able to work with the system without in-depth knowledge of the system. For the developers transparency meant that for them it should be clear which parts of the system provide which functionality, making it easier to understand the system as a whole.

Advantages
- The system architecture does not change.
- Adaptation is done in the server and not the clients, these just have to adapt the queries where needed.
- All data is stored in a single repository.

Disadvantages
- Very complex queries can be misunderstood by developers.

The main advantage of Sesame over SVN is that everything stays in one repository; this not only keeps the system architecture clean but also stores all of the data in a single Sesame repository instead of Sesame and SVN. Complex queries stay a problem and need to be documented good in order to be reused. Nevertheless the system is transparent and adaptable and therefore is graded good.

Maintainability: +
Maintainability refers to the ease of maintaining the application, chapter 3.2. How easy is it to connect new clients that have to use the versioning system, to the Knowledge Hub? How difficult is it to implement new functionality without altering to much throughout the system. A maintainable system is needed to easy the effort of implementation of these new clients or functionality.

Advantages
- Adapting the versioning system does not affect the clients.
- Connecting new clients is easy, since the current version and older versions are all accessible by the Sesame query language.
- Changing the data that clients use is done by formulation new queries.
- Adding more functionality to the system does not directly affect the clients

Disadvantages
- Complex queries are hard to change if a developer has no good understanding of them.
- Added functionality must be queried, to use it all queries client side have to be adapted.

Sesame has some advantages when it comes to maintainability, most of the adaptations are done in the server and do not affect the clients. With the exception of the complex queries this solution is very maintainable therefore Sesame is graded good on maintainability.

10.3 Conclusion

In the above section, the grading of the different requirement for the proposed solutions was done. The resulting grades were collected and inserted into
Table 2. The first two properties, storing and retrieving, are graded with present or not present. The other six have been graded from very good to very bad. The performance property, since it is so important, was counted double. The average score for each solution was calculated and the highest average was chosen as the optimal solution.

	SVN	**Sesame**
Storing	OK	OK
Retrieving	OK	OK
Performance	--	++
Reliability	++	+
Implementation	+	++
Querying	+	+
Transparency	-/+	+
Maintainability	-/+	+

Table 2 The Evaluation Framework

Average score for SVN:	Average	(--, --, ++, +, +, +/-, +/-)
Average score for Sesame:	Very good / Good	(++, ++, +, ++, +, +, +)

Sesame has the highest average score, scoring a Very good / good, SVN scores average. The score of Sesame is substantially higher than SVN; the main reason for this is the performance of SVN, which is far worse than Sesame. Since performance was the main requirement it counted double giving Sesame a good lead. SVN is more reliable than Sesame, a careful implementation and a thorough checking can boost the reliability of the Sesame solution. SVN is the winner for this requirement being proven in the industry for years now. Sesame on the other hand outperforms SVN on; implementation, querying, transparency and maintainability. Therefore Sesame was selected as the optimal solution to use as a versioning system inside the knowledge hub.

11 Validation / Re evaluation

The previous chapter concluded with the optimal solution for versioning inside the knowledge hub. In this chapter a prototype is described that was build using that solution. The prototype contains the basic functionality of the proposed versioning system. Artifacts can be inserted and will automatically be versioned and the versions can be retrieved from the repository through the standard sesame query language.

11.1 The prototype

When we look back at the inner workings of Sesame (chapter 8.2.2.1), the current repository uses a memory store, which is an extension of the sail repository. This class, where all the storage logic is located, is extended and equipped with methods and logic for versioning. An extra class is added which holds the logic for the versioning system. This class calculates the delta's and inserts them into the repository.

11.1.1 The inner workings

The Sesame architecture and the versioning system architecture, see chapter eight, presented the idea where the extension would be implemented. This architecture however told nothing about the workings of the versioning system only it's place inside the architecture.

Figure 10 (chapter eight) showed that the repository contents will be divided into sections using the context property. Sesame 2 introduced this fourth property, named context, making quads from the triples in Sesame 1. This context is used to divide the statements over the sections. All of the statements that are in the so called "*head*" version are placed into the current context. The versioning system calculates from the new inserts which statements are added and which are deleted. These statements are put into delta contexts. These contexts contain only the statements that where added or removed not the complete state of the object to save space and improve speed. The reference to these delta contexts is stored inside the system context. There is only one kind of statement inside this context and it looks like the following triple:

Subject / versionedAt / delta contex

But actually inside the repository is the following quad

Subject / versionedAt / delta contex / versioning

This allows for a very natural querying, the system context can be queried using the subject that is found in the current context. The result is a list of statements that contain the delta contexts of that particular subject. The delta context property is structured as followed:

- The versioning system URI.
- The subject
- A keyword ADD or REMOVE
- A date and time stamp

The URI is general over the entire versioning system and is of no concern. The subject is needed to assign the delta to the subject. The ADD or REMOVE tells the user if the statements where added or removed from the repository in that version and the date and time stamp tells when the version was created. Below we see an example of a statement from the versioning system.

http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000000/ADD/2010-01-28/09:40:46

The workings of the versioning system and the implemented prototype are explained using the sequence diagram from Figure 14. This diagram shows on the left a random client application, the application inserts a collection of statements into the Sesame repository using the *addGraph()* method. The workings of Sesame are left out, there is a lot going on there but it is of no interest to us. Only the relevant calls to the versioned memory store are shown. There are other calls to the memory store like a statement lock and more, but for a better uncluttered view they are left out. There are only two calls from the client application the first call that adds new statements and the second one that commits the inserted statements.

The first call that is of interest is *startTransaction()*, this reinitializes the statement cache to ensure that it is cleaned of all statements. Next, the *addStatement()* method is called for each statement that has to be added. The statements that are added will be put into a local cache instead of inserting them into the repository.

The second call, the *commit()* normally would add all the inserted statements into the repository. In the prototype, the versioning system is triggered from within the commit operation and runs before the actual commit. The first thing that is done is a calculation, on all the triples inside the cache using the logic defined in 11.1.2. This calculation results in two sets of triples: one set that are new and one set of triples that are removed, these triples represent the add and remove deltas. The triples are inserted into the versioning system class. The *buildVersionedStatements()* method operates on these two sets of triples and creates the add and remove deltas and puts those delta statements into a list. It also creates a list of statements that will be removed from the current context. The versioned memory store retrieves both lists, the delta statements are added to the repository and the other statements are removed from the current context. Finally, after clearing the caches and the versioning system the original commit is performed.

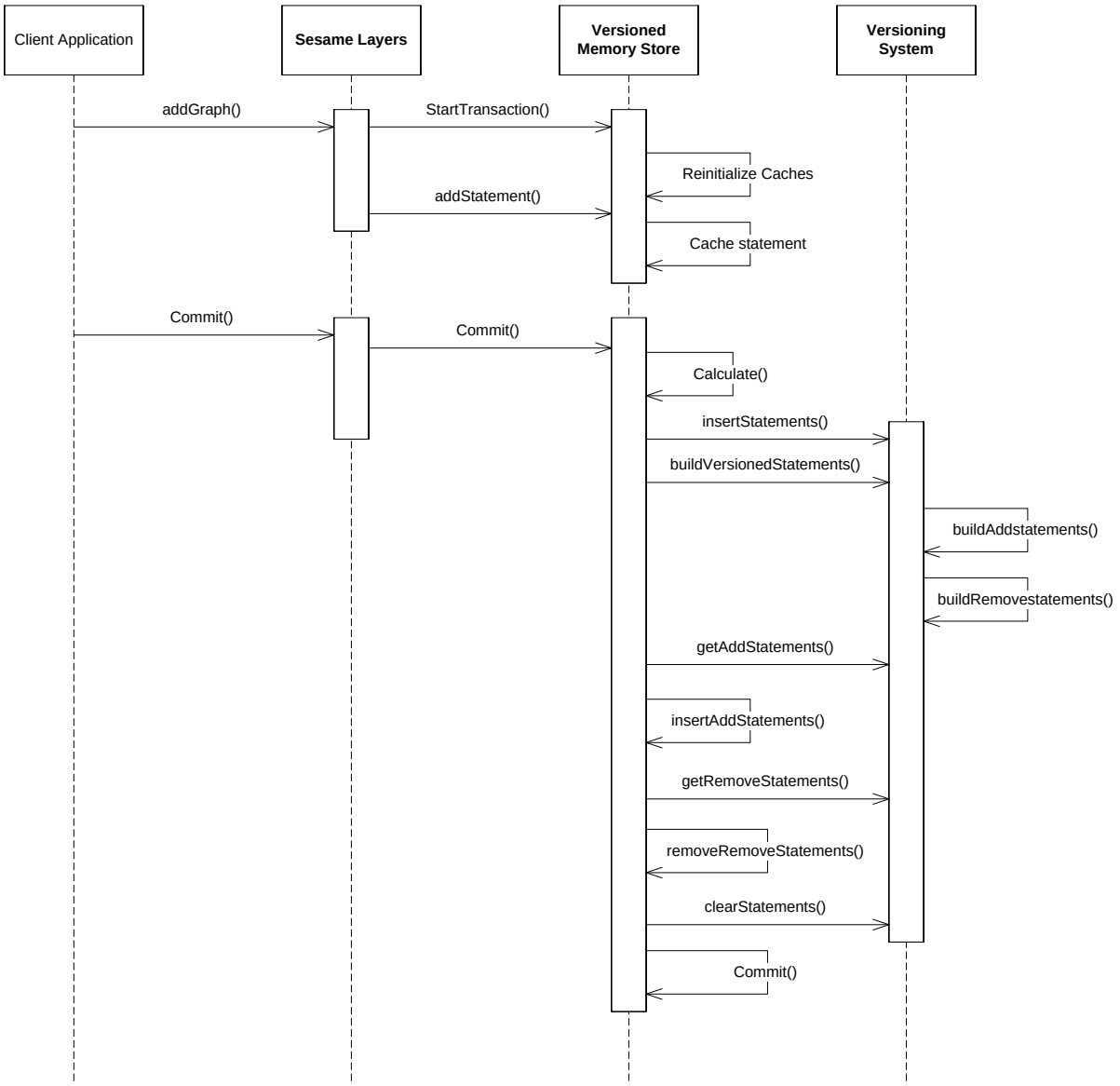


Figure 14 Versioning System Sequence Diagram

11.1.2 The Logic

The logic that will be implemented into the versioning system is a simple one. To be able to version the data we need to know the new and the old versions. So let's say the new version is a collection of statements called N and the old version is a collection of statements called O. Calculation then can show which statements are new and which are removed. The following two formulae are needed:

$$N - O = A$$
$$O - N = B$$

By subtracting O from N the newly added statements remain (A) and by subtracting N from O the removed statements are found (B). So by implementing the following two formulae the versioning system knows which statements to version and remove;

11.1.3 Rules for usage

The versioning system works from the principle that a unique subject is seen as an object in our repository. The repository versions on objects using this understanding of subjects and not on triples. Therefore four rules have to be taken into account when using this versioned memory store:

1. Only insert collections of statements, and not single statements.
2. Do not use the removal methods, removal is done through insertion.
3. When querying always query the *current* context, unless access to versioned data is required
4. The fourth property, *context*, may not be used, this is reserved for the versioning system.

The first rule is needed because of the workings of the versioning system. If only a subset of the statements of an object is inserted the repository thinks, through calculation, that the other statements have been removed from the object and versions them as deleted. This is also the reason for the second rule. Through insertion of the complete new status of an object we can calculate what has been deleted, thus a delete/remove function is not needed unless we want to fully delete an entire object/artifact from the repository. It is still possible to use the removal methods, but the removed data is not versioned and there is no way of retrieving the data. The removal methods can be used to remove complete artifacts from the repository and also removing all the versioned data.

The third and fourth rule are also intertwined, a normal repository operation operates on the *current* context instead of the sesame default. The prototype is built in such a way that all inserted statements automatically are put into the current context. Thus queries have to be formulated using only the current context. To retrieve versioned data, the queries should access to the versioning system context, from where they can retrieve the contexts that contain the deltas of the different versions of objects. This automatically leads to the fourth rule, since the versioning system uses the context property the user is not allowed to do so.

11.1.4 The test cases

To validate the prototype test cases were formulated and implemented. The test cases cover the different scenarios that are possible for the knowledge hub. The test cases are modeled using a sub model (see Figure 15) of the LOFAR domain model (See appendix II), and the higher meta-model (See appendix III). There are three test cases, simulating the normal evolution of a software architecture documentation. The first test case inserts the smallest model possible into the repository; the next two test cases make changes to that model and add or remove objects from it.

Test case one

The first test case inserts three Knowledge Entities (KE) into the repository, a concern, a decision topic, and a quick decision. This represents the smallest complete and correct model possible. According to the meta-model each KE has an Artifact Fragment (AF) and are connected to an Artifact. The test case inserts seven objects, three KE's, three AF's and one Artifact. The system should insert them normally without triggering the versioning system, because they are new objects and do not exist inside the repository.

Test case two

The second test case uses the same model as the first test case. Some random changes are made inside the KE's, AF's and Artifact and a new KE and AF are added. To ensure a correct

model the inserted KE will be of the type decision and the existing quick decision is changed to an alternative. To each KE also a second set of notes is added. The versioning system should recognize all these changes and build the correct deltas accordingly. Also the new KE and AF should be inserted without being versioned.

Test case three
The third and final test case tests the removing of a complete KE and some notes from existing KE's. To do this the original model is reinserted, making the versioning system believe that one of the KE's is removed and that the extra notes also are removed from the KE's. Also some random changes are made to the KE's, AF's and Artifact. The alternative is changed back to a quick decision and the decision is removed. The versioning system should build the new deltas and remove the obsolete statements from the repository.

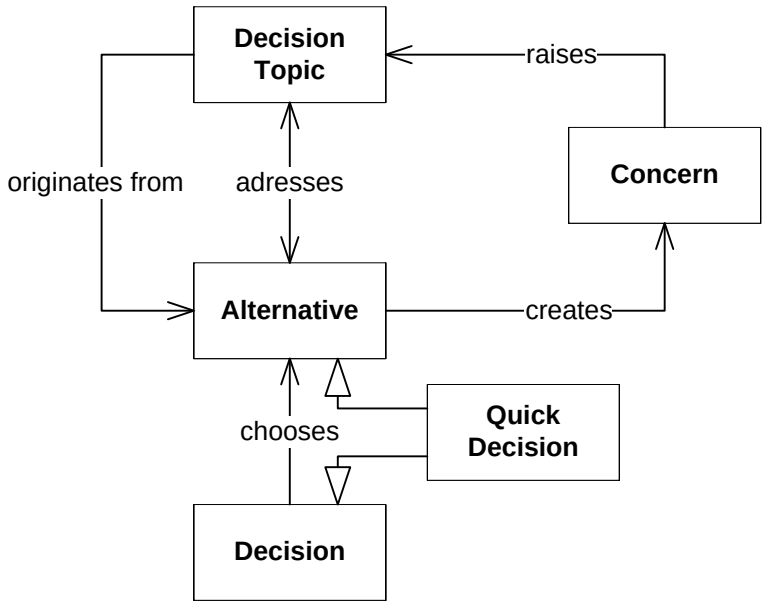


Figure 15 A Sub Model of the LOFAR Domain Model

11.1.5 Queries

The test cases described in the section above also have to be tested. For that purpose some queries where designed. These queries validate that we can access the current data and retrieve versions from the repository using Sesame's querying language SeRQL. Each query is presented here, to get a better understanding of the SeRQL query language see the Sesame documentation [7]. To test the working of the system the following four queries where formulated:

1. Retrieve a KE from the current repository
2. Retrieve ALL KE's that are connected to the Artifact
3. Retrieve all delta's from the versioning system
4. Retrieve all changes that are stored in a certain the delta.

Query one
The query retrieves a KE from the repository and queries only the current context as can be seen on line one of the query. This should return the KE and all its connected properties from the repository. This query answers three questions:

1. Is the inserted data put into the current context?
2. Can the data be retrieved from the current context?

3. Is the inferencer working?

Querying the current context answers the first two questions. However by querying for a KE from the repository the working of the inferencer is also checked. The inserted type was not a KE but a subtype of KE from the Astron model, see Figure 15, through that model we know that a concern is a KE. So by querying for a KE instead of a concern we also check that the inferencer is working properly.

```
SELECT * FROM CONTEXT <http://www.griffin.nl/versioning#current>
        {ke} rdf:type {ma:Knowledge_Entity};  y {z}
         WHERE ke LIKE  *KE1000000
         USING NAMESPACE ma = <http://www.archium.net/AstronGriffin#>;
```

The expected result of the query is the correct retrieval of the KE after each test case.

