Investigating the Use of Graph Databases for Large Model Repositories

Master's Thesis

Benjamin Kiel

June 3, 2013

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Sören Frey
Dipl.-Inform. Reiner Jung
Dipl.-Inform. André van Hoorn (Stuttgart University)
Thomas Stahl (b+m Informatik AG)

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

In the context of model-driven reverse engineering and model-driven analysis of software systems, models can become very large. Up to a certain point, these models fit into memory, but often the available memory is not sufficient. Model repositories like the *Connected Data Objects* model repository (CDO) in the context of the *Eclipse Modeling Framework* (EMF) provide support for storing models and allow to work with multiple concurrent users on these models. In general, models are represented as object graphs. This leads to the assumption that graph databases could be applicable for storing models. This thesis investigates the usage of graph databases in the context of model-driven engineering. It provides a survey of available model repository technologies in the context of EMF. The survey is based on requirements for the MAMBA (Measurement Architecture for Model-Based Analysis) framework for model-based analysis. We introduce our implemented graph database backend for CDO. Furthermore, this thesis proposes a benchmark that can be used to examine the performance for CDO stores with respect to the required times for storing, loading, and querying models. Finally, we evaluate available CDO stores as well as our implementation with the help of this benchmark. We show that a graph database backend does not provide any advantage compared with a relational database backend in the context of CDO with respect to execution times.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Goals 3
1.3 Summary of Results 4
1.4 Document Structure 4

2 Foundations and Technologies **7**
2.1 Model-Driven Engineering (MDE) 7
2.2 Eclipse Modeling Project (EMP) 8
2.3 Model Repositories 9
2.4 NoSQL Databases 10
2.5 Neo4j Graph Database 11
2.6 Data Persistence Technologies for Large-Scale Models 13

3 Evaluation of Model Repository Technologies **17**
3.1 Requirements 17
3.2 EMF's Default Persistence Mechanism 19
3.3 Investigated Model Repository Technologies 20
3.4 Summary 25

4 Connected Data Objects (CDO) **27**
4.1 Features 27
4.2 CDO Client and Server Architecture 29
4.3 CDO Model Repository Internals 31
4.4 Existing CDO Stores 35
4.5 Developing a Custom CDO Store 35

5 A CDO Store with Graph Database Backend **37**
5.1 Supported Features 37
5.2 General Design Desicions 38
5.3 General Overview of the Repository Schema 39
5.4 Persisting Meta-Models 41

5.5 Persisting CDO Revisions 41

6 Evaluation **51**
6.1 Statistical Foundations 51
6.2 Execution Environment 58
6.3 Benchmark for CDO 58

7 Conclusion **73**
7.1 Summary and Discussion 73
7.2 Future Work 74

Bibliography **75**

List of Figures

2.1 Architecture of a Model Repository 10
2.2 Example for a State of a Neo4j Database 11

4.1 General CDO Architecture 30
4.2 CDO Client Architecture 30
4.3 CDO Server Architecture 31
4.4 Consistency between CDO clients 33
4.5 Object States in the Context of CDO [Stepper 2010] 34
4.6 Overview of the high level class structure 36

5.1 Overview of the database schema 40
5.2 A meta-model stored in the database 42
5.3 Mapping between object graph and neo4j store representation 43
5.4 Three model elements with different versions 43
5.5 Database state before changes 45
5.6 Database state after write revision 47
5.7 Database state after commit 49

6.1 Structure of a Box-and-Whisker-plot 52
6.2 How to choose the appropriate statistical test method? 54
6.3 Used Meta-Model for Evaluation 59
6.4 Box-and-Whisker plots for the "model import" benchmark 61
6.5 "Model import" scenario results for model size XS 63
6.6 "Model import" scenario results for model size S 64
6.7 "Model import" scenario results for model size M 65
6.8 "Model import" scenario results for model size L 65
6.9 "Model import" scenario results for model size XL 66
6.10 "Model import" scenario results for model size XXL 67
6.11 Box-and-Whisker plots for the "model export" benchmark 69
6.12 Box-and-Whisker plots for the "small query" benchmark 69
6.13 Box-and-Whisker plots for the "large query" scenario 71

List of Tables

2.1 Types of NoSQL database systems 11
2.2 Model insertion results based on [Barmpis and Kolovos 2012] 14
2.3 Query results based on [Barmpis and Kolovos 2012] 15

3.1 Summary for the investigated model repository technologies 26

4.1 Supported features of the different CDO stores, default values underlined 35

6.1 Models considered in the benchmark 60
6.2 Test statistics for the "model import" scenario 62
6.3 CDO stores comparison for model size XS 63
6.4 CDO stores comparison for model size S 63
6.5 CDO stores comparison for model size M 64
6.6 CDO stores comparison for model size L 65
6.7 CDO stores comparison for model size XL 66
6.8 CDO stores comparison for model size XXL 66
6.9 "Model export" performance comparison 68
6.10 "Small query" performance comparison 70
6.11 "Large query" performance comparison 72

Listings

2.1 Neo4j Traversal Framework example 12
2.2 Neo4j Core API example 13
2.3 Cypher example 13
3.1 XMI: Persist a model 20
3.2 XMI: Retrieve a model 20
3.3 MongoEMF: Persist a model 21
3.4 MongoEMF: Retrieve a model 21
3.5 EMF Triple: Persist a model 22
3.6 EMF Triple: Retrieve a model 22
3.7 Morsa: Persist a model 23
3.8 Morsa: Retrieve a model 23
3.9 CDO: Persist a model 24
3.10 CDO: Retrieve a model 25
5.1 Write revisions 46
5.2 Revision commit 48
5.3 Revision rollback 48

Chapter 1

Introduction

1.1 Motivation

In the past, models were mostly used for high-level design and to communicate with other developers or, more importantly, with other stakeholders like managers [Mohagheghi and Aagedal 2007]. With the introduction of Model-Driven Engineering (MDE), models became primary software artifacts [Mohagheghi 2008].

In the last decade, MDE matured and increasing industry purposes were discovered because of the advantages in productivity, quality, and reuse [Espinazo-Pagán et al. 2011]. Especially in the area of reverse engineering and dynamic analysis of software systems, model sizes increased rapidly. Up to a certain point, these models fit into memory, but often the available memory is not sufficient.

For example, the Measurement Architecture for Model-Based Analysis (MAMBA) framework [Frey et al. 2011] is based on the *Structured Metrics Meta-Model v1.0* (SMM), defined by the Object Management Group (OMG).[1] SMM is used for representing measures and measurements. A common use case for MAMBA is the transformation of raw monitoring data into SMM measurements in order to apply measures such as *average response time per method* on the measurements. Software systems can be monitored at different levels of granularity, e. g., component level or method level. The finer the granularity is, the more monitoring records will be created. Providing larger hardware (more memory) cannot face the problem since the amount of monitored data can always be increased.

Another example for large models is related to the *Knowledge Discovery Meta-Model v1.3* (KDM), also defined by the OMG. KDM was defined in order to represent complete software systems out of different view points. Hence, a common use case

[1] `http://www.omg.org` (last visit: June 3, 2013)

is the transformation from source code into a KDM model. A popular transformation tool is MoDisco,[2] that is specialized on the transformation of Java source code to KDM models. MoDisco is limited by the resulting model size. For example the enterprise resource planning application, Apache OfBiz,[3] which has approximately 1.3 million lines of code, cannot be transformed on a computer with 6 GB RAM. Lübbe [2012] showed that reducing the detail level (e. g., removing all statements) will enable successful transformations.

There are different options to solve the problem of insufficient memory. The monolithic meta-models can be split into several smaller meta-models. So, the models can be stored in several files and only relevant files must be loaded. However, the problem of finding good separations can occur [Espinazo-Pagán et al. 2011], e. g., a model can be tightly interconnected. Another option is the use of model repositories. A model repository can be considered to be a database for models. In the context of the Eclipse Modeling Framework (EMF) [Steinberg et al. 2009] the most popular implementation is the Connected Data Objects (CDO) model repository.[4]

Another advantage of model repositories is the possibility to work collaboratively on models. It allows that models are kept consistent despite concurrent access to the models. Hence, the model repository clients can focus on the business logic (separation of concerns).

However, our pre-tests showed, that CDO does not work properly for large models (more than 200,000 objects). The tests committed chunks of 2,000 objects per transaction. The required time increased exponentially, i. e., the objects 6,001 to 8,000 could be committed faster than the objects 106,001 to 108,000. Obviously, the used data stores in combination with CDO do not scale for large models.

There is a justified assumption, that graph databases can solve this problem, since models are represented as object graphs. It seems to be very intuitve to map these graphs directly to a graph database. Furthermore, graph databases facilitate fast access to connected data [Robinson et al. 2013]. World's leading companies

[2] `http://www.eclipse.org/MoDisco` (last visit: June 3, 2013)
[3] `http://ofbiz.apache.org/` (last visit: June 3, 2013)
[4] `http://www.eclipse.org/cdo` (last visit: June 3, 2013)

like Facebook [5] or Twitter [6] process and query mass of data and use their own proprietary graph databases to arrange it.

1.2 Goals

The core question of this thesis is whether graph databases can bring any advantages associated with model repositories. Models are represented as object graphs in memory. So, it seems that the mapping from models to graph databases is intuitive. We want to investigate, if graph databases bring any advantages with respect to scalability and performance. In order to answer this question, we defined three goals G1-G3.

G1: Classification of Existing Model Repository Technologies

The first goal is to classify existing model repository technologies. At the beginning, requirements for the work with a model repository will be defined. Afterwards, the different model repository technologies will be investigated regarding these requirements. The result is an overview of existing model repositories in the context of EMF that allows to reason the selection of CDO.

G2: Development of CDO Store with Graph Database Backend

The core of the thesis is the implementation of a CDO store which uses a graph database for storing the data. The thesis provides an overview of the necessary steps in order to implement a custom CDO store. Furthermore, we provide an implementation based on Neo4j.[7]

G3: Benchmark for CDO Stores

The intention of this goal is to provide a benchmark that can be used to measure and compare the performance of different CDO stores. Therefore, different use cases will be investigated, like storing and loading complete models into respectively from the database. Another use case is considering the performance for different queries that are executed on models stored in CDO.

---

[5] `http://www.facebook.com` (last visit: June 3, 2013

[6] `http://www.twitter.com` (last visit: June 3, 2013

[7] `http://www.neo4j.org` (last visit: June 3, 2013)

1.3 Summary of Results

Based on these goals, a classification of existing model repository technologies in the context of CDO has been created. A set of requirements has been defined and it has been examined which model repository technology can fullfill these requirements.

A CDO store based on the graph database Neo4j was implemented. This store is named **Neo4jStore**. A discussion of problems that arises due to the use of Neo4j as well as the use of CDO followed.

Furthermore, the thesis provides a benchmark for CDO stores. It covers different use cases like importing and exporting models from respectively to the default persistence mechanism or querying models in the repository.

The benchmark has been applied to existing CDO stores as well as to the Neo4jStore. The results do not argue for the use of Neo4j in combination with CDO. As mentioned in the previous section, models are represented as object graphs that allow to define a direct mapping to a graph database. This thesis will show, that this assumption is not helpful when using CDO since CDO decouples the object graph and encapsulates model objects in a CDO-internal representation.

1.4 Document Structure

This thesis is structured as follows.

- ▻ Chapter 2 introduces foundations like model-driven engineering and graph databases.
- ▻ Chapter 3 deals with the requirements on model repositories. Furthermore, this chapter investigates several model repository technologies and classifies them based on the requirements. Finally, an explanation is given why the use of graph databases for large model repositories will be investigated based on CDO.
- ▻ Chapter 4 presents a detailed view on the CDO model repository. It starts with an overview of the architecture, followed by an explanation of how models are managed in the model repository. The last part of this chapter deals with CDO stores which provide the persistence layer for the model repository.

- ▻ Chapter 5 deals with the concrete Neo4j-based CDO store developed in this thesis. A description of design decisions and discussion of alternative implementation possibilities will be presented.
- ▻ Chapter 6 deals with the benchmarks that were developed and applied to the different CDO stores.
- ▻ Chapter 7 summarizes and discusses our findings and gives an overview of future work and possible improvements.

Chapter 2

Foundations and Technologies

This chapter outlines important topics for the thesis. In Section 2.1, an overview of MDE practices and their advantages compared to traditional software development are given. After a short introduction of the Eclipse Modeling Project in Section 2.2, a definition of "Model Repository" in Section 2.3 is given. Section 2.4 deals with NoSQL database management systems. Section 2.5 gives an overview of the graph database Neo4j. Finally, Section 2.6 summarizes the results from the study of persistence technologies for EMF-based models.

2.1 Model-Driven Engineering (MDE)

In this section, a summary of the different practices in MDE and the resulting advantages are given. It is based upon the results by Mohagheghi [2008]. The author identified four practices that are associated with MDE:

1. **Models everywhere**
   In MDE, models are first class entities in nearly all stages of the development process (e. g., business models or requirements). Staron [2006] proposed the model-centric approach as an ambitious goal for model-driven development respectively model-driven engineering.

2. **Multiple abstraction levels and seperation of concerns**
   Different models are created in order to reduce complexity. Each model refers to a separate view point, hence there are not not many information within one model.

3. **Metamodels and metamodeling**
   Metamodeling provides the possibility to define relations between different models. This is a basis for the next practice.

4. **Generation of artifacts from models**
   This is a key concept in order to reduce manual repetitive work. Generation involves two types of transformations: M2M (Model-to-Model) and M2T (Model-to-Text). During the generation process, models can be enriched by external information (e. g., platform-dependent information).

These practices cause several improvements compared to traditional software development. They facilitate the communication between developers themselves but also between domain and IT experts. One reason is a higher abstraction, that makes the design available for more people [MacDonald et al. 2005]. Another important improvement is the increasing software quality. Detailed models can be used for model-driven analysis and model-driven testing. Moreover, design patterns can be integrated in the generation process. A more detailed discussion of MDE can be found in [Mohagheghi 2008].

2.2 Eclipse Modeling Project (EMP)

Eclipse[1] is an open source software project that is managed by the Eclipse Foundation. Its purpose is to provide an *integrated development environment* (IDE) and a programming platform. It is organized into different top-level projects: The Modeling Project, the Tools Project, the Technology Project, and other projects. For this thesis only the Modeling Project is important. Its core ist the Eclipse Modeling Framework (EMF) that provides the basic framework for modeling [Steinberg et al. 2009].

The Ecore meta-model is used to represent models in EMF. An overview of the elements of Ecore can be found in [Steinberg et al. 2009]. Ecore can be considered as a meta-meta-model, i. e., instances of Ecore are meta-models themselves.EMF enables the definition and instantiation of these meta-models. It provides a graphical editor (like a class diagram editor) that supports the definition of meta-models.

Furthermore, EMF facilitates the generation of Java source code that can be used in the Eclipse Rich Client Platform (RCP) or in standalone applications.

Since the context of this thesis is Ecore-based modeling languages the brief form "modeling language" is used if the context is clear.

[1] `http://www.eclipse.org` (last visit: June 3, 2013)

2.3 Model Repositories

Model repositories can be seen as database management systems for models. Features that can be identified for model repositories are persistence, versioning, querying large models, partial loading, and collaboration (consistence and distributed working).