Query two
The second query check if we can retrieve all the KE's that are connected to an artifact. The test cases remove and add KE's to the artifact this query tests after each test case how many KE's there are connected. As can be seen it also queries the current context.

```
SELECT ke FROM CONTEXT <http://www.griffin.nl/versioning#current>
        {A} rdf:type {ma:Artifact};
        mv:describes_knowledge_entity {ke}
        USING NAMESPACE ma = <http://www.archium.net/AstronGriffin#>,
                             mv = <http://www.archium.net/AstronGriffin/versioning#>"
```

The expected result of this query after test one and three is to retrieve three KE´s after the second test it should receive four KE´s.

Query three
The third query is executed only after the second and third test case. It queries the versioning system and retrieves all contexts that contain deltas. Running it after the first test would return nothing since no data should be present in the versioning system at that moment. Looking at the first line of the query, it can be seen that it now connects to the system context. All deltas can be found in this context.

```
SELECT * FROM CONTEXT <http://www.griffin.nl/versioning#system>
        {sub} pred {obj}
        USING NAMESPACE mv = <http://www.archium.net/AstronGriffin/versioning#>";
```

The expected result is to retrieve a different add and remove delta´s from the versioning system for each of the KE´s and AF´s and Artifact inserted. Unless there have been no changes to that object.

Query Four
The fourth and final query retrieves a delta from the repository. One of the retrieved delta contexts of the third query is used and inserted into the query on line one. Then all data from that context is retrieved.

```
SELECT * FROM CONTEXT < --- insert context here --- >
        {sub} pred {obj}
        USING NAMESPACE ma = <http://www.archium.net/AstronGriffin#>
```

The result of this query should be the contents of a delta; it depends on the delta inserted. Expected is to see some statements that where added or removed from the repository.

The queries are run after each test case. Obviously queries one and two have no result after test case one since the versioning system is empty at that moment. The third query is run once to check if the versioning system is empty as is expected query for cannot run since there should be no context available to query.

11.2 Validation

For the validation the test cases, from the previous section, are implemented and executed. The validation of those test cases is done in two parts; first we will inspect the data inside the repository manually. To confirm that the versioning system is working as expected. Secondly the queries are used to check if the data inside the repository can be retrieved and that the versioning system is working.

The manual checks of the repository showed that the versioning system was working correctly. The expected KE's and AF's where present in the current context and the delta contexts where created. Also the inferencer was working, since lots of generated statements were also found in the repository. Therefore the versioning system is accepted as working as far as manual testing goes.

The second part of the validation discusses the queries that are run on the repository. The previous section described four queries and the order in which they were executed after the test cases. The results will be presented for each test case and checked if they match the expected results. A short overview of the results is given here, the full results of the queries can be found in appendix IV.

Test case one
The first test case inserted the smallest possible complete model. After the test, the first three queries where run on the repository. The first query returned a KE as was expected. This answered the first three questions and confirms that the prototype is inserting all statements into the current context, that the querying of that context is possible and that the inferencer is working correctly. Not only could the inserted statements of the KE be retrieved but there were also a few inferenced statements returned with the query.

The second query consisted of an initial check to see if the inserted artifact and KE's where present. The actual test was to see if we could query for the KE's connected to the artifact and if the inferencer was working. The test returned the three inserted knowledge entities as expected.

The third test was done as a formality, the versioning context should contain nothing, the query should return nothing. This was the result when it was run, so the versioning system is behaving properly when new data is added.

Test case two
The second test case inserted some changes and a new KE. This was the first time the versioning system was triggered. After which the four test queries where executed. The first query was run again to see if the current context only contained the new changed statement and not the old. The test showed that the notes attached to the KE where changed and that a

new note also was added to the KE. The second query's result showed that now there were four KE's connected to the Artifact, as was expected.

The third query, checked to see if the versioning system context contained the newly created statements that provide the links to the delta contexts. The query result showed that each changed object was present in the system.

The fourth query took one of the returned delta's from the third query and retrieved the statements inside that context. This context should contain the added or removed statements. The selected context was the add context of the KE from test case two. It showed that two sets of notes were added in this version. The changed notes and the new notes, a quick manual check shows also a remove delta that contains the old notes that where connected to the KE.

Test case three
The third and final test case removes a KE from the repository and makes some changes to the existing KE's and AF's. The first query checked the KE to see if the changes are inserted; this was the case. The second test, queried for the KE's that are connected to the Artifact, three KE's where returned. This is the expected result since one was removed. The third query checks if the versioning system contains the new deltas. The data showed that for each changed KE a new delta was created. The fourth and final query also succeeded and retrieved the data from one of the delta's

The above test cases and queries show that the versioning system is working correctly. As long as the rules for usage are followed, there should be no problem. Models can be inserted into the system and through the use of standard Sesame queries the versioned data can be retrieved.

11.2.1 Storing Architectural Evolution

In chapter two the second problem statement and the derived research questions can be found. The problem statement was formulated as following:

The repository is not able to store the evolution of a software Architecture

The research questions were answered in the previous chapters and from the potential solutions the optimal solution was selected, which lead to the building of the prototype. The test cases simulated the behavior of the production system and the manual inspection and the queries checked if the test cases produced the expected results. All of the tests and the inspection and queries succeeded validating the prototype providing an answer to the second problem statement.

11.3 Re-evaluation

The prototype validation was done in the previous section; this section re-evaluates the prototype using the requirements formulated in chapter seven and the grading in chapter ten. This section re-evaluates the requirements using the prototype. One of the performance tests from chapter nine was redone to check the actual performance. The table below shows the requirements as they were graded in chapter ten.

	Sesame
Storing	OK
Retrieving	OK
Performance	++
Reliability	+
Implementation	++
Querying	+
Transparency	+
Maintainability	+

11.3.1 Storing and retrieving

The first two requirements where a must for the system, looking at the test cases and queries these requirements are met. The repository is able to store and retrieve not only the current data but also the versioned data.

11.3.2 Performance

The same test case that was used in the original performance test was rerun on the prototype. The test case was configured so that the repository would contain 10 models and then 10 random models would be versioned. The average time of inserting a versioned model was 32 seconds. The original test case was two times faster and averaged on 15 seconds. Since it is a prototype optimizations and tweaks still need to be done. On the other hand the speed of the prototype is already faster than the SVN solution.

11.3.3 Reliability

The rollback features in Sesame provide reliability, the prototype left this rollback functionality intact. The implementation was the main concern for the reliability. The ease of implementation and the small impact on the architecture left the original sesame implementation fully intact. At the moment not much can be said about the reliability, field testing it must provide more results. Since it is still a prototype probably some issues will emerge.

11.3.4 Implementation

As was expected the ease of implementation was good. The MemoryStore and MemoryStoreConnection classes of the Sesame implementation had to be extended and one other classes that contains the versioning system had to be made. A reasonable understanding of the workings of Sesame and the Inferencer was enough to do the implementation. The implementation only affected one location in the Sesame architecture making the ease of implementation very good.

11.3.5 Querying

Querying was available in Sesame and the design of the versioning system did not damage its functionality allowing full querying capabilities. The queries from the previous section showed that the queries can easily be used to retrieve information from the system. However the original concern that graded the querying with a + and not a ++ is still in place. Some queries can become very complex, especially when we want to retrieve multiple objects and their version information in one query.

11.3.6 Transparency

Transparency remains good, the sesame architecture is not affected by the prototype and the only concern was that more complex queries would be needed. The test cases and queries show that normal queries are not overly complex. But the need for complex queries is still there.

11.3.7 Maintainability

The systems maintainability is good; no structural changes to the architecture of Sesame had to be made. The system logic is contained in one simple class that is called from only one point in the Sesame code. Thus changes that have to be made are local and not system wide. This makes maintainability good. Javadoc is added to the prototype to support the maintainability.

12 Related work

This section explores the related work for this thesis; however, since most work has been done in a very specialized niche field, no specific related work is available. The work has been done inside the context of the Semantic web, section one, a lot of the used technologies can be derived from the semantic web and its vision lies in line with the vision of the GRIFFIN project which sees the repository as an openly available intelligent collection of architectural knowledge [30] present on the (Semantic) web for everyone. The second section discusses RDF Repositories, which we used to store our data in. The third section gives an overview of the inferencers that are available nowadays. The thesis and technologies all operate in the field of architectural knowledge which is addressed in section four. Section five focuses on versioning systems, versioning systems where proposed as the solution for our second problem statement. Finally, an overview is presented of existing versioning systems and the developed prototype.

12.1 The Semantic web

The Semantic Web is an evolving development of the World Wide Web in which the semantics of information and services on the web is defined, making it possible for the web to understand and satisfy the requests of people and machines to use the web content [25]. It derives from World Wide Web Consortium director Sir Tim Berners-Lee's vision of the Web as a universal medium for data, information, and knowledge exchange [26.]

"I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A 'Semantic Web', which should make this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The 'intelligent agents' people have touted for ages will finally materialize."

Tim Berners-Lee, 1999

The semantic web includes at its core a set of design principles, collaborative working groups and a variety of enabling technologies. Some of these technologies include the Resource Description Framework (RDF), several data interchange formats (e.g. RDF/XML, N3, Turtle, NTriples), and notations such as RDF Schema (RDFS) and the Web Ontology Language (OWL), all of these provide a formal description of concepts, terms, and relationships within a given knowledge domain [28].

12.2 RDF Repositories

RDF repositories are data sources capable of storing RDF data. Most of these repositories come with their own framework that lets the user connect different data stores and sometimes also reasoners that are capable of handling RDF(S) or OWL. The repository that the GRIFFIN project uses is Sesame. Other repositories with similar functionality are Jenna [48], and BOCA [49]. Jenna and Sesame look very similar; both come equipped with different data stores, and provide an interface to connect different reasoners. BOCA is limited in that it only has a DB2 data store and no inferencing capabilities. Emerging in the field of RDF repositories are the distributed repositories, they provide higher performance and fault tolerance. RDFPeers is a P2P based repository [50], Sesame 3 will come with a concept called federation [51] it allows, through a specialized SAIL layer, the connection of multiple Sesame stores over HTTP. The prototype build for this thesis extends Sesame with versioning

capabilities adding to its feature set. It was build in such a generic way that it is not specific for the GRIFFIN project but it could be implemented everywhere.

12.3 Inferencers

Inferencers are also called truth maintenance systems; in short this exactly describes what they do. They maintain the "truth" in a system of statements based on the model inserted in the system. At the moment two systems or repositories can be identified, RDF (S) and OWL systems. OWL is an extension for RDF and has more descriptive power to model a specific domain. Through the logic defined in the RDF or OWL model deductions can be made to inference data. This is in basic how an inferencer works. There are different algorithms and implementations for inferencing. Two classes of inferencers can be distinguished: Forward chain and backward chain inferencing. Forward chain inferencers, inference data upon insertion, and backward chain inferencers when querying for data. Our focus lies with forward chain OWL inferencers, Sesame, standard, comes equipped with a forward chaining RDF inferencer [**7**]. This can be used but will not fully make use of our model. Sesame offers a clean interface to connect RDF(S) / OWL inferencers. There are multiple OWL inferencers available; F-OWL is a forward chain inferencer that is based on F-Logic [**60**]. PELLET is forward chain inferencer that works on descriptive logic [52]. OWLIM also is based on descriptive logic and forward chaining and comes in two flavors swift OWLIM and big OWLIM [**8**]. Swift is the fastest inferencer available, to date. OWLIM is the best choice to combine with the versioning sytem, according to the predicted size of the repository a choice for swift or big can be made.

12.4 Architectural Knowledge

The main driver behind this thesis was the facilitation of further research into architectural knowledge (AK) through stabilizing the current environment and developing new functionality. The research done by the GRIFFIN project was driven through different architectural knowledge tools. These AK tools support the architect in the architecting process. Different tools and tool suites have been developed, supporting different architecting processes [54].

ADkwik is a collaborative platform based on a wiki, although it is not a standard wiki but an application wiki [55]. At the Carnegie Mellon University they also use a wiki-based collaborative environment in the master software engineering program [56]. ADDSS is a webbased tool that stores manages and documents architectural design decisions [57]. Archium, is a tool that provides tracability among different AK concepts such as requirements, decisions, architecture and implementation. It can resolve conflicts and synchronize various parts during the life cycle of the system. The concepts can be expressed using a specialized archium language [58]. AREL is a UML-based tool it creates and documents architectural designs and focuses on decisions and rationale [59].

The Knowledge Architect (GRIFFIN) is a tool suite that is designed to capture, translate, share, use and manage AK. It is based around a central repository that is accessed by different clients. The prototype developed in this thesis enables the central repository to manage versioning. It allows the retrieval of versioned information about the stored architectural knowledge.

12.5 Versioning systems

Versioning systems where proposed to solve our second problem statement. Different types of versioning systems exist today; three are viewed as a possibility for the problem statement in this thesis. The first is the Ontology Middleware Module (OMM) [27]. OMM extends the Sesame RDF(S) repository [7]; it adds functionality like tracking changes, adding meta-information and improved access functionality. It provides versioning functionality, that is, it stores versioning information over the entire repository. The second are file based versioning systems like SVN and related versioning systems like CVS, GIT. SVN versions files by calculating deltas, deltas contain information about the changes made to a file. Through those deltas we can calculate older versions of the files inside the SVN. File versioning systems are not specifically an extension of the GRIFFIN repository but may be adapted to fit. Another possibility are RDBMS's, Sesame can use MySQL and postgress. MySQL does not have version control but postgress has a version control system called Post Facto [43] however it tracks data schema changes and not data. The prototype developed in this thesis is based on SVN versioning systems, it calculates and stores deltas and integrates into the existing knowledge hub. It was written as an extension of the Sesame Memory Store implementation. The developed prototype is not specific for the griffin project but can be used by everyone, as long as they follow the rules for usage.

13 Conclusion

The focus of this thesis revolved around the two problem statements defined in chapter two. Chapter three and four described and addressed the first problem statement and the results were presented in chapter five. Here it was concluded was that the first problem was solved through the upgrades made to the system.

Chapter six described the second problem statement followed by chapter's seven to nine, which addressed the problem and looked into potential solutions. Those solutions were compared using a framework in chapter ten and the optimal solution was selected. From this solution a prototype was build which was validated and reevaluated in chapter eleven. The prototype was graded using the framework from chapter eleven and the performance was tested. As was to be expected, the results of the performance test could be improved. But keeping in mind that this was just the prototype and some of the code needs to be optimized, the results looked good. Both problem statements have been addressed and solved. The first problem statements solution is working for a while now, the prototype, which is the solution to the second problems statement, needs some more work before it is ready to take into production.

13.1 Future work

At this moment the repository is running in a mature environment (PS1) and the prototype is working (PS2). However before the prototype can be taken into production some more work has to be or can be done, namely the following:

- Maturing the prototype;
- Implementing the prototype into the current repository;
- Adding extra functionality to the prototype;
- Inferencing on current context;
- Publish the new versioning repository and get it accepted into Sesame.

The current prototype works and performs reasonably well, however it still is two time as slow as predicted. There are two main concerns that have to be addressed, at the moment OWLIM is still in beta release and for that reason it was not tested. The final release of OWLIM should be connected with the prototype; this can be done without programming it is just a configuration of Sesame.

Secondly for the logic of the versioning system, the current version inside the repository must be compared with the new data. At the moment the implementation of the retrieval of the old statements is the most costly method inside the versioning system. Improving this method will improve the overall speed of the system greatly.

The prototype is an extension of the already used Sesame repository, implementing it into the current server should be no problem. However some of the clients may have to be adapted slightly to be able to work with the server. The main thing is connecting them to the current context so they query for the correct data.

Functionality that could be added is branching and merging and locking of data. However the adding of extra functionality lies out of scope for the prototype and therefore is future work.