Persistence differs from file serialization to DBMS. **Versioning** means that changes are tracked and can be assigned to users. Furthermore, old revisions can be restored. Model repositories often allow to **query large models** in a faster way. The queries will be mapped on DBMS-specific query languages that cause faster executions. **Partial loading** denotes the ability to load only parts of the model that are required by the clients. Hence, there is less memory consumption. At last, **collaboration** means a parallel work on the same model.

A distinction between *offline* and *online* collaboration can be made. Offline collaboration can be regarded as the classical process for source code versioning. A model can be shared and checked out afterwards. Then changes can be made and committed. These changes can be updated and if conflicts arise the changes will be merged through the user. On the other hand, online collaboration means that changes on the model will be immediately promoted to the server. So it is similar transactional behavior as known from DMBS. Before a resource is changed, the resource will be locked first. Then the changes will be commited and the lock will be released. In the context of EMF, there are implementations for both. An existing example for the former is the *EmfStore*, while *CDO* is an implementation for the latter.

Figure 2.1 shows common components of model repositories. The **persistence** component is responsible for the mapping between model entities and the existing data store. The **cache** enables faster access to model entities. The **transaction** component controls concurrent access to models. The **versioning** component observes and tracks changes on models. The **queries** component translates queries to a native query language (regarding the data store) that can execute the queries in a faster way. At last, the **views** component deals with views on models (similar to views in relational databases).

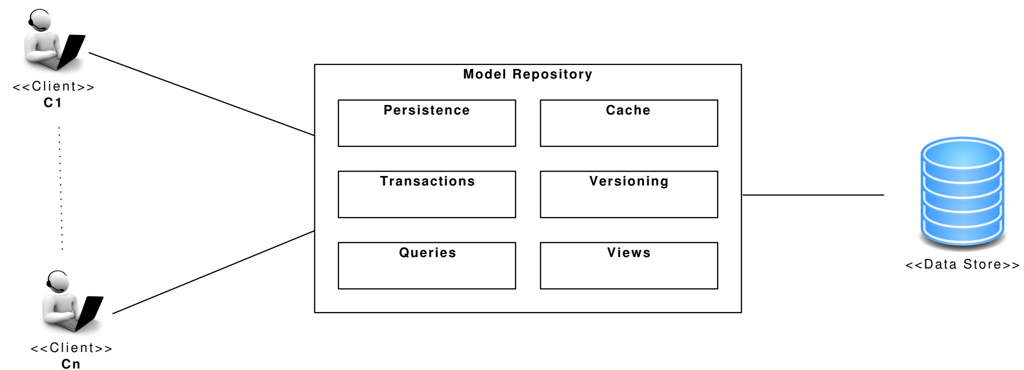

Figure 2.1. Architecture of a Model Repository

2.4 NoSQL Databases

The term "NoSQL" was introduced by Carlo Strozzi in 1998 [Edlich et al. 2010]. However, the underlying data model was still relational. NoSQL databases became popular with the beginning of the Web 2.0. It was necessary to work with mass of data (terabytes or even petabytes) by then. The relational databases were not able to cope with this new development.

The following list shows some properties that can be found by most of the NoSQL database systems [Edlich et al. 2010]:

- Distributed and horizontal scalability by default
- Easy data replication
- Data model is not relational
- No schema or weaker schema restrictions
- Simple API
- Other consistency model: Eventual consistency instead ACID

Table 2.1 shows four different types of NoSQL database systems. The most obvious difference is the underlying data model [Edlich et al. 2010].

Table 2.1. Types of NoSQL database systems

Type	Example
Key/Value systems	Amazon's Dynamo (http://aws.amazon.com/dynamodb)
Tabular stores	Hbase (http://hbase.apache.org/)
Document stores	MongoDB (http://www.mongodb.org)
Graph databases	Neo4j (http://www.neo4j.org)

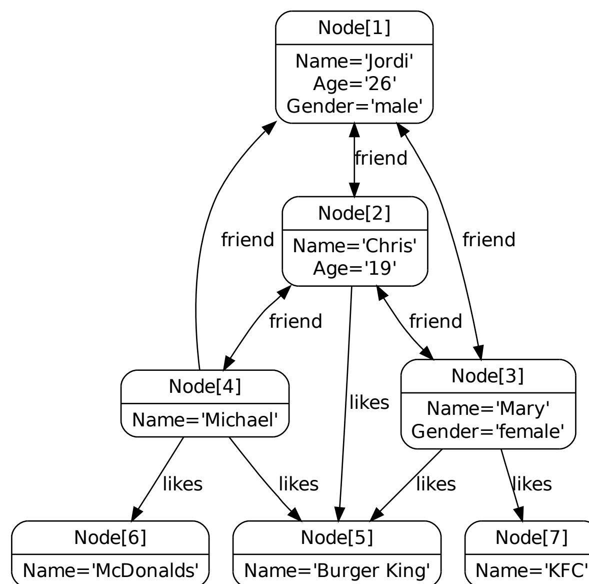

Figure 2.2. Example for a State of a Neo4j Database

2.5 Neo4j Graph Database

This master's thesis focusses on the Neo4j graph database. Its underlying data structure conforms to the property graph model [Robinson et al. 2013]. A property graph contains of nodes and named relationships. Nodes and relationships can contain properties which are key-value pairs. Relationships exist between two nodes and have a direction. Figure 2.2 shows an example graph of a Neo4j DB, which comprises different attributes. Obviously, some nodes represent persons

while other nodes represent fast food restaurants. Furthermore, there are two different kinds of relationships. Firstly, persons can be friends among themselves. Secondly, persons are able to like fast food restaurants.

The main advantage of graph databases compared to relational databases and other NoSQL databases is the good performance when working with tightly connected data [Robinson et al. 2013]. Relational databases split the data into multiple database tables. Even simple queries like "friends-of-friends" relationship require expensive "join" operations. Most NoSQL databases (regardless of whether key-value, document-, or column-oriented stores) use disconnected data structures (values, documents, columns). The relationships must be managed on application level by using aggregation identifiers. In contrast to the relational and the considered NoSQL which use implicit connections, graph databases uses the connections explicitly. These connections forms paths on the graph that must be followed for querying the graph database. The graph database is independent from the number of stored elements because only the parts of the data that are necessary will be considered.

Neo4j provides three ways to query the data: Cypher, Traversal Framework, or Core API [Robinson et al. 2013]. The Traversal Framework is a declarative Java API. It is similar to the Criteria API known from the Java Persistence API (JPA) [Goncalves 2010]. The programer specifies which relationships are allowed to be traversed, the type of the search (depth-first, breadt-first). An example is given in Listing 2.1. The Core API is an imperative Java API which works with the primitives nodes, relationships, and properties. The queries are lazily evaluated, i. e., relationships will not be evaluated until the relationship is accessed. Listing 2.2 shows a query with the Core API. The last way is the Cypher query language which is equals to SQL. An example is provided in Listing 2.3. Robinson et al. [2013] states that the Core API is the fastest possibility to query the graph database. Followed by the Traversal Framework. The slowest possibility in Cypher.

Listing 2.1. Neo4j Traversal Framework example

```
1 Traversal.description()
2         .relationships(FRIEND, Direction.OUTCOMING)
3         .breadthFirst();
```

Listing 2.2. Neo4j Core API example

```
1 Node marc = findByName("Marc");
2 Iterable<Relationship> relationships = marc.getRelationships(
3                                 Direction.OUTGOING, FRIEND);
4 for(Relationship rel : relationships) {
5     Node friend = rel.getEndNode();
6     System.out.println(friend.getProperty("name"));
7 }
```

Listing 2.3. Cypher example

```
1 START marc=node:names(name='Marc')
2 MATCH marc-[:FRIEND]->(friend)
3 RETURN friend.name
```

2.6 Data Persistence Technologies for Large-Scale Models

Barmpis and Kolovos [2012] investigated persistence technology is the most appropriate for EMF-based models throughout a comparison of five technologies.