There are some rumors about Sesame three that it will provide some functionality to inference only certain contexts. This could speed up the repository greatly, it works by adding the OWL model to a certain context and the inferences should only work on that context. At the moment the OWL model is added to the current context however the inferencer still works on the entire repository. Keeping an eye out for this could improve the speed of the application greatly.

Literature

[1] *The SVN website*, http://subversion.tigris.org/

[2] D. Marjanovic, *Developing a Meta Model for Release History Systems*, H. Gall, M. Pinzger, January 2006

[3] *The CVS website*, http://www.nongnu.org/cvs/

[4] P. Liang, A. Jansen, P. Avgeriou, Knowledge Architect: A Tool Suite for Capturing and Managing Software Architecture Knowledge, 2009

[5] P. Liang, P. Avgeriou, *Tools and Technologies for Architecture Knowledge Management,* In Software Architecture Knowledge Management: Theory and Practice, pages 91–111. Springer, 2009.

[6] The SVNKIT website, http://www.svnkit.com

[7] The Sesame website, http://www.openrdf.org/doc/sesame/users/ch01.html

[8] The OWLIM website, http://www.ontotext.com/owlim/index.html

[9] The ELMO website, http://www.openrdf.org

[10] The OMM website, http://www.ontotext.com/omm/

[11] Meier, Farre, Bansode, Barber, Rea, *Performance Testing Guidance for Web Applications*, August 2007

[12] P. Avgeriou, P. Kruchten, P. Lago, P. Grischam and D. Perry, *Sharing and Reusing Architectural Knowledge – Architecture, Rationale, and Design Inten,*. Shark 2007 Report

[13] J. Bosch, *Software Architecture: The Next Step in Software Architecture, First European Workshop (EWSA),* vol. 3047 of LNCS, pp. 194 – 199, Springer, May, 2004

[14] D. Falessi, G. Cantone, and M. Becker. *Documenting design decision rationale to improve individual and team design decision making: an experimental evaluation. In proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering (ISESE '06),* pages 134-143, New York, NY, USA, 2006, ACM Press.

[15] R.C. de Boer, R. Farenhorst. In Search of 'architectural knowledge'. In *SHARK '08: Proceedings of the 3rd international workshop on Sharing and reusing architectural knowledge,* pages 71-78, New York, NY, USA, 2008. ACM

[16] P. Kruchten, P. Lago, and H. van Vliet. Building up reasoning about architectural knowledge. In *Procedings of the Second Internation Conference on the Quality of Software Architectures (QoSA 2006),* June 2006.

[17] I. Habli and T. Kelly. Capturing and replaying architectural knowledge through derivational analogy. In *SHARK-ADI '07: Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, page 4, Washington, DC, USA, 2007. IEEE Computer Society.

[18] R.C. de Boer, R. Farenhorst, P. Lago, H. van Vliet, and A.G.J. Jansen. Architectural knowledge: Getting into the core. In *Proceedings of the Third International Conference on the Quality of Software Architectures (QoSA 2007),* volume 4880 of LNCS, pages 197-214, july 2007.

[19] I. Nonaka and H. Takeuchi. *The Knowledge-creating Company: How Japanese Companies Create the Dynamics of Innovation.* Oxford University Press Inc, USA, 1995.

[20] J. Tyree and A. Akerman, Architecture Decisions: Demystifying Architecure, IEEE Software 22 (2005), no. 2, 19-27.

[21] J. S. van der Ven, A. G. J. Jansen, J. A. G. Nijhuis and J. Bosch, Design Decisions: The Bridge between Rationale and Architecture, in Rationale Management in

Software Engineering, A. H. Dutoit, R. McCall, I. Mistrik and B. Paech, eds., ch. 16, pp. 329{348. Springer-Verlag, March, 2006.
[22] A. Tang, M. A. Babar, I. Gorton and J. Han, A survey of architecture design rationale, Journal of Systems & Software 79 (2006), no. 12, 1792-1804.
[23] A. G. J. Jansen and J. Bosch, Software Architecture as a Set of Architectural Design Decisions, in Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA 2005), pp. 109-119. November, 2005.
[24] H. Zhuge, The Knowledge Grid. 2004. World Scientific Publishing Co.
[25] T. Berners-Lee, J. Hendler, O. Lassila, *The Semantic Web*, May 2001
[26] Herman, Ivan (2008-03-07). "Semantic Web Activity Statement". W3C. http://www.w3.org/2001/sw/Activity.html. March 2008.
[27] A. Kiryakov, D. Ognyanov, B. Popov,*Ontology Middleware System Documentation,* September 2009
[28] The Semantic Web Wikipedia, http://en.wikipedia.org/wiki/Semantic_Web
[29] P. Liang, A.G.J. Jansen, P. Avgeriou, *Knowledge Architect: A Tool Suite for Managing Software Architecture Knowledge*, RUG-SEARCH-09-L01, February 2009
[30] P. Lago, R. Fahrenhorst, P. Avgeriou, R.C. de Boer, V. Clerc, A. Jansen, and H. van Vliet, *The GRIFFIN Collaborative Virtual Community for Architectural Knowledge Management*,
[31] The Astron website, http://www.astron.nl/, the Dutch institute for radio astronomy.
[32] The GRIFFIN project website, http://www.rug.nl/informatica/onderzoek/programmas/softwareengineering/griffin/ProjectDescription.
[33] The OWL website, http://www.w3.org/2004/OWL/
[34] The OWL Features website, http://www.w3.org/TR/2004/REC-owl-features-20040210/#s1
[35] The Wikipedia Knowledge Management Entry, http://en.wikipedia.org/wiki/Knowledge_management
[36] P. Liang and P. Avgeriou, *Tools and Technologies for Architecture Knowledge Management*, 2009
[37] The Protege website, http://protege.stanford.edu/overview/index.html.
[38] The jax ws website, http://java.sun.com/developer/technicalArticles/J2SE/jax_ws_2/
[39] J.broekstra and A. Kampman, *Inferencing and Truth Maintenance in RDF Schema*.
[40] List of revision control systems, http://en.wikipedia.org/wiki/List_of_revision_control_software
[41] The Sesame 2 website, http://www.openrdf.org/doc/sesame2/2.3-pr1/users/userguide.html
[42] The PBXT website, http://www.primebase.org/, PBXT, MySQL version control
[43] The post-facto website, http://www.post-facto.org/, Postgres version control
[44] The Oracle version control systems website, http://www.dba-oracle.com/t_version_control_change_control.htm
[45] MySQL, Postgres and Oracle Benchmarks, http://phplens.com/phpeverywhere/node/view/23
[46] Postgres and Oracle benchmarks, http://www.informationweek.com/news/software/linux/showArticle.jhtml?articleID=201001901
[47] MySQL and Postgres benchmark, http://www.randombugs.com/linux/mysql-postgresql-benchmarks.html
[48] The Jena website, http://jena.sourceforge.net/tutorial/RDF_API/index.html
[49] The Boca website, http://ibm-slrp.sourceforge.net/wiki/index.php/BocaUsersGuide-2.x

[50] M. Cai, M. Frank, *RDFPeers: A Scalable Distributed RDF Repository Based on a Structured Peer-to-Peer Network*, 2004
[51] The Sesame Federation website, http://wiki.aduna-software.org/confluence/display/SESDOC/Federation
[52] The pellet website, http://www.mindswap.org/2003/pellet/
[53] J. Rasmussen, *Understanding Software Architectures: Tracing Architectural Knowledge In Software Architecture Documentation*, 2009
[54] P. Liang, A. Jansen, P. Avgeriou, *Knowledge Architect: A Tool Suite for Capturing and Managing Software Architecture Knowledge*, 2009
[55] N. Schuster, O. Zimmermann, C. Pautasso, *ADkwik Web2.0 collaboration system for architectural decision engineering*, Proceedings of the19th International Conference on Software Engineering & Knowledge Engineering (SEKE), pp.255–260 (2007)
[56] F. Bachmann, P. Merson, *Experience using the web-based tool wiki for architecture documentation*. Tech.Rep.SEI-2005-TN-041, Carnegie Mellon University (2005)
[57] R. Capilla, F. Nava, S. P´erez, J.C. Due˜nas, *AWeb-based Tool for Managing Architectural Design Decisions.* 1st ACM Workshop on Sharing Architectural Knowledge (SHARK). Torino,Italy(2006)
[58] A. Jansen, J. Bosch, *Software Architecture as a Set of Architectural Design Decisions*, Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA), pp.109–119.IEEE Computer Society(2005)
[59] A. Tang, Y. Jin, J. Han, *A Rationale-Based Architecture Model for Design Traceability and Reasoning.* The Journal of Systems and Software (2007)
[60] The F-OWL website, http://fowl.sourceforge.net/

APPENDIX I The Performance Testing Report

A Quantitative Analysis report

Performance Testing SVN and Sesame

Hubert ten Hove
University of Groningen

14 January 2010

1 Introduction

This report was written to support the decision making process between different proposed solutions for a versioning system. For that system several options were presented and for those options properties were selected. One of those properties was the performance of those solutions. A performance test was done on the proposed solutions. This report shows the results and the process of the testing effort. At the end of the report we make some conclusions about the best model to use in respect to the performance.

In the paper Performance Testing Guidance for Web Applications [1], they provide a thorough method for performance testing. This method is followed in this report; however some parts are left out or slightly adapted to suit the needs for our specific performance tests. Also some parts are added to provide the possibility to make some predictions. It provides a structured test plan, for each step there is a chapter in this report, the steps are:

- Identify Test Environment
- Identify Performance Acceptance Criteria
- Plan and Design Tests
- Configure Test Environment
- Implement Test Design
- Execute Tests
- Analyze, Report and Retest

2 Identify the test environment

Identifying the test environment can be seen as evaluating the system to increase performance testing. Why is there the need to identify the test environment, according to [1]; "The intent of system evaluation is to collect information about the project as a whole, the functions of the system, the expected user activities, the system architecture, and any other details that are helpful in guiding performance testing to achieve the specific needs of the project". So evaluating our system on forehand gives a better understanding of the system as a whole and results in a better test design. This is essential for good performance testing. Evaluating the system is composed of the following activities:

- Identify the user-facing functionality of the system.
- Identify non–user-initiated (batch) processes and functions.
- Determine expected user activity.
- Develop an exact model of both the test and production architecture.
- The test System

2.1 Identify the user-facing functionality of the system.

This means what does a user need to know of the functionality of the system and how does he/she interact with the system and what actions can be performed. This can be divided into two parts since the system will be used by two different applications. The Knowledge EXtractor (KEX) and the Matrix EXplorer (MEX).

When looking at the KEX the user does not have much functionality, it is expected that the user has some sort of SVN client, like tortoise [7], which is used to commit all of the users files. Expected is that users commit and update their files several times a day. After a commit the SVN triggers the KEX, the user has no control over this process.

A Second system is het MEX, the MEX is used to explore all of the architectural knowledge inside the repository, and therefore will also be used to check the versioned data. Here the user can checkout different artifacts or knowledge entities and check their version history. A checkout of an artifact is important for the performance, but working with knowledge entities only is a small amount of data. This is then also not tested in the performance tests.

2.2 Identify non-user-initiated (batch) processes and functions

After the user performs a commit to the SVN a trigger inside the SVN launches the KEX (**Error! Reference source not found.**). The KEX intercepts the files that contain the models, and parses the required data from those files. This data is then stored in the Repository of the versioning system. This can be seen as both a user functionality or a batch process. The user has nothing to do with it and also has no knowledge of how it is done. Thus we speak of a non-user-initiated process, keeping in mind that actually the commit to the SVN is the user trigger that activates this process on the side.

2.3 Determine expected user activity

It is expected that most of the systems load comes from the users committing new and changed model into the repository, which is done through the trigger in the SVN. Also it is expected that load will come from users navigating through the data using the MEX. At the moment it is unknown how many users will actually use this functionality. This may be reevaluated after the system is in place. However the MEX mostly is just checking out data from the repository, which is also part of the updating process. Thus the performance of checkouts is measured.

2.4 Develop an model of both the test and production architecture

To get a better understanding of the system and the test system an exact model of the logical architecture is made and is shown in Figure 1. The user commits his files to the SVN, while SVN inserts the files into the SVN database a trigger launches the KEX. The KEX then parses the architectural knowledge from the files and commits or updates those files inside the repository. In case of the MEX , Figure 2, the user asks for a view of a certain object inside the repository, the MEX retrieves this data, and commits if the user makes changes to the data. The dotted area is the testing architecture. The SVN trigger has nothing to do with the performance of the application since it will be the same for every solution. The KEX and MEX both retrieve and commit data to the repository so that can also be combined into the same tests.

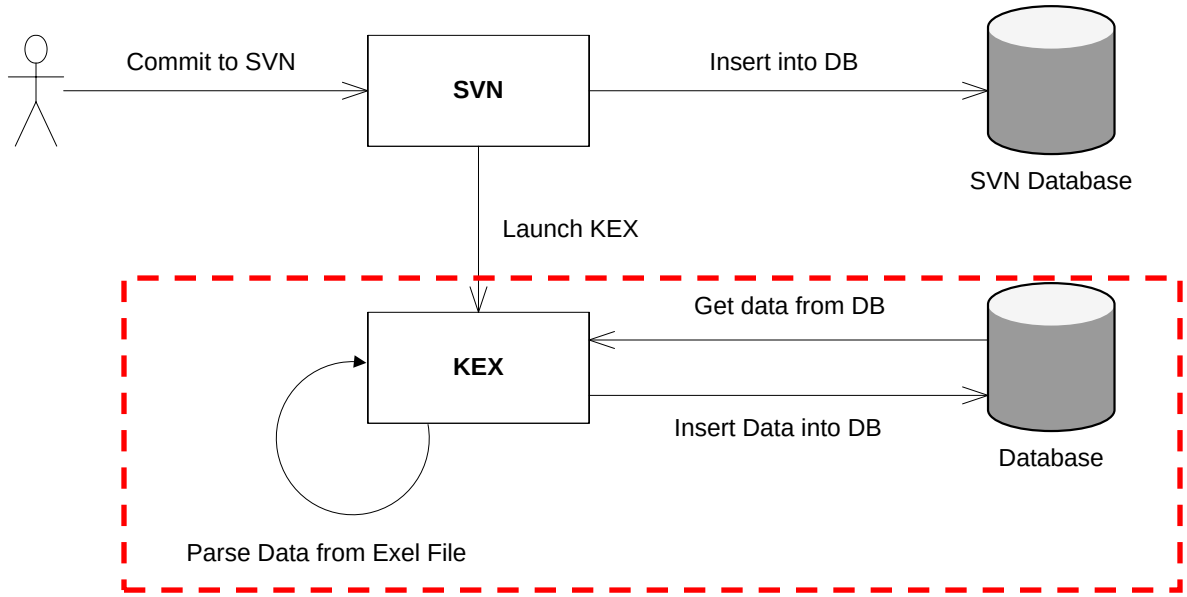


Figure 1 The logical architecture of the KEX production system

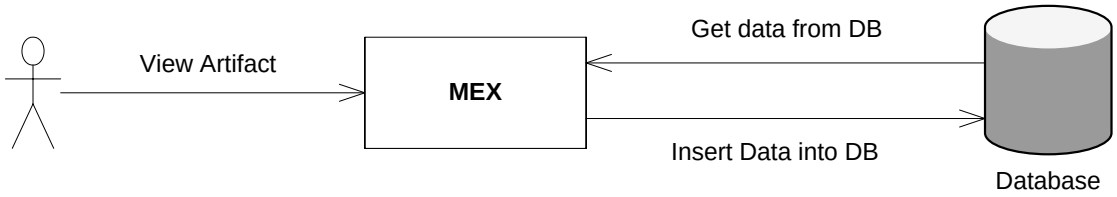


Figure 2 The logical architecture of the MEX production system

Next to the logical architecture there is also the physical architecture. Figure 3 shows this architecture for the production systems. The client performs a commit on his pc and the files are send to the SVN server. Here the trigger to the KEX is made, at the moment it is unclear if the KEX and the SVN will run on the same machine, for convenience we assume they are. There is also the case of the MEX where the MEX runs on the user system and communicates with the repository server. Since in the test environment there are limited resources it will consist of a single computer where everything is run. But the test environment does not include the SVN as can be seen in Figure 1, so a single pc is sufficient for testing.