- *XMI Serialization*, this is the default persistence mechanism provided by EMF. The models will be written to XMI files.
- *Teneo/Hibernate* is an object-relational mapper that stores models in nearly arbitrary relational databases by using the Hibernate framework.
- *Morsa* [Espinazo-Pagán et al. 2011] stores models in the document store MongoDB.
- A *Neo4j* prototype was contributed by the authors
- A *OrientDB* prototype was also contributed by the authors. It stores th models into an hybrid database (document and graph)

The study focussed on memory consumption and response time for model insertion and model querying. They took five different models which vary in size from 8.75 MB to 645.53 MB.

2.6.1 Storing Models

Table 2.2 shows the results for model insertion. For the "insertion" case, the authors did not consider the XMI serialization, hence there are no results for this issue. Moreover, there are no results for the Morsa implementation available as well. The

Table 2.2. Model insertion results based on [Barmpis and Kolovos 2012]

		Persistence Mechanism (Time in s)				
Model	**Size (MB)**	**XMI**	**Teneo/Hibernate**	**Morsa**	**Neo4J**	**OrientDB**
Set0	8.75	n/a	58.67	-	12.43	19.58
Set1	26.59	n/a	218.20	-	32.52	57.10
Set2	270.12	n/a	-	-	499.09	589.80
Set3	597.67	n/a	-	-	2210.17	2245.45
Set4	645.53	n/a	-	-	2432.16	2396.88

Teneo implementation did not pass all tests. It was only possible to store rather small models. Unfortunately there is a large gap between the second and the third model. Hence, the size limit can not be determined exactly. Both NoSQL implementations could handle even large models. Except the last model, Neo4j required less time than OrientDB. It would be interesting to investigate whether OrientDB would be the better solution for models larger than 645.53 MB.

2.6.2 Querying Models

Table 2.3 shows the results for the queries on the same models as before. Teneo could be considered for the first two models only since other models were not created. Notably, Neo4j offers the best performance regarding response time and memory consumption. The larger the models gets, the better is the performance of Neo4j. Hence, the authors conclude that graph databases in general and especially the Neo4J implementation are adequate to persist even large models. Since model storage is not the only aspect of interest for this thesis but also other features of model repositories, this study can only be an indication for our thesis. Nevertheless, the results are promising.

Table 2.3. Query results based on [Barmpis and Kolovos 2012]

			Persistence Mechanism				
Model	**Size (MB)**	**Metric**	**XMI**	**Teneo/Hibernate**	**Morsa**	**Neo4J**	**OrientDB**
		Time	1.20	4.53	0.71	0.11	0.43
Set0	8.75	Mem(Max)	42	248	-	15	10
		Mem(Avg)	19	117	5	11	10
		Time	2.28	7.34	0.99	0.62	1.18
Set1	26.59	Mem(Max)	111	323	-	18	27
		Mem(Avg)	48	176	8	13	17
		Time	16.51	-	9.72	3.10	9.83
Set2	270.12	Mem(Max)	813	-	-	401	742
		Mem(Avg)	432	-	168	195	255
		Time	84.91	-	26.76	6.71	24.41
Set3	597.67	Mem(Max)	1750	-	-	960	2229
		Mem(Avg)	844	-	205	620	881
		Time	145.67	-	29.34	7.16	29.65
Set4	645.53	Mem(Max)	1850	-	-	1070	2463
		Mem(Avg)	939	-	254	866	1314

Chapter 3

Evaluation of Model Repository Technologies

This chapter gives an overview of the existing model repository technologies in the context of EMF. Section 3.1 deals with requirements that we impose on model repositories. Section 3.2 shows the EMF default persistence mechanism based on XML serialization and considers its drawbacks. Section 3.3 considers different model repositories. Each model repository will be examined with regard to the imposed requirements. Furthermore, it will be shown how to persist and load models from the respective model repository. Section 3.4 summarizes the findings and compare the model repositories. Moreover, it will be explained why we decided to use CDO as base technology for further investigations.

3.1 Requirements

Section 2.3 gives an overview of the features of model repositories. In order to find an adequate model repository, we defined the following eight requirements that must be met.

EMF Support

As mentioned above, the context of the thesis are EMF-based modeling. Hence the model repository must be able to work with models that conform to EMF-based modeling languages. It is important that the model repository does not depend on the modeling language itself but rather on the EMF language facilities.

Standalone mode

It is required that the model repository works without the Eclipse IDE. Sometimes tools, e. g., MAMBA, are used in an environment that does not support graphical user interfaces. However it is allowed to use the Eclipse platform.

Transactions

One of the drawbacks of the standard XMI-based model storage is lack of multi-user support. Only one user can access the model at the same time. In business applications often multiple users access business models at the same time. So, we need the ability to answer multiple requests simultaneously. This means especially the concurrent read/write access.

Queries

This requirement deals with the question whether the model repository provides querying facilities. EMF conform models can be surely queried by OCL. But maybe it is also possible that a model repository supports native queries which could be quite faster.

Cache

Models can get large and changes on models often involve only small parts of the models. But these parts are often requested multiple times. If there is a cache available, the response time can be reduced immensely.

Partial Loading

This point have two advantages. The first advantage is the faster access of models similar to "Cache". Only the parts that are required will be loaded. The second advantage is the ability to work with large models i. e., models that do not fit in the memory.

Versioning

Versioning means the ability to track changes on models as we know it from source code management systems like Git or Subversion. Furthermore, it means the ability to restore old versions.

Invasivity

This requirement deals with the question how complex the usage of the respective model repository actually is. This can vary from small changes only to the client applications to invasive changes in the meta-models.

3.2 EMF's Default Persistence Mechanism

The default persistence mechanism stores the models in an XMI format in the file system. Stepper [2010] presents several drawbacks of this approach when the models become too large. EMF certainly allows to partition models, so they can be stored in several files. This causes various problems. The decision about partitioning is required at design time. Often, suitable partitions cannot be figured out at this time. Another problem is that the storage into files is not transactional safe. Assume, there is a model partioned into three files. Now, changes are made to the model, that affect all three files. If an error occurs after the first file was saved and the procedure is cancelled, the model is inconsistent. In order to avoid this, a manager, that provides transactional access, is needed. This could be a source code management system that allows to store different versions of the files in order to move back to earlier versions. But this causes another problem. If conflicts arise, they must be resolved in text form. Often, this can be inconvenient and impossible also there are tools like EMFCompare. So, notifications would be reasonable when other clients did changes on the model. This can only be done on the file level, i. e., it can be identified which file was changed but not which actual object was concerned. Another issue is that it is still impossible to load single objects. The granularity of demand loading is on the resource level. So, if you have to load an object, you do not need to load all files but the complete file that contains the object. In addition to that, it is not possible to do a garbage collection of objects that are not needed anymore. So, Stepper [2010] concludes, that the default persistence mechanism can cause many problems and does not scale very well for large models.

Now, we will consider how to persist and load models in order to compare this procedure with the investigated model repositories. Lets start with the persistence case. Listing 3.1 shows the required steps in oder to write a EMF-based model to an XMI file. A more detailed view can be found in Steinberg et al. [2009].

First, a `ResourceSet` will be created (line 1). Then, it will be told that it has to use the XMI format to save the model, this is only needed for stand-alone (lines 2–5). Then, the model will be created (lines 6–8). Due to simplicity, the models in this chapter will consist of only one object. Then, a `Resource` with the location of the file will be created (line 9–10). Afterwards, the model will be added to the `Resource` (line 11) and, finally, the `Resource` will be stored in the file system (line 12).

Listing 3.1. XMI: Persist a model

```
1  ResourceSet resourceSet = new ResourceSetImpl();
2  resourceSet.getResourceFactoryRegistry()
3              .getExtensionToFactoryMap()
4              .put(Resource.Factory.Registry.DEFAULT_EXTENSION,
5                      new XMIResourceFactoryImpl());

6  User user = ModelFactory.eINSTANCE.createUser();
7  user.setName("Test User");
8  user.setEmail("test@example.org");

9  Resource resource = resourceSet.createResource(
10                             URI.createURI("/tmp/file.xmi"));

11 resource.getContents().add(user);
12 resource.save(Collections.EMPTY_MAP);
```

Listing 3.2. XMI: Retrieve a model

```
1  ResourceSet resourceSet = new ResourceSetImpl();
2  resourceSet.getResourceFactoryRegistry()
3              .getExtensionToFactoryMap()
4              .put(Resource.Factory.Registry.DEFAULT_EXTENSION,
5                      new XMIResourceFactoryImpl());

6  resourceSet.getPackageRegistry().put(ModelPackage.eNS_URI,
7                  ModelPackage.eINSTANCE);

8  Resource resource = resourceSet.getResource(
9                      URI.createURI("/tmp/file.xmi"), true);

10 User user = (User) resource.getContents().get(0);
```

Listing 3.2 shows how to load models from XMI files into the memory. Like for persisting, first a `ResourceSet` must be created and it must be informed, that it must handle XMI files (lines 1–5). Then the package must be registered. If this was not done, EMF cannot know how to convert the XMI representation into real objects (lines 6–7). Then, the file will be actually loaded (lines 8–9). Finally, the model can be retrieved and the first object in the list will be taken.

3.3 Investigated Model Repository Technologies

The following list shows the model repository technologies which will be considered in this section:

▻ MongoEMF (https://github.com/BryanHunt/mongo-emf)

▻ emftriple (http://code.google.com/a/eclipselabs.org/p/emftriple)

▻ Morsa (http://modelum.es/trac/morsa)

▷ EMFStore (http://eclipse.org/emfstore)

▷ CDO (http://www.eclipse.org/cdo)

3.3.1 MongoEMF

MongoEMF allows to persist EMF-based models in the NoSQL database MongoDB. Its advantage lies in the small amount of additional code that is necessary to use it. It is possible to use this model repository without the Eclipse IDE. The usage is very similar to the XMI facility. Hence, the features Transactions, Queries, Cache, and Versioning are not supported. Listing 3.3 and Listing 3.4 show the usage of MongoEMF.

Listing 3.3. MongoEMF: Persist a model

```
1  ResourceSet resourceSet = resourceSetFactory.createResourceSet();

2  User user = ModelFactory.eINSTANCE.createUser();
3  user.setName("Test User");
4  user.setEmail("test@example.org");

5  Resource resource = resourceSet.createResource(
6                  URI.createURI("mongodb://localhost/db/users/"));
7  resource.getContents().add(user);

8  try {
9    user.eResource().save(null);
10 } catch(IOException e) {
11   e.printStackTrace();
12 }
```

Listing 3.4. MongoEMF: Retrieve a model

```
1  ResourceSet resourceSet = resourceSetFactory.createResourceSet();
2  Resource resource = resourceSet.getResource(URI.createURI(
3          "mongodb://localhost/app/users/4d6dc268b03b0db29961472c"), true);

4  User user = (User) resource.getContents().get(0);
```

3.3.2 emftriple

This model repository can be used in different ways. The first way is very simlar to the previous model repository. But the second way uses an `EntityManager` which supports transactions an queries. emftriple can also be used in a standalone context. It provides more feature than MongoEMF, but there are still some features missing like caching or versioning.

Listing 3.5 and Listing 3.6 show the usage of emftriple without the entity manager.

Listing 3.5. EMF Triple: Persist a model

```
1 ResourceSet resourceSet = resourceSetFactory.createResourceSet();

2 User user = ModelFactory.eINSTANCE.createUser();
3 user.setName("Test User");
4 user.setEmail("test@example.org");

5 Resource resource = resourceSet.createResource(
6         URI.createURI("emftriple://data?graph=http://graph"));


7 resource.getContents().add(person);
8 resource.save(null);
```

Listing 3.6. EMF Triple: Retrieve a model

```
1 Resource resource = resourceSet.createResource(
2                     URI.createURI(
3                         "emftriple://data?graph=http://graph"));
4 resource.load(null);

5 Person obj = (Person) EcoreUtil.getObjectByType(
6                     resource.getContents(),
7                     ModelPackage.eINSTANCE.getPerson());
```

3.3.3 Morsa

Morsa is also a model repository that is based on MongoDB. It can be used in standalone mode and supports transactions, queries, and cache. On the other hand, it does not support versioning and partial loading. The biggest feature is that it requires only small changes in the application client code. Compared with CDO, Morsa seems to be faster [Espinazo-Pagán et al. 2011].

Listing 3.7 and Listing 3.8 shows the usage of the Morsa model repository.

3.3.4 EMFStore

In contrast to the previous model repositories, EMFStore supports features like versioning and cache besides the features that are also supported by the previous model repositories. EMFStore is based on the commit-update-merge pardigm that is known from source code management (scm) tools like Subversion. When a user commits a change on the model, than another user that changed the same

Listing 3.7. Morsa: Persist a model

```
ResourceSet rs = new ResourceSetImpl();
rs.getResourceFactoryRegistry().getExtensionToFactoryMap()
        .put("ecore", new EcoreResourceFactoryImpl());
rs.getResourceFactoryRegistry().getProtocolToFactoryMap()
        .put("morsa", new MorsaResourceFactoryImpl(
                new MongoDBMorsaBackendFactory()));

Resource mmr = rs.getResource(
                URI.createFileURI("model.ecore"), true);
for (EObject o : mmr.getContents()) {
    registerPackages((EPackage) o);
}

User user = ModelFactory.eINSTANCE.createUser();
user.setName("Test User");
user.setEmail("test@example.org");


Resource morsaResource = rs.createResource(URI.createURI(morsaURI));
morsaResource.getContents().add(user);

try {
    Map options = new HashMap();
    // Set options ...
    morsaResource.save(options);
} catch (IOException e) {
    e.printStackTrace();
}
```

Listing 3.8. Morsa: Retrieve a model

```
Resource morsaResource = rs.createResource(URI.createURI(morsaURI));
try {
    Map options = new HashMap();
    options.put(MorsaResource.OPTION_SERVER_URI, backendURI);
    options.put(MorsaResource.OPTION_PRINT_TRACE, printTrace);
    options.put(MorsaResource.OPTION_DEMAND_LOAD, false);
options.put(MorsaResource.OPTION_READ_ONLY_MODE, true);

    morsaResource.load(options);

    User user = (User)  morsaResource.getContents().get(0);
} catch (Exception e) {
     e.printStackTrace();
}
```

element on the model must update his local working copy from the repository. If no conflicts occur, the scm tool can merge both changes automatically. Otherwise, the user has to solve the conflict manually.

A big drawback is that EMFStore only works within the Eclipse IDE. Hence, EMFStore is completely uninteresting and we abstain from giving a source code example.

3.3.5 CDO

CDO is one of the most matured model repositories in the area of EMF. It encapsulates the model repository facilities from the actual data store. This allows to use arbitrary data stores. Hence, numerous data stores are shipped with CDO. There is already support for relational databases due to the DBStore or the HibernateStore. Furthermore, there is a MEMStore which provides an in-memory solution. The MongoDBStore provides a NoSQL-database backend based on MongoDB. The Db4oStore enables CDO to persist models in an object-oriented database. Finally, there are proprietary data stores like ObjectivityStore. It integrates very well in the Eclipse IDE but also provides a standalone mode. Furthermore, it provides all facilities that can be expected. Write access must always be done within a transaction. Furthermore, it provides query possibilities e. g., query langugage for the respective data store.