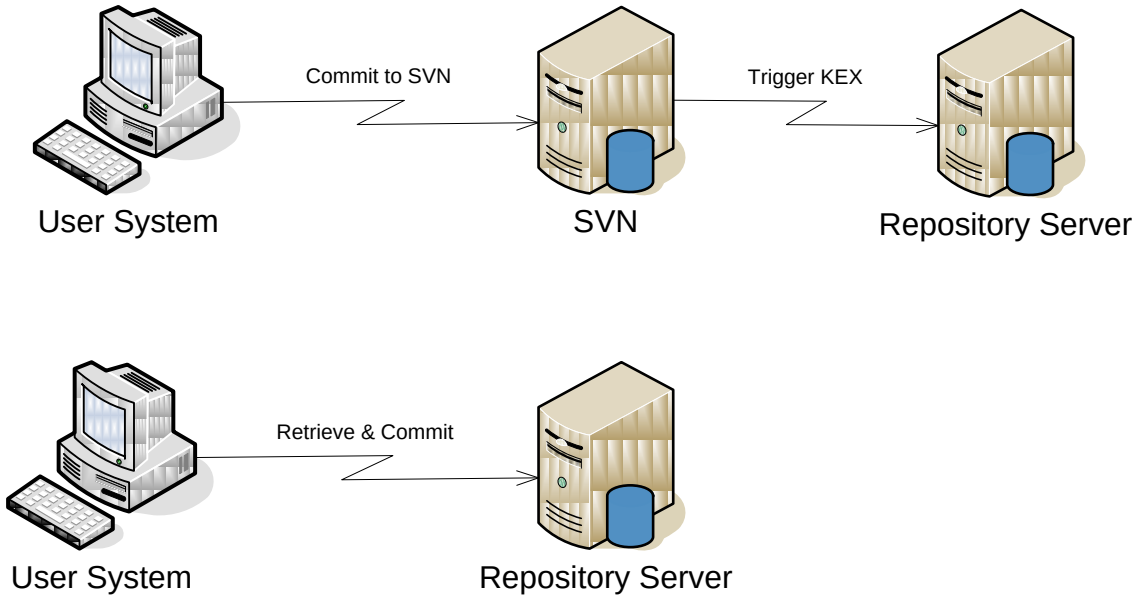


Figure 3 The physical architecture of the production systems

2.4.1 The test system

It would have been better to use a complete desktop system or even better a server for the testing environment. But due to limited resource availability we were only able to test everything using a laptop. In Tables Table 1 and Table 2 is the hardware and software specification of the test system. The production system will have better performance, but all tests are run on the laptop so we can make assumptions for the same tests running on a dedicated server.

Hardware specification	
System:	Dell latitude D600
Processor:	Intel Pentium Mobile (Centrino) 2.0 GHz
	32Kb L1 Cache
	32Mb L2 Cache
Front side Bus:	400 MHz
Memory:	1024Mb
	266 MHz
Hard disk:	ATA 100
	8Mb Cache
	60Gb
	5400 rpm
	Average seek time 12.5 ms

Table 1 Hardware Specification of the test system

Software specification	
Operating System:	Windows XP sp3
Java runtime:	JRE 1.6
SVN:	Visual SVN Server v1.0.1
Tools:	SVNKit 1.1.4
	Sesame 2.0.1
	ELMO 1.0 RC1
IDE:	Eclipse

Table 2 Software Specification of the test system

3 Identify performance acceptance criteria

Identifying the performance acceptance criteria is necessary to get a better understanding of the critical areas in an application and to what criteria these areas must comply. It can be seen as follows, where do we want to get performance measurements from, what criteria do apply to these measurements and which metrics do we use for them. Determining performance-testing objectives can be thought of in terms of the following activities [1]:

- Determine the objectives of the performance-testing effort.
- Capture or estimate resource usage targets and thresholds.
- Identify metrics.

3.1 Determine the objectives of the performance-testing effort.

The Objective of this performance-testing effort is to get a better view of the difference in performance of our proposed solutions. By testing the performance of the different proposed solutions, results can be compared to find the fastest solution. This comparison gives us one of the needed variables which will help us decide which solution will be implemented in the final system.

There are some bottlenecks that have to be kept in mind. For the SVN solutions this will probably be the I/O that will determine the maximum speed. For Sesame this will be the memory, the speed of the memory but also the available amount of memory is crucial for performance [2].

Models will be committed and updated inside the repository. This will be reflected in the objectives, tests have to be designed that commit new models into the repository, other tests update files models the repository. For updating models it is necessary to know how much change there is so we can design tests that approach a real life scenario. During his work at Astron [6] found that the average change that is made to model is about 20% of the total model. This is explained by the fact that changing a minor value below in the calculation, values throughout the entire model change.

3.2 Capture or estimate resource usage targets and thresholds.

It is important to capture or estimate some of the usage targets and thresholds. These targets and thresholds will keep the performance of the system optimal if they are met. The target is what the system does under normal circumstances and the threshold is where the system may peek to. One of the most common targets is the use of processing power. At it is generally accepted that a computer performs worse when it's CPU usage rises above 80%, we the define the following:

CPU Usage	Target 70%
	Threshold 80%

In our testing environment, everything runs on one system. In the production environment this will be different. The client will send the data to the server over the network. For the network I want to define a target of 80%, more load will result in slower speeds and this is not advised, and a threshold of 90%. This cannot be tested in our testing environment but must be taken into account when developing the production environment.

The system will be tested for SVN and Sesame implementation. Below for both implementations specific thresholds are defined.

SVN
The most crucial thing when using an SVN is the I/O performance. This is also the Bottleneck of the application so it will have no use to define Targets and Thresholds here since it will be performing at 100% all the time, if the speed of the CPU allows it. So the main thing we have to be aware of during the SVN testing is the CPU Usage to maximize our speed through the bottleneck. As defined above it is not preferable if the CPU usage rises above 80%. So the target will be set to 70% to increase maximum efficiency and not use an overpowered system. The threshold is 80% since above that the system performs a bit worse.

Sesame
The most crucial variable for Sesame is its memory usage. When more than 50% of memory is used the performance of Sesame will degrade [2] so therefore we set the Target to 50%. Nevertheless the system response time will still be good, so the Threshold will be a lot higher and can be set to 75%. Since all operations will be performed in the memory we expect the CPU to perform at 100% or near 100%.

3.3 Identify metrics.

Above defined are objective target and thresholds, but to be able to meet those proposed variables some metric need to be assigned to all of them. These metrics give means to measure and compare the results. All of the metrics apply to resource utilization and are fairly easy to measure.

For the threshold and target of CPU usage the “Windows Task Manager” is kept open during testing. From this the average CPU usage during the test can be read and noted down. We look at the average CPU usage during the different stages of the tests. If a threshold is needed peeks can also be recorded.

The task manager is also used for the targets and thresholds associated with memory. At the start and end of the performance test the memory of the application can be recorded from the task manager. To get even a better view of the memory usage, before and after each test we export all data from the Sesame repository. This way we get a better idea of how much memory is used for inferencing and how much is used to store data.

The metric that is most interesting is “time”. To be more precise the total time it takes for the system to perform one of the designed tests.

These tests are modeled for different proposed solutions. The results of those measurements can then be used to compare the performance of the different proposed solutions. To measure time, upon start of the test the system time is recorded. At the end of the test the time is recorded again. Subtract the start time from the end time and the result is the total time the test took.

Next to the metrics of time it is also preferable to check for scalability. Meaning, what is the effect of the growth of the repository has on the time it takes for a test to complete. And how large can our repository get before the performance is not acceptable anymore. We can calculate the scalability by doing performance testing with different repository seizes, and then make some predictions about it's behavior.

4 Plan and Design Tests

With the metrics designed the planning and designing of the needed tests can start. The tests will simulate the same behavior as the system will have during normal operation. These tests give results that will reflect the final production system. Therefore the tests have to reflect the reality as closely as possible else the results will be useless. The process that is used to identify the usage profiles that will be used in the performance testing is known as *workload modeling* [1]. We will do the workload modeling following the following steps:

- Identify the objectives.
- Identify key usage scenarios.
- Determine individual user data and variances.
- Determine the relative distribution of scenarios.
- Identify application load distribution.

4.1 Identify Objectives

The objective of the performance testing effort is to get an idea how the different proposed solutions perform. Two programs where considered to use as a repository, SVN and Sesame, to store the architectural versioning info in. The performance tests perform measurements during the testing of these two programs and their proposed solutions. The key objective is to get accurate measurements about how long it takes to insert, update or retrieve models from the repository. Also important is looking at the impact of the amount of data that is inside the repository and its effect on the performance.

4.2 Identify key usage scenarios

In our current test environment we identify six usage scenarios. The first two scenarios are for the SVN solutions, the last four are concerned with the Sesame solutions. The ones that use the SVN are; "Committing a new model into the SVN" and "Updating an existing model inside the SVN". For sesame we have defined four use cases.

- Committing a new model into Sesame using ELMO
- Committing a new model into Sesame using a RDF-file
- Committing a new model into Sesame using SeRQL
- Updating an existing model inside Sesame using SeRQL

The first three use cases test the speed for the different connection types we can have with Sesame, the fourth test is the update test done only with the fastest connection type from the first three tests! The connection types tested are:

- ELMO
- File Insert
- SeRQL

Use Case 1	Committing a new model into the SVN
Description	After the architect makes a new model he commits everything into the SVN. A trigger launches the KEX. The KEX parses the data from the model and after updating, inserts this into a local SVN repository. The entire model is than committed from the local repository into the SVN.
Actor	Architect of the model
Precondition	Connection with the SVN
Post condition	Model is present in the SVN
Steps	1. User inserts a new model into the SVN 2. KEX parses the data from the model 3. KEX updates the local repository 4. KEX inserts the data into the local repository 5. KEX commits the changes from the local repository into the SVN

Use Case 2	Updating an existing model inside the SVN
Description	The architect edits an existing model and commits it to the SVN. A trigger launches the KEX. The KEX parses the data from the model and after updating, inserts this into a local SVN repository. The changes from the model are then committed from the local repository into the SVN.
Actor	User who changes the model
Precondition	Connection with the SVN
Post condition	Model is present in the SVN
Steps	1. User inserts a updated model into the SVN 2. KEX parses the data from the model 3. KEX updates the local repository 4. KEX inserts the data into the local repository 5. KEX commits the changes from the local repository into the SVN

Use Case 3	Committing a new model into Sesame using ELMO
Description	After the architect makes a new model he commits everything into the SVN. A trigger launches the KEX. The KEX parses the data from the model and makes ELMO objects from each KE. These are automatically inserted into the remote Sesame repository by ELMO.
Actor	Architect of the model
Precondition	Connection with the SVN
Post condition	File is present in the SVN and model is present in Sesame
Steps	1. User inserts a new model into the SVN 2. KEX parses the data from the model

	3. KEX makes ELMO objects

Use Case 4	Committing a new model into Sesame using a RDF-file
Description	After the architect makes a new model he commits everything into the SVN. A trigger launches the KEX. The KEX parses the data from the model and makes ELMO objects from the KE's these objects are stored in a local Sesame Repository. The KEX extracts all data from the local repository into a file. The file is uploaded into the remote Sesame repository
Actor	Architect of the model
Precondition	Connection with the SVN
Post condition	File is present in the SVN and model is present in Sesame
Steps	1. User inserts a new model into the SVN 2. KEX parses the data from the model 3. KEX makes ELMO objects 4. KEX extracts all statements from the local Sesame Repository into a file 5. KEX commits the file into the remote Sesame repository

Use Case 5	Committing a new model into Sesame using SeRQL
Description	After the architect makes a new model he commits everything into the SVN. A trigger launches the KEX. The KEX parses the data from the model and inserts this data into a local constructed graph structure, this structure is then committed into the SVN.
Actor	Architect of the model
Precondition	Connection with the SVN
Post condition	File is present in the SVN and model is present in Sesame
Steps	1. User inserts a new model into the SVN 2. KEX parses the data from the model 3. KEX updates the local graph 4. KEX inserts the graph into the remote Sesame repository

Use Case 6	Updating an existing model inside Sesame using SeRQL
Description	After the architect makes a new model he commits everything into the SVN. A trigger launches the KEX. The KEX parses the data from the model and inserts this data into a local constructed graph structure, this structure is then committed into the SVN.
Actor	Architect of the model
Precondition	Connection with the SVN
Post condition	File is present in the SVN and model is present in Sesame

Steps	1. User inserts a new model into the SVN 2. KEX parses the data from the model 3. KEX updates the local graph 4. KEX inserts the graph into the remote Sesame repository

4.3 Determining Individual User Data and Variances

The individual user data and variance is not taken into account during the performance testing. Each user will make their own models and commit them to the system. However since the interest lies on time we can just as easily take an average model and perform all test with that. An average model consists of 1000 elements and change concerns about 20% of the elements. So an average model is used which represents these numbers. The tests generate data according to these values, so the tests are representative of the production environment negating the effect of individual user data and variances in that data. Of course the data in the production environment is not average, but by taking the averages the measurements that need to be done produce average results.

There is variation into the number of commits a user makes during the day, but this value has nothing to do with the performance of the system. It may only be relevant if we look at peak hours of committing, the numbers for committing of different users are not know so we cannot take them into account.

4.4 Determine the relative distribution of scenarios.

There are two scenarios that have to be taken into account, committing a new model or updating an existing model. Obviously updating is the scenario that is executed more often than committing. A user commits his files even when models are not totally finished at the end of the work day. Thus the next day there is an update, next to that slight changes are always made to models. We know that slight changes produce 20% of change inside a model. Assumed is that changing a model is done much more frequent than making new ones. This makes updating the most used scenario.

4.5 Identify Target Load Levels

At this moment we can only make some predictions about the load of the application. We know that the trigger is the commit operation on the SVN. Everyone will have a different point to commit and synchronize with the SVN. Depending on the time of committing and activity in the repository the load will be high. Assumed is that the most active point will be the end of the work day, everyone will commit their files at that point. Also before the afternoon break we assume an above average number of users will do a commit. However the users committing will notice nothing of this, and the users that are using the MEX will have their break during the assumed peak periods. The only time a user encounters speed problems is when he is retrieving data to explore using the MEX and another user insert a large amount of files into the SVN that triggers the KEX.

5 Configure test environment

To be able to run the tests correctly not only the test design needs to be good, also the environment must be configured correctly. For our tests the configuring of the environment was a fairly easy task. First look at all the services that are running on the test machine, and make sure that it is stable and the same over all the tests. Disabling the screensaver so that is does not start running during a test is always a good thing. Also kill and remove all applications and processes that have nothing to do with the testing. And turn of the virus scanner so that it does not interfere with actions that write to the disk or to the active memory.

Having taken those things into account also the "Java Virtual Machine" (JVM) needed some tweaking to ensure optimum performance. Default the JVM takes 300 Megabyte of memory. After a few tests SVN crashed due to an out of memory error. The option –Xmx500M was added to the run target of eclipse which ensured that the JVM had 500MB of available memory. This speeded up the performance a bit. 500 was the maximum since the machine only had 1 GB available and eclipse also took some of the memory the system as well. Above 500MB we found that performance dropped due to swapping with the hard drive. For the Sesame tests the same was then done, only we now gave tomcat 500MB instead of the default 300MB.

6 Implement test design

The implementation of the test design is done in Java using eclipse. Java has to be used for Sesame, since it is written in Java. For SVN we use the Java SVNKit to. Eclipse is used mainly because it is one of the best Java IDE's around and it is free. We take the following points into account [1] when we are implementing the model:

- Do not change your model without serious consideration simply because the model is difficult to implement in your tool.
- If you cannot implement your model as designed, ensure that you record the details about the model you do implement.
- Implementing the model frequently includes identifying metrics to be collected and determining how to collect those metrics

Java classes will be created for each of the, in chapter 3, defined use cases. When looking at the use cases we see that steps one and two are similar for all use cases and therefore will not be implemented into the tests. Since the time for steps one and two will always be the same for each model the overall performance will not be affected. Normally the SVN triggers the KEX to activate, the start of the test now does this. The metrics that the tests will gather have been defined in chapter 4. A log class is made which the test notifies at certain points during the testing, collecting data needed to satisfy the pre defined metrics. At the end of the test, the performance class outputs a performance log generating data collected during the tests.

All of the classes are configurable so we have some control over the testing, we can adjust variables as the amount of data present in the repository, how much data needs to be inserted, what connection type is needed and some more. This gives some flexibility when testing and may help to specify some tests on crucial points.

Tests are always run more than once. So there is also a parameter that sets how many times a test should run. This is done to get an average time instead of running the test only once. The average time counters the problem of an erratic sample.

7 Execute Tests

Before the execution of the test, check the test environment. The goal of the test environment is to mirror the execution environment as close as possible. The test environment is not a complete match for the production system as in respect to the full functionality. But the test environment is a good simulation of the actual processes that are going on in the execution environment. They simulate the most performance consuming processes and leave out the small one. The processes of updating and inserting complete models is simulated, however the KEX will update some minor things like one KE. This has no effect on performance and the user will not notice much when doing small stuff like this. This is then also not embedded in the performance tests. We can assume that our test environment matches our production environment close enough to produce the needed results.

Finally you can simulate some background activity to approach a real world scenario. In our case this was not necessary. We assume that the production environment will be run on a standalone server, so there will be no background activity.

Next to checking the test environment we also have to validate the tests. Instead of blindly accepting the outcomes of the test we make sure that the generated values are correct. Run a single test and inspect it step by step to see if the test does what is expected of it. System outs can be used to print intermediate data or an object inspector. System out where used for our test validation. Data validation is also a very good way to check if the tests perform as expected, this way it is made sure data is not transformed in some way during the testing. For SVN and Sesame all tests were done in a single run, for SVN we used the visual SVN program to inspect the data inside and compare it to the inserted data. For Sesame we used the web interface to randomly check some KE's after the inserting. Values were checked for consistency and compared with the input values. The inspections showed correct values so the tests were accepted as valid.

Here a short description how the SVN tests were executed. First I decided to run all tests 20x but after seeing the performance of the system and amount of time it would take I decided to bring this back to 10x. Also the original plan was to test with 0, 100, 500 and 1000 models inside the SVN. But the second test with 100 models already indicated that Java was out of memory. So we decided to allow 500Mb of memory. The decision was made then to take an increment of 50 from 0 – 300 and test all the values in between. The first given values where retested with the 500Mb of memory so see if this would make a performance difference. This was not the case but for consistency the new results were kept.

In the initial tests we did a complete commit of the entire local SVN directory. Committing only the directory where the new files of changed files are inserted seemed to increase the performance. So all tests where retested with this method.

In the Sesame we immediately took the 50 models increment that was changed in de SVN test. In the tests Tomcat was configured to run with 300Mb of available memory

space. After inserting 50 models it was obvious that this was not enough, so Tomcat was configured to use 500Mb of memory space. This allowed for more testing and a faster performance. This allowed for testing up to 250 models inside the repository. We exported the data from the repository before and after the test to see if it did not exceed the maximum amount of available memory.

The other metrics that had to be recorded where done so using the data presented in the "Windows Task Manager". This could just be monitored while the test was running. System outs indicated what part of the testing process was running.

8 Analyze, Report and Retest

Below the results of the six use cases has been plotted into graphs. The complete results can be found in appendix I. The results are divided over SVN and Sesame, Sesame is the sub divided over the ELMO, File and SeRQL use cases. First the results of the SVN tests are presented followed by the results of Sesame. Each section is divided into a result, analysis and improvements section. Finally the results are combined into two graphs. One for inserting and one for updating. All of the graphs show the average results over ten tests. The tests where done using an increment of 50 models inside the repository, this is plotted on the X-axis. On the Y-axis we see the time in seconds.

8.1 SVN

The SVN results are composed of two use cases. Use case 1: "Committing a new model into the SVN" and use case 2: "Updating an existing model inside the SVN". Below the results of both use cases is presented. Finally there is an improvements section that describes some methods to improve the speed.

8.1.1 Use Case 1: Committing a new model into the SVN

8.1.1.1 Results

The use case consists of three actions; Update, Insert and Commit. First the local repository has to be updated, the files have to be inserted there and then the repository has to be committed. Figure 4 shows the results of all three processes and the total time for all processes combined. Table 3 shows the CPU usage during the different processes.

CPU Usage		
	Average	**Peak**
Update	25%	-
Insert	35%	-
Commit	60%	100%

Table 3 Task Manager Results Use Case 1.

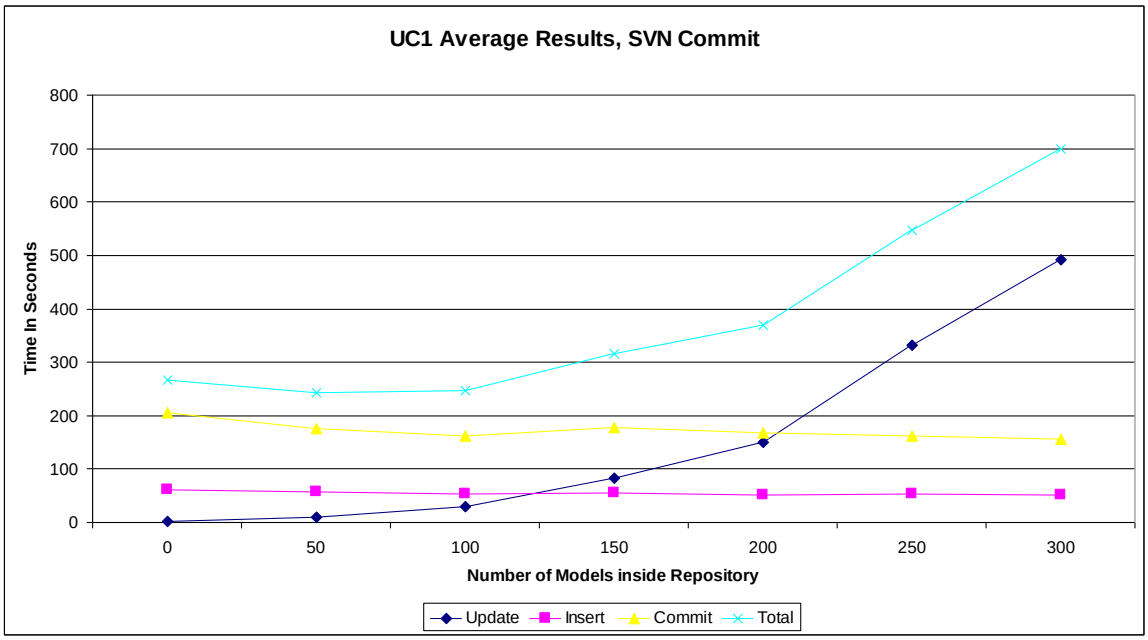


Figure 4 Use Case 1: Committing new models into the SVN

8.1.1.2 Analysis

Looking at the results it can be seen that the insert and commit operations stay constant over all the tests. This can be explained since the insert is done in the local directory which is the later committed to the SVN. This means that each test 1000 files get written to the hard drive, it was expected that there would be not much variance in time. As for the commit each test a commit of 1000 files is done, since this number of files is constant the time also is constant. The number of files inside the repository has no effect on the commit time. We do see that a commit operation takes a considerable amount of time. A single commit consists of 1000 files (KE's) for each file SVN has to make a new version log, thus taking a long time. It has nothing to do with the amount of data pushed into the SVN since one commit consists of about 4 – 5 MB of data. The only changing factor is the update, the more models inside the repository the longer an update takes, and this was expected. Since the SVN has to check for all files inside the repository which ones are changed.

Looking back at the set targets and thresholds from chapter 3 and comparing them with Table 3 the requirements are more that met. The update takes 25% of CPU and the commit is 60% on average. We see a peak of 100% on the commit operation, this only lasted for a few seconds during the start of the test and can be explained by the caches of the hard drive and the buffer of the SVN. It was only such a short amount of time it would have little effect on the overall performance.

8.1.2 Use Case 2: Updating an existing model inside the SVN

8.1.2.1 Results

This use case also consists of three actions; Update, Insert and Commit. First the local repository has to be updated, the changed files have to be inserted there and then the

repository has to be committed. Figure 5 shows the results of all three processes and the total time for all processes combined. Table 4 shows the CPU usage during the different processes.

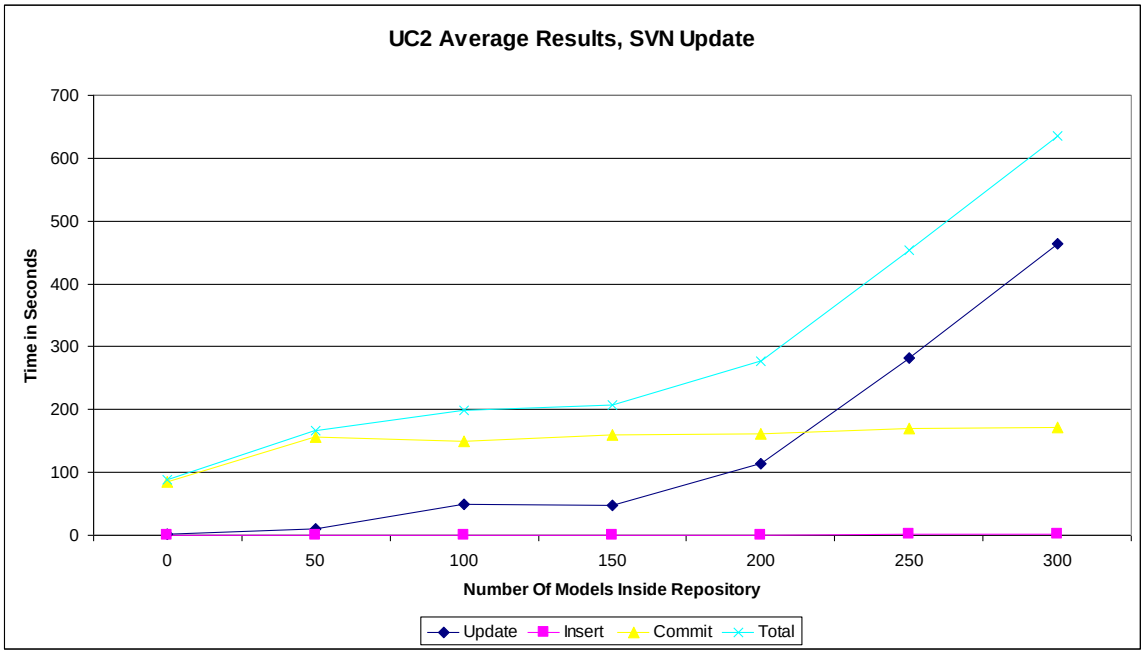


Figure 5 Use Case 2: Updating an existing model inside the SVN

CPU Usage			
	Average		**Peak**
Update	25%		-
Insert	-		-
Commit	60%	75%	100%

Table 4 Task Manager Results Use Case 2.

8.1.2.2 Analysis

As mentioned in Use Case 1 the insert and commit operations stay constant and the update is the only changing factor. The insert operation is so fast that the time can almost not be measured. In respect to Use Case 1, we now insert only 20% of change instead of the full 1000 KE's. The commit is about the same size, since everything has to be committed and SVN has to calculate what is changed and what stays the same.

Here we also take a look at the targets and thresholds and comparing with the results in Table 4. As with use case 1 the conclusion is that everything is ok. We only see here that the commit operation takes 75% instead of 60% after the second test. This however stays within the range of our target. The update did not peak it stayed stable around 25%.

8.1.3 Improvements

In this section there are some remarks how to improve the performance of the SVN tests. This must be seen as recommendations since no official tests where do to validate the impacts to the performance of these improvements. Expected is that the biggest improvement in speed would be a faster hard drive. The test environment had a hard drive of 5400 rpm, a normal desktop or even a server can be equipped with drives that have 7200 or even 10 to 15K hard drives. Those drives have much better performance and seek times. Also a solid state disk could be used with an even better seek time and performance. Secondly a switch from windows to linux could improve performance, since it is known that windows sandboxes Java. Finally compiling the Java code to run native instead of inside the JVM increases performance.

8.2 Sesame

This section contains the results of the four Sesame use cases:

- Committing a new model into Sesame using ELMO
- Committing a new model into Sesame using an RDF-file
- Committing a new model into Sesame using SeRQL
- Updating an existing model inside Sesame using SeRQL

At the end there is an improvements section, here some improvements are given which make the solutions perform better.

8.2.1 Use Case 3: Committing a new model into Sesame using ELMO

8.2.1.1 Results

In Table 5 the results of the first test with ELMO are presented. The test consisted of inserting 10 models into an empty repository. The average speed is so slow that the remainder of the tests is not executed. Inserting into an empty repository takes on average 100 minutes per model. In practice we are not able to work with these numbers.

Test 1

Test Number	Time to add 1 Artifact
1	2566
2	6503
3	5649
4	8276
5	7688
6	7422
7	5024
8	4759
9	4488
10	4437
Average time	5681.2

Table 5 ELMO Results

8.2.1.2 Analysis

Described her is an explanation why ELMO seems to be so slow. The test was done using all of the different repository connection types. It made not much difference. The remote connection might be the slowing factor. But inserting the KE's into a local repository was also slow. This ruled out the connection as the problem area. Adding one artifact seemed to be relatively fast, but when the second one was inserted speeds dropped considerably. Everything seemed to point at the inferencer, disabling the inferencer speeded up the process very much. When looking at ELMO objects, they are in direct connection with the repository. Each time we make an KE, we open a connection to the

repository and each time we add a value the inferencer is triggered. Assumed this is the slowing factor. ELMO also used 100% of CPU, 90% was used by Tomcat and 10% by the test application Therefore it is decided that ELMO is not usable for inserting lots of data into a repository. For editing objects it is very handy.

8.2.2 Use Case 4: Committing a new model into Sesame using an RDF-file

8.2.2.1 Results

Committing a file into the repository is a fast process. The process consists of three actions, inserting the data into the local repository, exporting it into a file and then committing that file into the remote repository. Figure 5 shows the results.

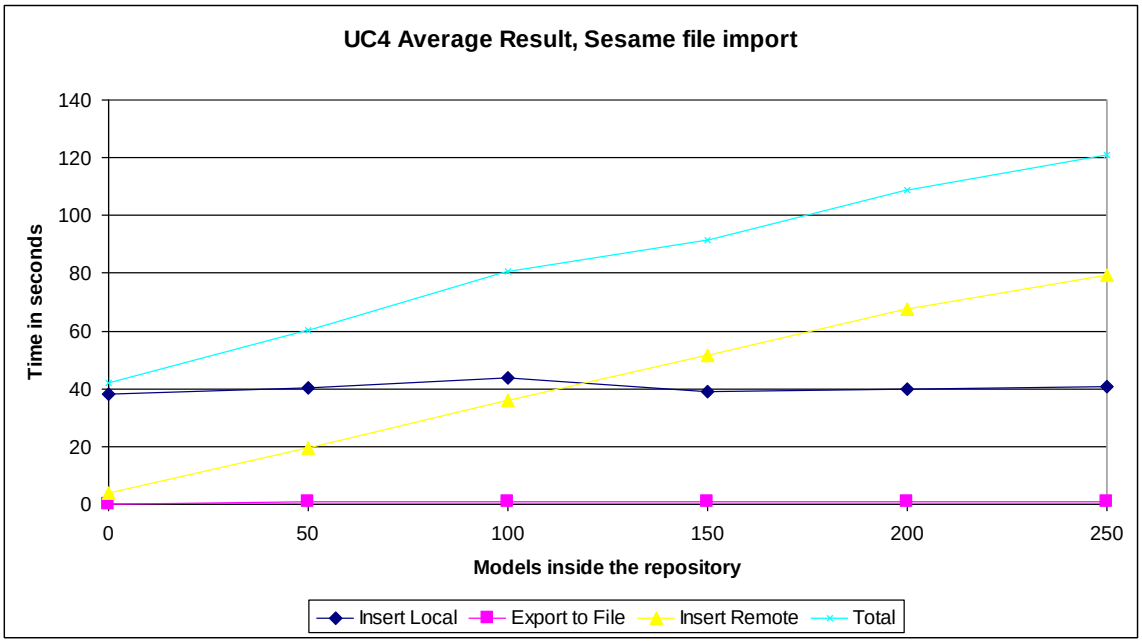


Table 6 Use Case 4: Committing a new model into Sesame using and RDF-File

CPU Usage	
local insert	100%
remote insert	100%

Table 7 Task Manager Results Use Case 4.