Listing 3.9. CDO: Persist a model

```
1  session = createSession();
2  CDOTransaction transaction = session.openTransaction();
3  try {
4      CDOResource resource = transaction
5              .getOrCreateResource("/users");

6      User user = ModelFactory.eINSTANCE.createUser();
7      user.setName("Test User");
8      user.setEmail("test@example.org");

9      resource.getContents().add(user);
10     transaction.commit();
11 } catch (CommitException e) {
12     e.printStackTrace();
13     transaction.rollback();
14 } finally {
15     cleanup();
16 }
```

Listing 3.9 shows the usage of CDO in order to persist a model. Before the CDO server can be used, a connection as well as a session must be established (line 1). After a transaction is started (line 2), a resource will be requested. If the

resource does not exist, it will be created (line 4). After the user was created (lines 6–8) the user object will be added to the retrieved resource (line 9). Afterwards, the transaction can be committed (line 10). This can lead to a CommitException that must be caught and makes it necessary to rollback the transaction (lines 11–13). Finally, some cleanup work must be done e. g., closing the session (line15).

Listing 3.10. CDO: Retrieve a model

```
1 session = createSession();

2 CDOView view = session.openView();
3 CDOResource resource = view.getResource("/company", true);

4 User user = (User) resource.getAllContents().next();
```

Listing 3.10 shows how to retrieve a model from the CDO server. The first step is to establish a connection and a session like for the persist case (line 1). As opposed to the persistence, now a cdo view will be created (line 2). This allows only read access to the CDO server. Then, the resource will be retrieved and then the first object will be taken (lines 3–4).

3.4 Summary

This section summarizes the previous sections. Based on that, we explain why we decided to develop an extension to the CDO server.

Table 3.1 shows a summary of the las section. All investigated model repositories provide EMF support and all except EMFStore provide a standalone mode. MongoEMF and EMFStore support less than the half of the requirements, emftriple supports exactly the half of the requirements, Morsa provides three fourth, and CDO supports all except one. Invasivity is not a key requirement. Hence, we decided to use CDO for further investigations because it provides a good basis and we do not need to reinvent the wheel. The next chapter gives a detailed view on CDO.

Table 3.1. Summary for the investigated model repository technologies

Technology	EMF Support	Standalone mode	Transactions	Queries	Cache	Partial loading	Versioning	Invasivity
MongoEMF	++	++	–	–	–	–	–	++
emftriple	++	++	++	++	–	–	–	–
Morsa	++	++	++	++	++		–	++
EMFStore	++	–	++	–	–		++	–
CDO	++	++	++	++	++	++	++	–

Chapter 4

Connected Data Objects (CDO)

In the last chapter, we got a first impression of the features that are supported by CDO. This chapter presents a detailed view on CDO will be presented. It is based on the documentation of CDO, available on the Internet [*CDO Model Repository Overview*]. Section 4.1 presents a detailed view on the features of CDO. Section 4.2 deals with a general overview of the architecture. Section 4.3 deals with the transformation from EMF-based models in a representation that can be handled by CDO. Section 4.4 reveals CDO stores that are already shipped with CDO since these stores are our comparison level in the evaluation section. The Section 4.5 describes the classes in order to create an own CDO store.

4.1 Features

Now, the features of the CDO model repository will be regarded in more detail. This section is based on Stepper [*CDO Model Repository Documentation*].

4.1.1 Multi User Access

Different repository sessions allow a multi user access to the models. To ensure the secure authentication of users, model repositories can be configured in a particular way. Therefore, different authorization policies can be used.

4.1.2 Transactional Access

Transactional access to the models is possible through optimistic and/or pessimistic locking on a per object granule. Multiple savepoints are used to provide the opportunity to roll back to changes. All kinds of locks can be used in form of long lasting locks that outlast repository restarts. Modifications of models with respect to transactions in multiple repositories can be made through XA transaction notions with a two phase commit protocol.

4.1.3 Transparent Temporality

Audit views provide a transparent temporality, since they are special read only transactions that ensure a consistent model object graph. This graph has the same state than in the past. The storage of the audit data can lead to increased database sizes in time. Repositories can be configured to solve this problem.

4.1.4 Parallel Evolution

Parallel evolution of the object graph stored in a repository can be observed, i.e. the concept of branches similar to source code management systems like Subversion. There are different possibilities to compare branch points, f.e. sophisticated APIs or reconstruction of committed change sets.

4.1.5 Scalability

Scalability means the ability to store and access models of any random size. That is possible through loading single objects on demand and caching them in the application.Therefore, objects that are no longer needed have to be garbage collected automatically. There are various attendants concerning lazy loading, such as monitoring of the object graph's usage and the calculation of optimal fetch rules.

4.1.6 Thread Safety

Thread safety manages the synchronization of multiple threads of the application, so that the object graph is accessable all time. It ensures that multiple transactions can be opened and that all transactions share the same object data.

4.1.7 Collaboration

If an application agrees on being notified about remote changes to the graph, collaboration on models with CDO can be very simple. The graph transparently changes by configurating the local object, when it has changed remotely.

4.1.8 Data Integrity

Several checks can be used to ensure data integrity, f.e. referential integrity checks, containment cycle checks or custom checks.

4.1.9 Fault Tolerance

Fault tolerance on multiple levels, such as fail-over clusters or special session types like reconnecting sessions ensure that applications are able to hold a copy of the object graph. That is possible even when the repository connection is broken.

4.1.10 Offline Work

Two possible mechanisms provide the opportunity to work offline with the models: Firstly, the whole repository including all history of the branches can be cloned. The clone will be synchronized with its master constantly and it can be applied as an embedded repository for single applications or as a server for multiple clients. Secondly, it is possible to check out a single version of the graph from a special branch point into a local CDO workspace. This workspace acts like a local repository, branching or history information is not needed. Multiple concurrent transactions and remote functionality are supported by this method.

4.2 CDO Client and Server Architecture

Most applications that use CDO conform to a classical three-tier architecture as shown in Figure 4.6. Client applications can access the model repository through the network. The CDO server itself accesses a CDO store in form of different database management systems as well as web services or file systems [Kloos et al. 2012]. Several CDO stores are already shipped with the distribution as we will describe in Section 4.4.

Figure 4.2 shows the components of a CDO client. In general, all components are implmented as Open Services Gateway Initiative (OSGi[1]) bundles. OSGi is a dynamic module system for Java. However, the client does not need to use OSGi in order to work proper. At the top, there is the actual application that uses CDO. Between these layers there is a layer that combines EMF and the CDO model repository. On the one side there are components that deal with the transport of the data across the network. The transport is based on Net4j[2]. On the other side there are components for the EMF support. Both sides are connected through the CDO client component.