Memory Usage			
	Before	**After**	**Exported**
Test 1	35	75	19
Test 2	219	251	110
Test 3	355	360	202
Test 4	372	438	294
Test 5	509	509	387
Test 6	508	540	479

Table 8 Task Manager Results Use Case 4.

8.2.2.2 Analysis

File importing far out performs ELMO. The inserting into the local repository is bit slow but stable process. This is because it is always the same amount of data. The same goes for the export; it is always the same amount of data so the results are expected to be stable. The only changing variable is the inserting into the remote repository. We see that it takes more time as there are more models inside the repository. This is the result that was expected.

After the fourth test the memory target has been reached, the threshold is reached after the fifth test. However Sesame still has some memory left for inferencing after test 6, since the export from the repository only shows 480MB of data, which leaves 20MB for inferencing. When inserting a few more models Sesame is out of memory and then the speed goes way up. So testing was stopped after 250 models inside the repository. As for the CPU usage it was 100% as expected.

8.2.3 Use Case 5: Committing a new model into Sesame using SeRQL

8.2.3.1 Results

Committing a new model consists of two processes, inserting the data into a local graph and committing that graph into the remote repository. Figure 6 shows the result of the use case, it shows only the result of the three different connection types not for the individual processes, for those results look in appendix I. Also the CPU and memory usage results are presented.

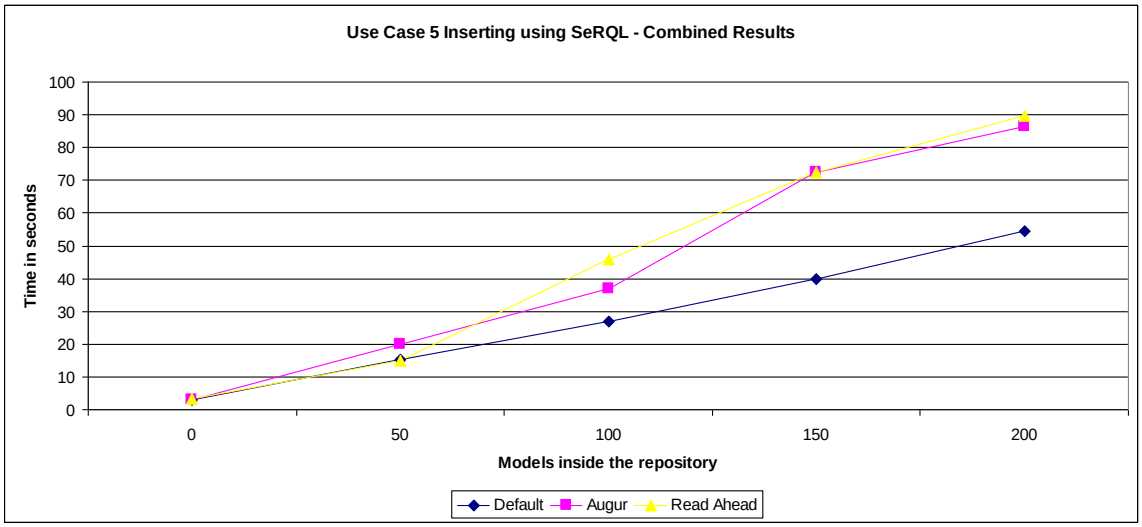


Figure 6 Use Case 5: Committing a new model into Sesame using SeRQL

CPU Usage	
local insert	100%
remote insert	100%

Table 9 Task Manager Results Use Case 5

Memory Usage			
	Before	**After**	**Exported**
Test 1	36	84	18
Test 2	207	250	109
Test 3	377	386	202
Test 4	509	461	296
Test 5	510	510	388

Table 10 Task Manager Results Use Case 5.

8.2.3.2 Analysis

Above was mentioned we left out the processes individually, however the insert takes about zero time and all time comes from the commit operation. The results of the sixth test where left out intentionally for the scalability of the graph. The graph shows that the default connection type is the fastest of the three. The default connection processes each request and does not cache anything; an augur connection type tracks the requests and anticipates related information. It is best used when the results are expected to fit into memory and not all properties will be read [2]. The read ahead connection type reduces the number of hits to the repository for the same subject. It is best used when the complete results may not fit into memory and most of the concept properties will be retrieved [2]. It is obvious that the augur and read ahead connection type are optimized for retrieving data instead of sending it. This is why we also see that the default connection type outperforms them easily.

The same targets and thresholds apply here as in use case 4, the results are very similar. The export is actually the same, since the repository contains the same data. Memory usage differs somewhat but also is close to each other.

8.2.4 Use Case 6: Updating a model inside Sesame using SeRQL

8.2.4.1 Results

Updating a model consists of three processes instead of the two needed for inserting. First we have to retrieve the data from the repository, then the change has to be calculated and then the changes have to be committed to the repository. Figure 7 shows the results for this use case, as use case 5 not the full results are shown only the totals from the three different connection types.

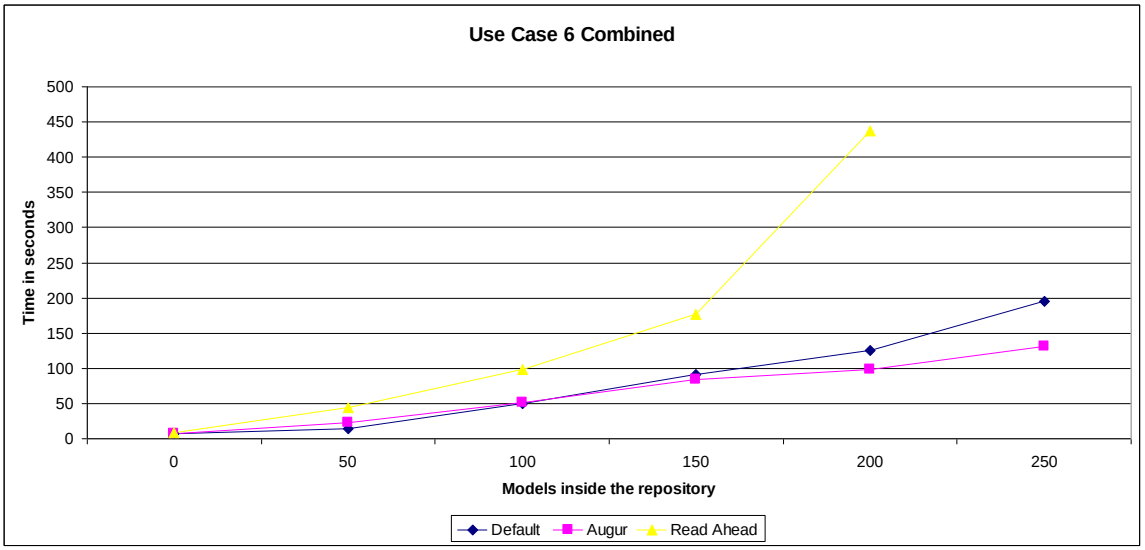


Figure 7 Use Case 6: Updating a model inside Sesame using SeRQL

Memory Usage			
	Before	**After**	**Exported**
Test 1	46	178	19
Test 2	204	221	111
Test 3	248	265	203
Test 4	341	516	294
Test 5	439	454	386

Table 11 Task Manager Results Use Case 6.

CPU Usage	
local insert	100%
remote insert	100%

Table 12 Task Manager Results Use Case 6.

8.2.4.2 Analysis

The Read Ahead connection type is clearly the slowest of the three in this process. The sixth test of the Read Ahead is left out for scalability of the Graph. Default and Augur in the beginning show about the same speeds. Default has a slightly faster commit and Augur a faster retrieve process, this is where Augur wins in the later tests. The augur and read ahead connection types are optimized for retrieving data. Here it then also shows that we do see a big difference between them. Read ahead performs the worst, it is supposed to reduce the number of hits for the same subject; however each subject is only called once. So it does not apply and probably the read ahead caches to much unneeded stuff. The auger does outperform the default connection. It tracks requests and anticipates

on them. This probably improves the performance when checking out one complete model, since it is build up of 1000 KE's .

The same targets and thresholds apply here as in use case 4, the results are very similar. The export is actually the same, since the repository contains the same data. Memory usage differs somewhat but also is close to each other.

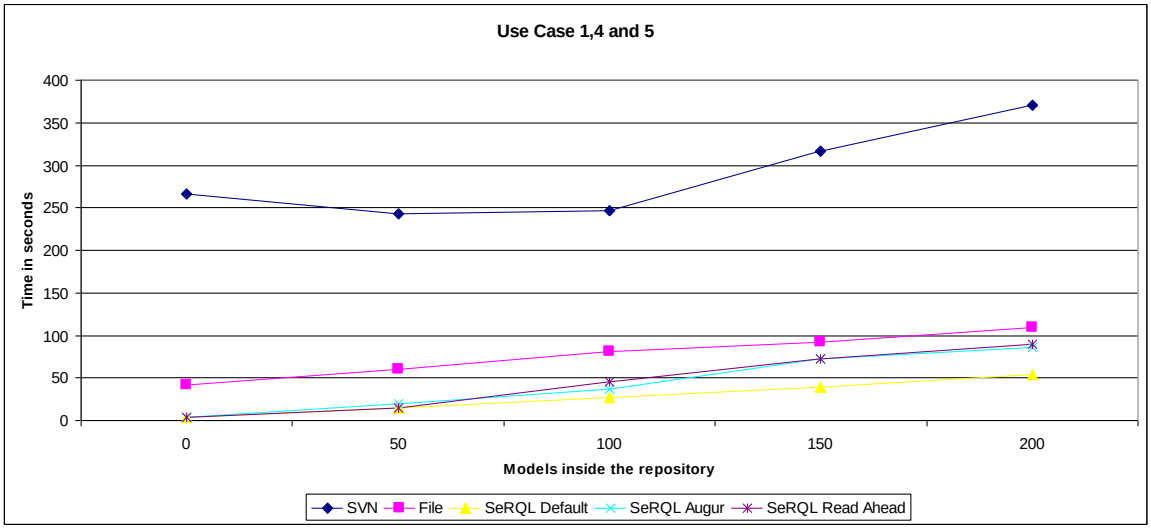
8.2.5 Improvements

As was said for the improvement section for SVN we have no actual results of how much the performance will improve. Sesame relies for its most part on memory since the repository is memory based. So the main point of focus for upgrading the performance is installing faster memory. DDR3 is now getting on the market so this would greatly improve performance due to faster access and read times. Increasing the amount of available memory gives us a higher target and even higher threshold and leaves Sesame with more working memory. And as said for SVN we could migrate to Linux since Java is sandboxed inside Windows and compiling to native code could also increase the performance.

8.3 The Combined Results

Above the results for all the separate use cases was plotted. The first graph below here shows the result of all the use cases, with the exception of use case three, which commit models into a repository. The second graph does the same for all use cases that update models inside a repository. The first graph scales until 200 models this is done for scalability reasons. For the second graph only the results of the read ahead with 250 models is left out.

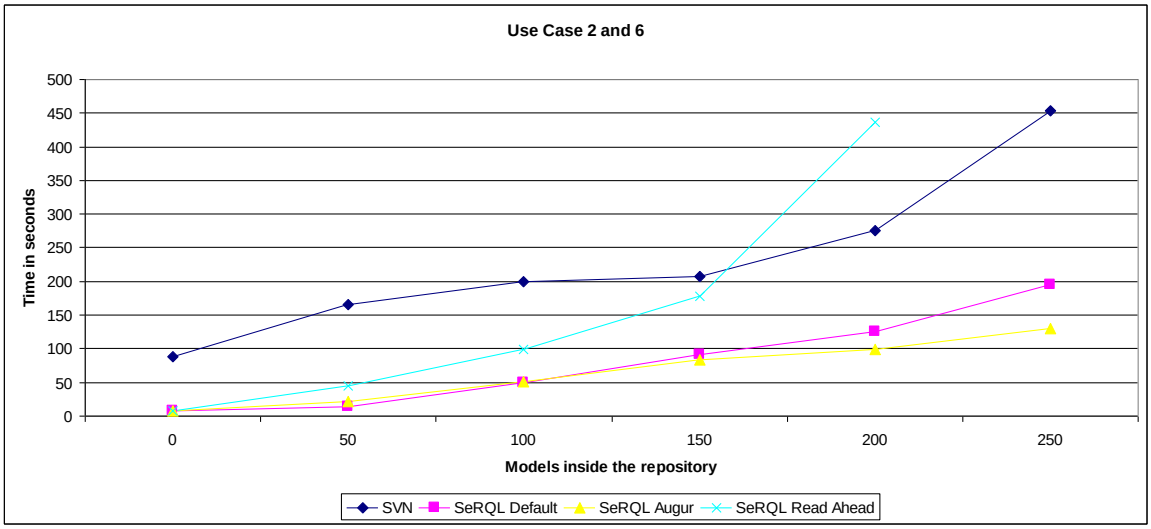
8.3.1 The committing use cases



8.3.1.1 Analysis

Clearly it can be seen that SVN is outperformed by Sesame. And looking at Sesame the file inserting is the slowest as was expected from the results. The inserting into the local repository was the slowing factor. From the three different repository types the default is the fastest.

8.3.2 The updating use cases



8.3.2.1 Analysis

For updating Sesame is also faster than SVN. Unless for the case of the read ahead repository after the fourth test. This is due to a lack of memory and a wrong repository type. Default and Augur go head to head until the fourth use case then augur becomes faster since it has a faster retrieve than sesame.

8.4 Retest

Retesting some of the results is done to validate our results. If the test show about the same results as before, the results would be repeatable and therefore valid. Random tests were selected from all of the use cases, with the exception of use case three. All found measurements where comparable with our previous results. Therefore the results are accepted as valid.

9 Conclusion

Looking at the combined results it can be clearly seen that Sesame has the upper hand when it comes to performance. What not can be seen is that SVN out performs Sesame when Sesame runs out of memory. Those tests were not completed since they just took so much time. Sesame is the better option but make sure that there is enough available memory. This can be clearly seen when looking at the used memory. For the connection type default is recommended, it performs the best when seen over the combined results. Better would be to have a system that can switch between both connections. Then default can be used for committing and augur for updating. Keep in mind that switching connection types also takes some time.

One of the other things to look at was scalability. When we compare both options, Sesame and SVN we see that SVN can be scaled up much further than Sesame. It can easily handle thousands of models, since it stores everything on the hard drive. But the system will be notoriously slow. Sesame scales up pretty good as long as there is a machine with lots of available memory like a big server. For production uses it is also possible to have a per project repository so overall speeds can even be increased more.

10 Literature

1. Meier, Farre, Bansode, Barber, Rea, *Performance Testing Guidance for Web Applications*, August 2007
2. The Sesame website, http://www.openrdf.org/doc/sesame/users/ch01.html
3. The ELMO website, http://www.openrdf.org
4. The Open RDF website, http://www.openrdf.org
5. J. Rasmussen, *Understanding Software Architectures: Tracing Architectural Knowledge In Software Architecture Documentation*, 2009
6. T. de Vries, *Architectural Knowledge in Quantitative Architecture Analysis, 2008*
7. The Tortoise Website, http://tortoisesvn.tigris.org/
8. *The SVN website,* http://subversion.tigris.org/

Appendix I Complete Test Results

Use case 1 Inserting a new model into the SVN

Test 1 — Committing 0 - 10 models

Test Nr	Update	Insert	Commit	Total Time
1	1	65	226	292
2	0	61	228	289
3	0	60	213	273
4	1	67	214	282
5	0	67	200	267
6	1	61	208	270
7	2	56	193	251
8	2	58	208	268
9	2	61	183	246
10	2	53	173	228
Averages	1,1	60,9	204,6	266,6

Test 2 — Committing 50 - 60 models

Test Nr	Update	Insert	Commit	Total Time
1	12	62	172	246
2	10	57	180	247
3	9	60	178	247
4	9	58	176	243
5	11	57	174	242
6	9	58	178	245
7	11	60	168	239
8	12	56	175	243
9	10	57	178	245
10	12	55	171	238
Averages	10,5	58	175	243,5

Test 3 — Committing 100 - 110 models

Test Nr	Update	Insert	Commit	Total Time
1	43	65	175	283
2	39	53	166	258
3	23	53	169	245
4	21	50	152	223
5	21	50	155	226
6	21	51	163	235
7	21	51	157	229
8	23	55	167	245
9	44	51	160	255
10	45	55	164	264
Averages	30,1	53,4	162,8	246,3

Test 4 — Committing 150 - 160 models

Test Nr	Update	Insert	Commit	Total Time
1	118	62	221	401
2	89	60	182	331
3	31	60	182	273
4	32	58	179	269
5	29	58	177	264
6	31	53	178	262
7	114	53	169	336
8	122	51	160	333
9	118	53	174	345
10	136	53	163	352
Averages	82	56,1	178,5	316,6

Test 5 — Committing 200 - 210 models

Test Nr	Update	Insert	Commit	Total Time
1	118	62	171	351
2	145	50	170	365
3	151	55	168	374
4	153	54	225	432
5	155	52	110	317
6	156	50	137	343
7	145	52	171	368
8	154	47	186	387
9	159	50	169	378
10	164	47	175	386
Averages	150	51,9	168,2	370,1

Test 6 — Committing 250 - 260 models

Test Nr	Update	Insert	Commit	Total Time
1	269	60	162	491
2	238	52	170	460
3	381	55	161	597
4	395	50	161	606
5	334	50	157	541
6	303	53	160	516
7	353	54	185	592
8	423	52	156	631
9	361	53	159	573
10	267	50	156	473
Averages	332,4	52,9	162,7	548

Average CPU Usage	
Update	15 - 30%
Insert	Too fast to measure
Commit	50 - 70%

Use case 2 Updating an existing model inside the SVN

Test 1 — 10 models inside the SVN

Test Nr	Update	Insert	Commit	Total Time
1	2	0	95	97
2	3	0	88	91
3	3	0	83	86
4	2	0	84	86
5	3	0	85	88
6	2	0	87	89
7	2	0	84	86
8	3	0	82	85
9	3	0	82	85
10	2	0	82	84
Averages	2,5	0	85,2	87,7

Test 2 — 50 models inside the SVN

Test Nr	Update	Insert	Commit	Total Time
1	4	0	172	176
2	14	1	171	186
3	10	1	164	175
4	10	0	112	122
5	11	1	167	179
6	10	1	168	179
7	11	0	173	184
8	11	0	161	172
9	10	0	157	167
10	9	0	112	121
Averages	10	0,4	155,7	166,1

Test 3 — 100 models inside the SVN

Test Nr	Update	Insert	Commit	Total Time
1	48	0	168	216
2	115	1	103	219
3	43	1	191	235
4	45	0	175	220
5	63	1	110	174
6	46	1	153	200
7	44	0	163	207
8	43	1	164	208
9	19	1	110	130
10	22	0	161	183
Averages	48,8	0,6	149,8	199,2

Test 4 — 150 models inside the SVN

Test Nr	Update	Insert	Commit	Total Time
1	34	0	160	194
2	43	1	166	210
3	29	0	156	185
4	31	1	165	197
5	31	0	167	198
6	39	2	156	197
7	73	1	157	231
8	64	1	155	220
9	64	0	159	223
10	64	0	160	224
Averages	47,2	0,6	160,1	207,9

Test 5 — 200 models inside the SVN

Test Nr	Update	Insert	Commit	Total Time
1	135	2	171	308
2	126	0	166	292
3	112	0	121	233
4	112	1	168	281
5	112	1	179	292
6	100	1	143	244
7	106	0	175	281
8	114	0	135	249
9	113	1	175	289
10	114	0	179	293
Averages	114,4	0,6	161,2	276,2

Test 6 — 250 models inside the SVN

Test Nr	Update	Insert	Commit	Total Time
1	314	1	161	476
2	186	0	183	369
3	292	3	167	462
4	334	2	187	523
5	334	0	157	491
6	273	1	168	442
7	292	1	176	469
8	348	1	163	512
9	274	1	172	447
10	171	1	165	337
Averages	281,8	1,1	169,9	452,8

Average CPU Usage Test 1 & 3	
Update	15 - 30%
Insert	Too fast to measure
Commit	50 - 70%

Average CPU Usage Test 2, 4, 5 & 6	
Update	15 - 30%
Insert	Too fast to measure
Commit	70 - 85%

Use case 3 Inserting a new model into Sesame using ELMO

Test 1 **0 - 10 models**

Test Number	Time to add 1 Artifact
1	2566
2	6503
3	5649
4	8276
5	7688
6	7422
7	5024
8	4759
9	4488
10	4437

Averages 5681,2

Data read from the Task Manager
Average CPU usage during tests 100%

Use case 4 Inserting a new model into Sesame using a RDF-file

Test 1 **Commit 0-10 models**

Test Nr	Insert Local	Export to file	Insert Remote	Total Time
1	39	0	3	42
2	38	0	3	41
3	38	0	3	41
4	38	0	3	41
5	38	0	4	42
6	38	0	4	42
7	38	0	4	42
8	38	0	4	42
9	38	0	5	43
10	38	0	5	43
Averages	38,1	0	3,8	41,9

Test 2 **Commit 50 - 60 models**

Test Nr	Insert Local	Export to file	Insert Remote	Total Time
1	43	1	18	62
2	39	1	18	58
3	39	1	18	58
4	39	0	19	58
5	39	1	19	59
6	39	1	20	60
7	46	1	20	67
8	39	1	21	61
9	39	1	20	60
10	39	1	21	61
Averages	40,1	0,9	19,4	60,4

Test 3 — Committing 100 – 110 models

Test Nr	Insert Local	Export to file	Insert Remote	Total Time
1	44	1	35	80
2	40	1	35	76
3	40	1	35	76
4	40	1	35	76
5	40	1	36	77
6	39	1	36	76
7	60	1	36	97
8	55	1	37	93
9	40	1	37	78
10	39	1	37	77
Averages	43,7	1	35,9	80,6

Test 4 — Committing 150 - 160 models

Test Nr	Insert Local	Export to file	Insert Remote	Total Time
1	45	1	51	97
2	39	1	50	90
3	39	1	50	90
4	39	1	51	91
5	39	1	51	91
6	38	1	51	90
7	38	1	52	91
8	38	1	52	91
9	38	1	52	91
10	38	1	54	93
Averages	39,1	1	51,4	91,5

Test 5 — Committing 200 - 210 models

Test Nr	Insert Local	Export to file	Insert Remote	Total Time
1	43	1	67	111
2	40	1	66	107
3	40	1	66	107
4	40	1	67	108
5	39	1	67	107
6	39	1	67	107
7	39	1	68	108
8	39	1	72	112
9	40	1	69	110
10	39	1	69	109
Averages	39,8	1	67,8	108,6

Test 6 — Committing 250 - 260 models

Test Nr	Insert Local	Export to file	Insert Remote	Total Time
1	42	1	76	119
2	40	1	77	118
3	40	1	78	119
4	44	1	78	123
5	43	1	79	123
6	41	1	80	122
7	40	1	80	121
8	39	1	81	121
9	39	1	82	122
10	39	1	83	123
Averages	40,7	1	79,4	121,1

CPU Usage	
local insert	100%
remote insert	100%

Memory Usage			
	Before	**After**	**Exported**
Test 1	35	75	19
Test 2	219	251	110
Test 3	355	360	202
Test 4	372	438	294
Test 5	509	509	387
Test 6	508	540	479

Use case 5 Inserting a new model into Sesame using SeRQL
Default Connection Type

Test 1 — **Committing 0 - 10 models**

Test Nr	Insert	Commit	Total Time
1	0	2	2
2	0	2	2
3	0	3	3
4	0	3	3
5	0	3	3
6	0	3	3
7	0	3	3
8	0	4	4
9	0	4	4
10	0	4	4
Averages	0	3,1	3,1

Test 2 — **Committing 50 - 60 models**

Test Nr	Insert	Commit	Total Time
1	0	14	14
2	0	14	14
3	0	15	15
4	0	15	15
5	0	15	15
6	0	15	15
7	0	15	15
8	0	17	17
9	0	16	16
10	0	16	16
Averages	0	15,2	15,2

Test 3 — **Committing 100 - 110 models**

Test Nr	Insert	Commit	Total Time
1	0	26	26
2	0	26	26
3	0	26	26
4	0	26	26
5	0	27	27
6	0	27	27
7	0	27	27
8	0	27	27
9	0	28	28
10	0	28	28
Averages	0	26,8	26,8

Test 4 — **Committing 150 - 160 models**

Test Nr	Insert	Commit	Total Time
1	0	44	44
2	0	38	38
3	0	39	39
4	0	39	39
5	0	39	39
6	0	39	39
7	0	40	40
8	0	40	40
9	0	40	40
10	0	40	40
Averages	0	39,8	39,8

Test 5 — **Committing 200 - 210 models**

Test Nr	Insert	Commit	Total Time
1	0	51	51
2	0	54	54
3	0	54	54
4	0	51	51
5	0	55	55
6	0	56	56
7	0	52	52
8	0	52	52
9	0	63	63
10	0	56	56
Averages	0	54,4	54,4

Test 6 — **Committing 250 - 260 models**

Test Nr	Insert	Commit	Total Time
1	0	63	63
2	0	77	77
3	0	63	63
4	0	76	76
5	0	63	63
6	0	124	124
7	0	64	64
8	0	79	79
9	0	78	78
10	0	140	140
Averages	0	82,7	82,7

Results using Augur Connection Type

Test 1 — **Committing 0 - 10 models**

Test Nr	Insert	Commit	Total Time
1	0	2	2
2	0	2	2
3	0	3	3
4	0	3	3
5	0	3	3
6	0	3	3
7	0	3	3
8	0	4	4
9	0	4	4
10	0	4	4
Averages	0	3,1	3,1

Test 2 — **Committing 50 - 60 models**

Test Nr	Insert	Commit	Total Time
1	0	15	15
2	0	14	14
3	0	15	15
4	0	15	15
5	0	15	15
6	0	15	15
7	0	15	15
8	0	15	15
9	0	15	15
10	1	64	65
Averages	0,1	19,8	19,9

Test 3 — **Committing 100 - 110 models**

Test Nr	Insert	Commit	Total Time
1	0	28	28
2	0	27	27
3	0	28	28
4	0	48	48
5	0	27	27
6	0	27	27
7	1	28	29
8	1	28	29
9	1	28	29
10	1	95	96
Averages	0,4	36,4	36,8

Test 4 — **Committing 150 - 160 models**

Test Nr	Insert	Commit	Total Time
1	0	38	38
2	0	38	38
3	0	38	38
4	0	39	39
5	0	59	59
6	0	108	108
7	1	115	116
8	1	52	53
9	0	123	123
10	0	111	111
Averages	0,2	72,1	72,3

Test 5 — **Committing 200 - 210 models**

Test Nr	Insert	Commit	Total Time
1	0	50	50
2	0	50	50
3	0	50	50
4	0	51	51
5	0	54	54
6	0	62	62
7	1	168	169
8	1	61	62
9	0	145	145
10	0	170	170
Averages	0,2	86,1	86,3

Test 6 — **Committing 250 - 260 models**

Test Nr	Insert	Commit	Total Time
1	0	77	77
2	0	77	77
3	0	168	168
4	1	108	109
5	0	196	196
6	2	117	119
7	0	989	989
8	0	1308	1308
9	0	1740	1740
10	0	1376	1376
Averages	0,3	615,6	615,9

Results using Read Ahead Connection Type

Test 1 — Committing 0 - 10 models

Test Nr	Insert	Commit	Total Time
1	0	2	2
2	0	2	2
3	0	3	3
4	0	3	3
5	0	3	3
6	0	3	3
7	0	4	4
8	0	4	4
9	0	4	4
10	0	4	4
Averages	0	3,2	3,2

Test 2 — Committing 50 - 60 models

Test Nr	Insert	Commit	Total Time
1	0	14	14
2	0	14	14
3	0	14	14
4	0	16	16
5	0	15	15
6	0	15	15
7	0	15	15
8	0	15	15
9	0	16	16
10	0	16	16
Averages	0	15	15

Test 3 — Committing 100 - 110 models

Test Nr	Insert	Commit	Total Time
1	0	26	26
2	0	26	26
3	0	26	26
4	0	26	26
5	0	27	27
6	0	27	27
7	0	27	27
8	0	27	27
9	0	102	102
10	3	142	145
Averages	0,3	45,6	45,9

Test 4 — Committing 150 - 160 models

Test Nr	Insert	Commit	Total Time
1	0	38	38
2	0	38	38
3	0	39	39
4	0	39	39
5	0	39	39
6	0	42	42
7	0	40	40
8	0	119	119
9	1	159	160
10	1	169	170
Averages	0,2	72,2	72,4

Test 5 — Committing 200 - 210 models

Test Nr	Insert	Commit	Total Time
1	0	49	49
2	0	49	49
3	0	233	233
4	1	52	53
5	0	50	50
6	0	50	50
7	0	50	50
8	0	51	51
9	1	257	258
10	1	52	53
Averages	0,3	89,3	89,6

Test 6 — Committing 250 - 260 models

Test Nr	Insert	Commit	Total Time
1	0	594	594
2	0	977	977
3	0	1145	1145
4	1	1751	1752
5	0	1508	1508
6	0	1513	1513
7	0	1392	1392
8	0	1347	1347
9	0	1307	1307
10	0	1426	1426
Averages	0,1	1296	1296,1

Memory usage only recorded for the default test cases

Memory Usage			
	Before	**After**	**Exported**
Test 1	36	84	18
Test 2	207	250	109
Test 3	377	386	202
Test 4	509	461	296
Test 5	510	510	388

CPU Usage	
local insert	100%
remote insert	100%

Use case 6 Updating an existing model inside Sesame using SeRQL

Results using Default Connection Type

Test 1 **Committing 0 - 10 models**

Test Nr	Retrieve	Calculate	Commit	Total Time
1	1	4	3	7
2	1	3	2	5
3	1	3	3	6
4	1	3	3	6
5	1	0	5	5
6	2	3	3	6
7	2	3	3	6
8	2	3	3	6
9	2	3	3	6
10	2	3	3	6
Averages	1,5	2,8	3,1	5,9

Test 2 **Committing 50 - 60 models**

Test Nr	Retrieve	Calculate	Commit	Total Time
1	2	1	11	12
2	1	1	11	12
3	2	0	12	12
4	2	1	12	13
5	2	0	12	12
6	2	1	12	13
7	2	0	12	12
8	2	1	12	13
9	2	0	12	12
10	2	1	12	13
Averages	1,9	0,6	11,8	12,4

Test 3 **Committing 100 - 110 models**

Test Nr	Retrieve	Calculate	Commit	Total Time
1	15	3	24	27
2	14	0	27	27
3	14	3	24	27
4	14	3	24	27
5	14	3	24	27
6	14	3	24	27
7	14	3	24	27
8	14	3	25	28
9	14	3	24	27
10	15	19	96	115
Averages	14,2	4,3	31,6	35,9

Test 4 **Committing 150 - 160 models**

Test Nr	Retrieve	Calculate	Commit	Total Time
1	49	3	39	42
2	49	3	39	42
3	49	3	39	42
4	49	3	39	42
5	49	3	39	42
6	50	3	39	42
7	50	3	39	42
8	50	3	39	42
9	50	3	39	42
10	51	3	39	42
Averages	49,6	3	39	42

Test 5		Committing 200 - 210 models		
Test Nr	**Retrieve**	**Calculate**	**Commit**	**Total Time**
1	62	3	52	55
2	62	3	51	54
3	62	3	51	54
4	62	3	51	54
5	62	3	51	54
6	62	3	51	54
7	62	21	123	144
8	63	3	51	54
9	63	3	51	54
10	63	3	51	54
Averages	62,3	4,8	58,3	63,1

Test 6		Committing 250 - 260 models		
Test Nr	**Retrieve**	**Calculate**	**Commit**	**Total Time**
1	89	3	61	64
2	90	3	62	65
3	90	3	61	64
4	110	20	127	147
5	91	3	61	64
6	91	3	61	64
7	92	3	61	64
8	105	14	163	177
9	93	3	61	64
10	109	22	193	215
Averages	96	7,7	91,1	98,8