[1] http://www.osgi.org

[2] http://wiki.eclipse.org/Net4j

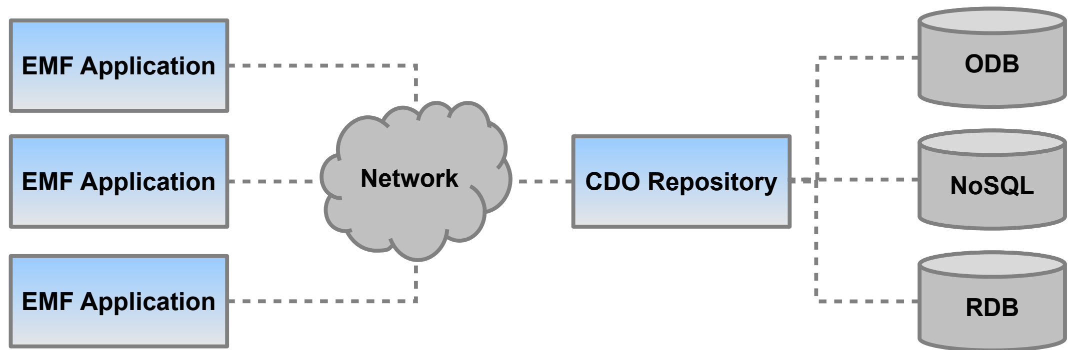

Figure 4.1. General CDO Architecture [**cdo_documentation**; Kloos et al. 2012]

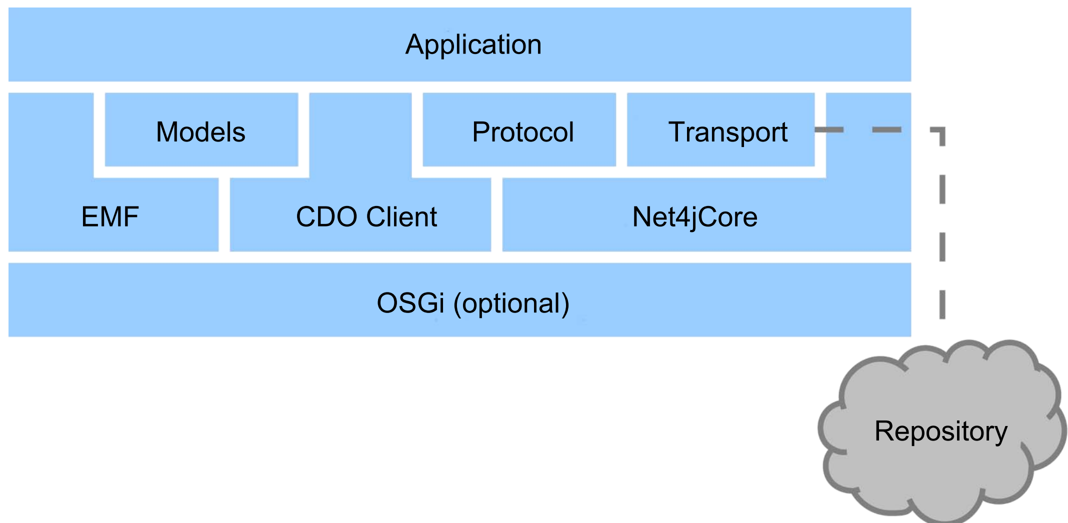

Figure 4.2. CDO Client Architecture [**cdo_documentation**; Kloos et al. 2012]

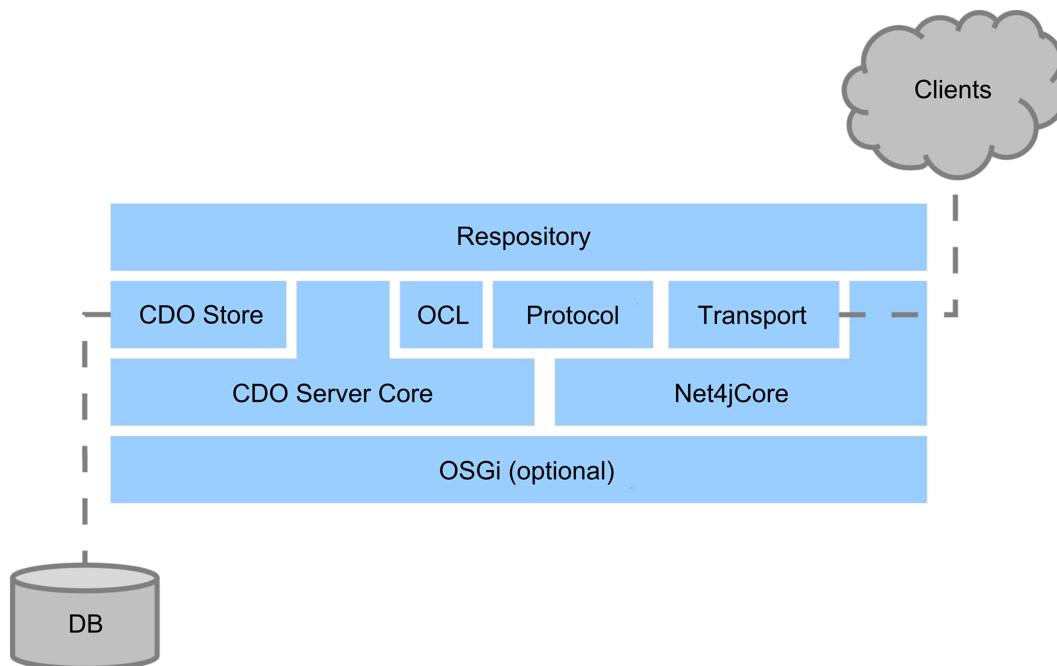

Figure 4.3. CDO Server Architecture [**cdo_documentation**; Kloos et al. 2012]

Figure 4.3 shows the architecture of the CDO server respectively the CDO model repository. The components of the model repository are also implemented as OSGi bundles. Moreover, the model repository also consists of the components that handle the transport to the clients. The CDO server core consists of the major components, e. g., cache or session manager. These components provide the services that the model repository offers. The CDO store component represents the persistence backend. Finally, it is remarkable that the CDO model repository does not have any dependency on EMF although it stores EMF-based models. That is because

4.3 CDO Model Repository Internals

After the brief architectural overview of the CDO model repository, this section deals with the general processing of model objects. The transition from model

objects to CDO revisions will be presented first, followed by the properties of revisions.

4.3.1 Converting from Revisions to Objects

The CDO server encapsulates the model object in an own representation and replaces references through CDOID objects. So, the object graph is flattened down into a list of object elements and references are represented as id objects. CDO does not work with model objects but rather with revisions. A revision consists of CDO-specific data like the related EClass, an internal ID which is generated by the model repository. It is globally unique and does never change. This ensures moving objects in the containing hierarchy without changing the ID. Furthermore, it belongs to a specific branch, has got a specific version, and an interval during that this revision was the latest. On the other side, the revision contains EMF-specific data like the ID of the inherent resource, the id of the containing object, and of course the values of the model object.

The main challenge is to suspend strong Java references between model objects. Assume a class `Company` with an association to the class `Address`. As long as the company object exists, there is a reference on the address object and hence, the address object cannot be garbage collected although it is not needed. CDO solves this problem by cutting out all generated fields, i. e., not only references to other objects, but also primitives. Therefore, CDO uses the generator pattern *Reflective Feature Delegation*. So the generator does not create instance variables to store values of the features. The feature accessors rather invoke the reflective eSet() and eGet() methods [Steinberg et al. 2009]. Reflective Delegation and RootsExtendsClass

CDO introduces a three-dimensional view on model objects. The first dimension consists of the object itself. The second dimension is formed by the time, i. e., the different existing versions of an object. The third dimension is given through parallel branches of object versions.