Results using Augur Connection Type

Test 1		Committing 0 - 10 models		
Test Nr	**Retrieve**	**Calculate**	**Commit**	**Total Time**
1	1	3	3	6
2	1	3	3	6
3	1	3	3	6
4	2	3	3	6
5	2	3	3	6
6	2	3	3	6
7	2	3	3	6
8	2	3	3	6
9	2	3	3	6
10	2	3	3	6
Averages	1,7	3	3	6

Test 2		Committing 50 - 60 models		
Test Nr	**Retrieve**	**Calculate**	**Commit**	**Total Time**
1	7	3	12	15
2	7	3	12	15
3	7	3	12	15
4	7	3	13	16
5	7	3	12	15
6	7	3	12	15
7	7	3	12	15
8	7	3	12	15
9	7	3	12	15
10	7	3	12	15
Averages	7	3	12,1	15,1

Test 3		Committing 100 - 110 models		
Test Nr	**Retrieve**	**Calculate**	**Commit**	**Total Time**
1	16	3	25	28
2	16	3	24	27
3	16	3	24	27
4	16	3	25	28
5	16	3	25	28
6	16	3	25	28
7	16	3	25	28
8	16	3	25	28
9	17	3	26	29
10	17	10	94	104
Averages	16,2	3,7	31,8	35,5

Test 4		Committing 150 - 160 models		
Test Nr	**Retrieve**	**Calculate**	**Commit**	**Total Time**
1	24	3	36	39
2	21	3	36	39
3	21	3	36	39
4	21	3	36	39
5	21	3	42	45
6	124	3	36	39
7	21	3	36	39
8	135	3	36	39
9	21	15	36	51
10	22	3	37	40
Averages	43,1	4,2	36,7	40,9

Test 5 **Committing 200 - 210 models**

Test Nr	Retrieve	Calculate	Commit	Total Time
1	32	3	50	53
2	30	3	49	52
3	30	3	50	53
4	30	3	50	53
5	30	3	50	53
6	30	3	50	53
7	30	3	50	53
8	30	22	180	202
9	32	3	51	54
10	31	3	50	53
Averages	30,5	4,9	63	67,9

Test 6 **Committing 250 - 260 models**

Test Nr	Retrieve	Calculate	Commit	Total Time
1	41	3	62	65
2	39	3	62	65
3	39	3	62	65
4	40	3	62	65
5	40	59	76	135
6	44	3	62	65
7	40	22	126	148
8	40	3	62	65
9	40	3	62	65
10	40	90	75	165
Averages	40,3	19,2	71,1	90,3

Results using Read Ahead Connection Type

Test 1 **Committing 0 - 10 models**

Test Nr	Retrieve	Calculate	Commit	Total Time
1	1	3	2	5
2	1	3	6	9
3	1	3	3	6
4	2	3	3	6
5	2	3	3	6
6	2	3	3	6
7	2	3	3	6
8	2	3	3	6
9	2	3	3	6
10	2	3	3	6
Averages	1,7	3	3,2	6,2

Test 2 **Committing 50 - 60 models**

Test Nr	Retrieve	Calculate	Commit	Total Time
1	7	3	12	15
2	7	3	12	15
3	8	3	13	16
4	8	3	13	16
5	8	3	13	16
6	8	6	48	54
7	50	24	15	39
8	8	3	12	15
9	8	3	13	16
10	57	26	49	75
Averages	16,9	7,7	20	27,7

Test 3 **Committing 100 - 110 models**

Test Nr	Retrieve	Calculate	Commit	Total Time
1	13	3	24	27
2	13	3	24	27
3	13	3	24	27
4	13	3	24	27
5	13	3	24	27
6	14	11	118	129
7	105	3	24	27
8	14	3	24	27
9	14	23	120	143
10	148	45	119	164
Averages	36	10	52,5	62,5

Test 4 **Committing 150 - 160 models**

Test Nr	Retrieve	Calculate	Commit	Total Time
1	24	3	37	40
2	24	3	38	41
3	24	3	37	40
4	24	3	127	130
5	24	3	136	139
6	35	3	143	146
7	65	3	345	348
8	32	3	146	149
9	34	3	264	267
10	32	3	152	155
Averages	31,8	3	142,5	145,5

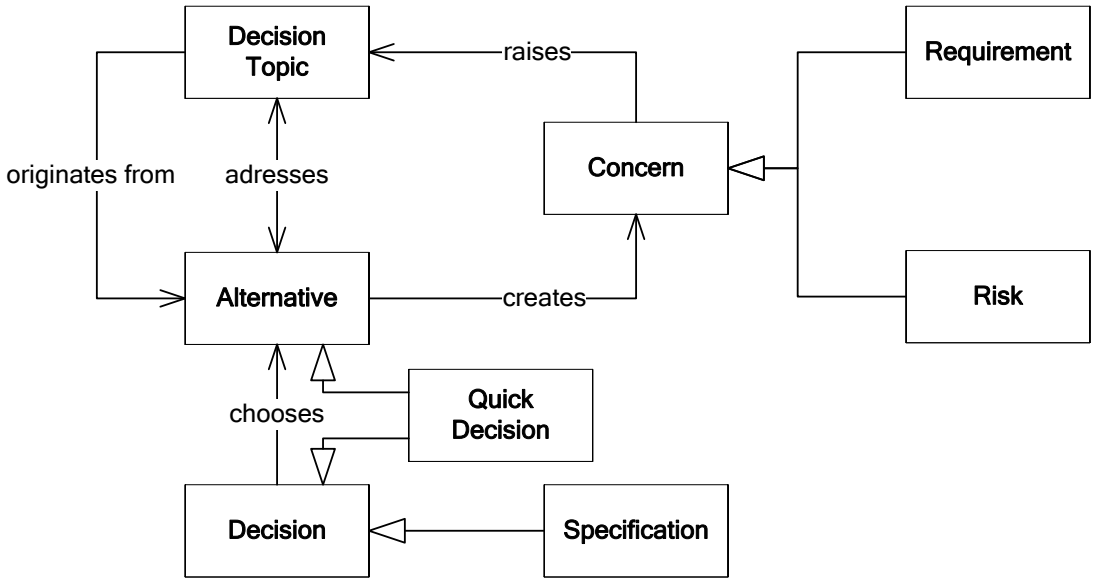
Test 5	Committing 200 - 210 models			
Test Nr	**Retrieve**	**Calculate**	**Commit**	**Total Time**
1	31	3	50	53
2	31	3	50	53
3	31	21	1387	1408
4	33	3	160	163
5	130	21	1505	1526
6	172	3	50	53
7	31	19	185	204
8	74	3	50	53
9	31	3	54	57
10	183	3	50	53
Averages	74,7	8,2	354,1	362,3

Memory usage only recorded for the default test cases

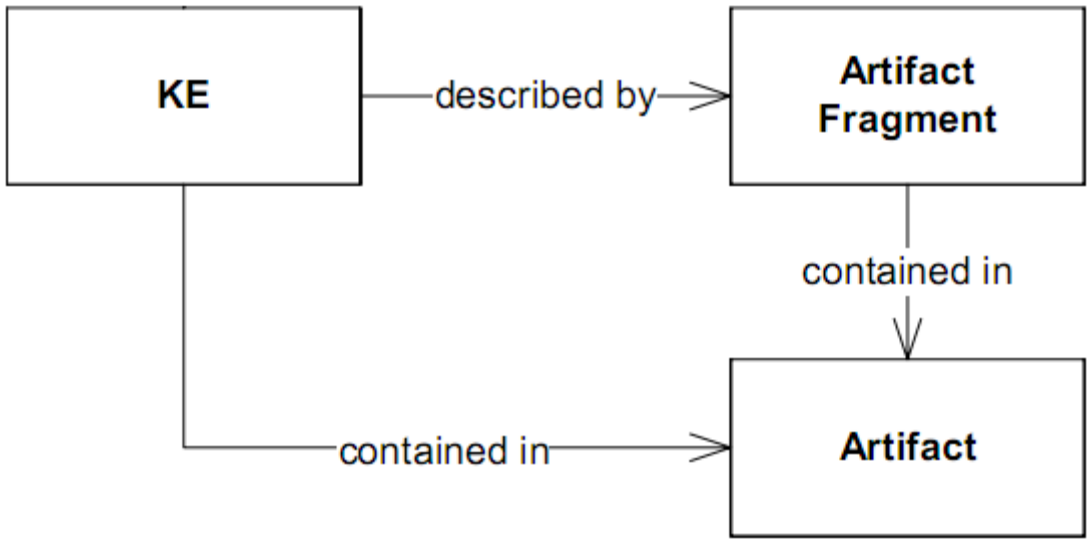
Memory Usage			
	Before	**After**	**Exported**
Test 1	46	178	19
Test 2	204	221	111
Test 3	248	265	203
Test 4	341	516	294
Test 5	439	454	386
Test 6	485	512	478

CPU Usage	
local insert	100%
remote insert	100%

APPENDIX II The LOFAR Domain Model



APPENDIX III the Meta model



APPENDIX IV Query results of the prototype repository

TESTCASE ONE

** Inserting artifacts, KE's and AF's
** AF Added: 1050000
** KE Added: 1000000
** AF Added: 1050001
** KE Added: 1000001
** AF Added: 1050002
** KE Added: 1000002

QUERY ONE

http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.archium.net/AstronGriffin#Concern
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#ID
"1000000"^^<http://www.w3.org/2001/XMLSchema#int>
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#Name
"KE_1000000"^^<http://www.w3.org/2001/XMLSchema#string>
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#status
"Checked"^^<http://www.w3.org/2001/XMLSchema#string>
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#Notes
"NOTES THAT GO WITH KE: 1000000"^^<http://www.w3.org/2001/XMLSchema#string>
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#described_by_Artifact_Fragment
http://www.archium.net/AstronGriffin/versioning#AF1050000
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#described_in_Artifact
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#raises_DecisionTopic
http://www.archium.net/AstronGriffin/versioning#AF1000001
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/2000/01/rdf-schema#Resource
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.archium.net/AstronGriffin#Knowledge_Entity

QUERY TWO

http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#KE1000001
http://www.archium.net/AstronGriffin/versioning#KE1000002

TESTCASE TWO

** Inserting artifacts, KE's and AF's
** AF Added: 1050000
** KE Added: 1000000
** AF Added: 1050001
** KE Added: 1000001
** AF Added: 1050002
** KE Added: 1000002
** AF Added: 1050003
** KE Added: 1000003

QUERY ONE

http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.archium.net/AstronGriffin#Concern
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#ID
"1000000"^^<http://www.w3.org/2001/XMLSchema#int>
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#Name
"KE_1000000"^^<http://www.w3.org/2001/XMLSchema#string>
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#status
"Checked"^^<http://www.w3.org/2001/XMLSchema#string>
http://www.archium.net/AstronGriffin/versioning#KE1000000

http://www.archium.net/AstronGriffin/versioning#described_by_Artifact_Fragment
http://www.archium.net/AstronGriffin/versioning#AF1050000
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#described_in_Artifact
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#raises_DecisionTopic
http://www.archium.net/AstronGriffin/versioning#AF1000001
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/2000/01/rdf-schema#Resource
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.archium.net/AstronGriffin#Knowledge_Entity
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#Notes
"THESE NOTES THAT GO WITH KE: 1000000 are CHANGED!"^^<http://www.w3.org/2001/XMLSchema#string>
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#Notes
"EXTRA NOTES: THESE NOTES THAT GO WITH KE: 1000000 are CHANGED!"^^<http://www.w3.org/2001/XMLSchema#string>

QUERY TWO

http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#KE1000001
http://www.archium.net/AstronGriffin/versioning#KE1000002
http://www.archium.net/AstronGriffin/versioning#KE1000003

QUERY THREE

http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000000/ADD/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#AF1050000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050000/ADD/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#AF1050002
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050002/ADD/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#AF1050001
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050001/ADD/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#KE1000001
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000001/ADD/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000/ADD/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#KE1000002
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000002/ADD/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000000/REMOVE/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#AF1050000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050000/REMOVE/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#AF1050002
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050002/REMOVE/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#AF1050001
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050001/REMOVE/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#KE1000001
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000001/REMOVE/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000/REMOVE/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#KE1000002
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000002/REMOVE/2010-01-28/09:40:46

QUERY FOUR

http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#Notes
"THESE NOTES THAT GO WITH KE: 1000000 are CHANGED!"^^<http://www.w3.org/2001/XMLSchema#string>
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#Notes
"EXTRA NOTES: THESE NOTES THAT GO WITH KE: 1000000 are CHANGED!"^^<http://www.w3.org/2001/XMLSchema#string>

TESTCASE THREE

** Inserting artifacts, KE's and AF's
** AF Added: 1050000
** KE Added: 1000000
** AF Added: 1050001
** KE Added: 1000001
** AF Added: 1050002
** KE Added: 1000002

QUERY ONE

http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.archium.net/AstronGriffin#Concern
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#ID
"1000000"^^<http://www.w3.org/2001/XMLSchema#int>
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#Name
"KE_1000000"^^<http://www.w3.org/2001/XMLSchema#string>
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#status
"Checked"^^<http://www.w3.org/2001/XMLSchema#string>
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#described_by_Artifact_Fragment
http://www.archium.net/AstronGriffin/versioning#AF1050000
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#described_in_Artifact
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#raises_DecisionTopic
http://www.archium.net/AstronGriffin/versioning#AF1000001
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/2000/01/rdf-schema#Resource
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.archium.net/AstronGriffin#Knowledge_Entity
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#Notes
"NOTES THAT GO WITH KE: 1000000"^^<http://www.w3.org/2001/XMLSchema#string>

QUERY TWO

http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#KE1000001
http://www.archium.net/AstronGriffin/versioning#KE1000002

QUERY THREE

http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000000/ADD/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#AF1050000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050000/ADD/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#AF1050002
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050002/ADD/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#AF1050001
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050001/ADD/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#KE1000001
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000001/ADD/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000/ADD/2010-01-28/09:40:46

http://www.archium.net/AstronGriffin/versioning#KE1000002
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000002/ADD/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000000/REMOVE/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#AF1050000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050000/REMOVE/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#AF1050002
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050002/REMOVE/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#AF1050001
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050001/REMOVE/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#KE1000001
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000001/REMOVE/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000/REMOVE/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#KE1000002
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000002/REMOVE/2010-01-28/09:40:46
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000000/ADD/2010-01-28/09:40:49
http://www.archium.net/AstronGriffin/versioning#AF1050000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050000/ADD/2010-01-28/09:40:49
http://www.archium.net/AstronGriffin/versioning#AF1050002
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050002/ADD/2010-01-28/09:40:49
http://www.archium.net/AstronGriffin/versioning#AF1050001
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050001/ADD/2010-01-28/09:40:49
http://www.archium.net/AstronGriffin/versioning#KE1000001
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000001/ADD/2010-01-28/09:40:49
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000/ADD/2010-01-28/09:40:49
http://www.archium.net/AstronGriffin/versioning#KE1000002
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000002/ADD/2010-01-28/09:40:49
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000000/REMOVE/2010-01-28/09:40:49
http://www.archium.net/AstronGriffin/versioning#AF1050000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050000/REMOVE/2010-01-28/09:40:49
http://www.archium.net/AstronGriffin/versioning#AF1050002
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050002/REMOVE/2010-01-28/09:40:49
http://www.archium.net/AstronGriffin/versioning#AF1050001
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#AF1050001/REMOVE/2010-01-28/09:40:49
http://www.archium.net/AstronGriffin/versioning#KE1000001
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000001/REMOVE/2010-01-28/09:40:49
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#ARTIFACT_1000000/REMOVE/2010-01-28/09:40:49
http://www.archium.net/AstronGriffin/versioning#KE1000002
http://www.griffin.nl/versioning#versionedAt
http://www.archium.net/AstronGriffin/versioning#KE1000002/REMOVE/2010-01-28/09:40:49

QUERY FOUR

http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#Notes
"THESE NOTES THAT GO WITH KE: 1000000 are
CHANGED!"^^<http://www.w3.org/2001/XMLSchema#string>
http://www.archium.net/AstronGriffin/versioning#KE1000000
http://www.archium.net/AstronGriffin/versioning#Notes
"EXTRA NOTES: THESE NOTES THAT GO WITH KE: 1000000 are
CHANGED!"^^<http://www.w3.org/2001/XMLSchema#string>