4.3.2 Distributed Shared Models

Now, the CDO work with revisions shall be considered. If a revision is loaded by a client, this revision is connected with the model repository all the time. This allows the model repository to notify clients about changes on certain revisions. The sequence diagram in Figure 4.4 shows what happens if multiple clients access the same object. We can see two clients *A* and *B* that connect to the model repository

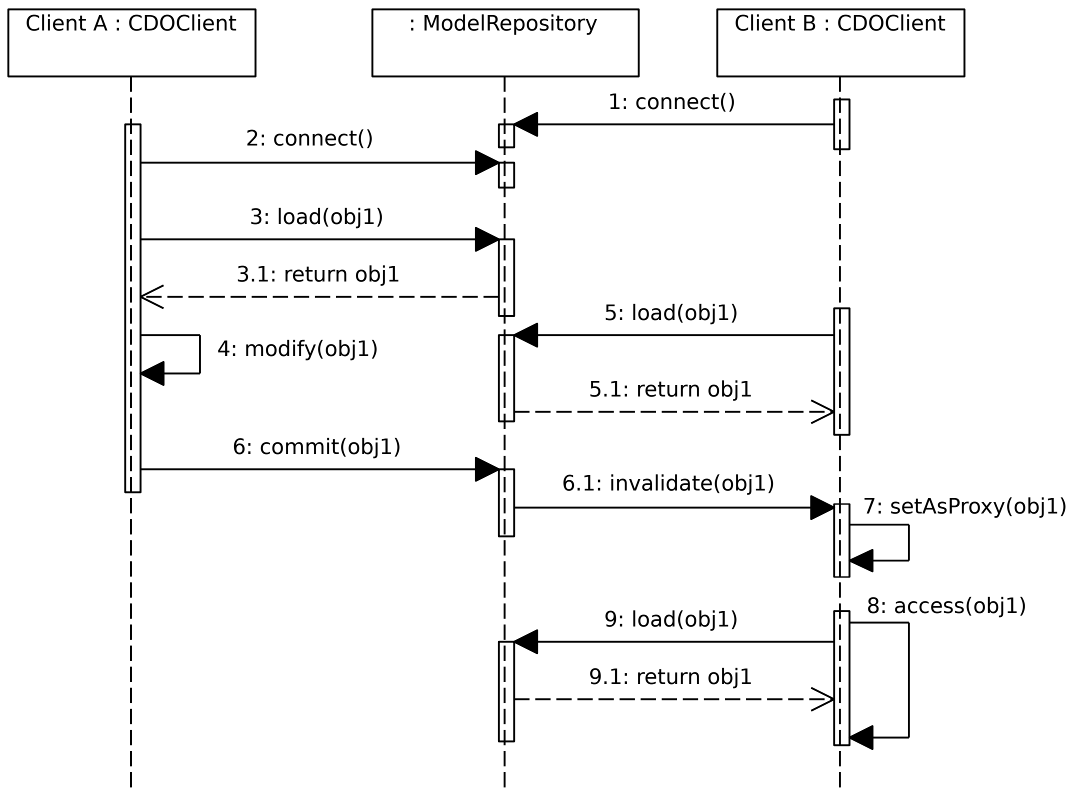

Figure 4.4. Consistency between CDO clients (based on [Stepper 2010])

and load the object *obj*1. After that, client *A* elicits some changes on the object and commits the changed object afterwards. This prevails the model repository to send *invalidate* messages to all other clients that set the modified object in proxy status. The garbage collector is now able to remove this object. If client *B* wants to access the object in proxy status, the object will be loaded transparently. This approach assures that the user always accesses the latest version of model objects without taking care of possible changes that have been made.

4.3.3 States of Objects

Figure 4.5 shows the different states that an object in the context of a CDO client can attain. When the object was created by the corresponding factory, it is "transient".

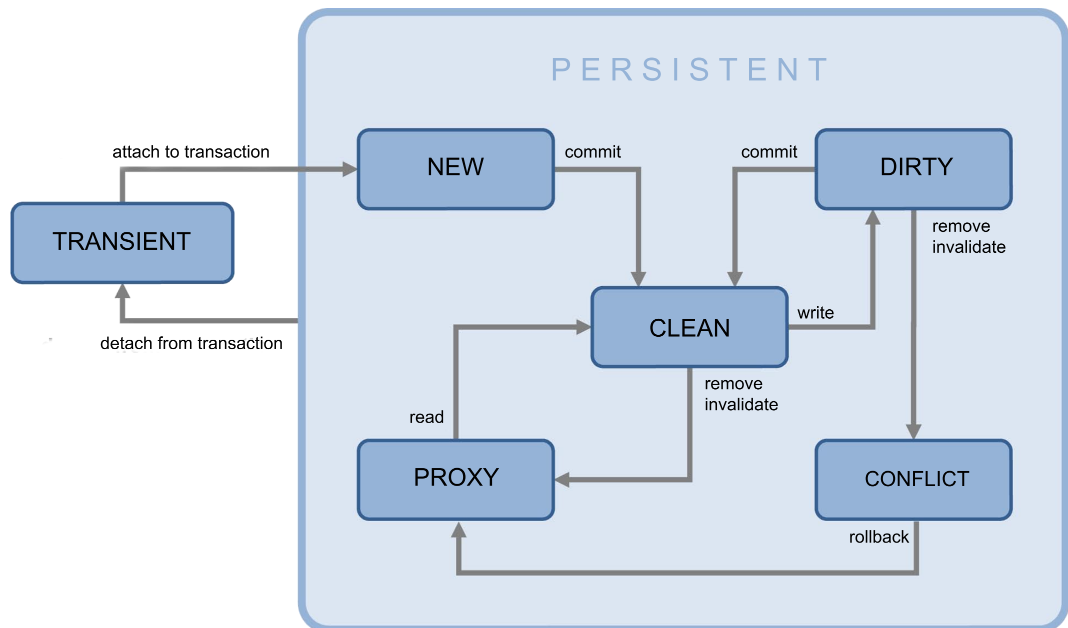

Figure 4.5. Object States in the Context of CDO [Stepper 2010]

After attaching it to a transaction it moves to the state "new". When the transaction was committed, the object becomes "clean". When objects in the "clean" state will be udpated they move to the "dirty" state. After a commit of the modifying transaction, the object becomes "clean" again. When another client has changed an object, this object moves to the "proxy" state for this client. When the client accesses the object in this state, the object will be loaded from the server again and becomes "clean". If a dirty object is changed by another client, a conflict arises and the transaction must be rolled back in order to solve this conflict.

4.3.4 Resources and Resource Folder

CDO uses the design pattern *Composite* to organize the models in resources and resource folders. Figure **??** shows the structure of the pattern. A resource folder can contain other resource folders or resources. This is not accidentally similar to a filesystem. A resource corresponds to an XMI file that we know from ordinary EMF persistence technology.

Table 4.1. Supported features of the different CDO stores, default values underlined

	Audits	Branches
DBStore	true/false	true/false
MEMStore	true/false	true/false
Db4OStore	true/false	true/false
MongoDBStore	true/false	true/false
HibernateStore	false	false

4.4 Existing CDO Stores

CDO is already shipped with six different CDO stores. The stores mainly differ in the used database backends. There are two different stores that use relational databases like MySQL, PostgreSQL, H2, etc. The HibernateStore uses Hibernate[3] as object-relational mapper (OR-mapper) while the DBStore implements its own OR-mapper, for good support of the CDO framework. Furthermore, there is the MongoDBStore that uses the NoSQL database MongoDB. You can also find a memory-based store called MemStore. The advantage seems to be obvious. Since it is not necessary to access the harddisk, this store is very fast. But on the other hand, the data gets lost when the server shuts down. Another CDO store is the DB4OStore which uses the object-oriented database DB4O[4]. All these CDO stores use open-source software. At last, there is an ObjectivityStore which is based on the proprietary object-oriented datbase Objectivity[5]. The latter store will not be further investigated since it is not based on open-source software.

Table 4.1 shows the supported features of the different CDO stores. The HibernateStore does not support either audits or branches. All other CDO stores support both but the default values can differ. The DBStore and the MEMStore support audits and branches per default while the DB4OStore and the MongoDBStore must be configured first.

4.5 Developing a Custom CDO Store

Figure 4.6 gives an overview of the class structure used by the CDO server. The `IRepositoryFactory` creates a `IRepository` that is represented by its name. A reposi-

[3] http://www.hibernate.org
[4] http://www.db4o.com
[5] http://www.objectivity.com

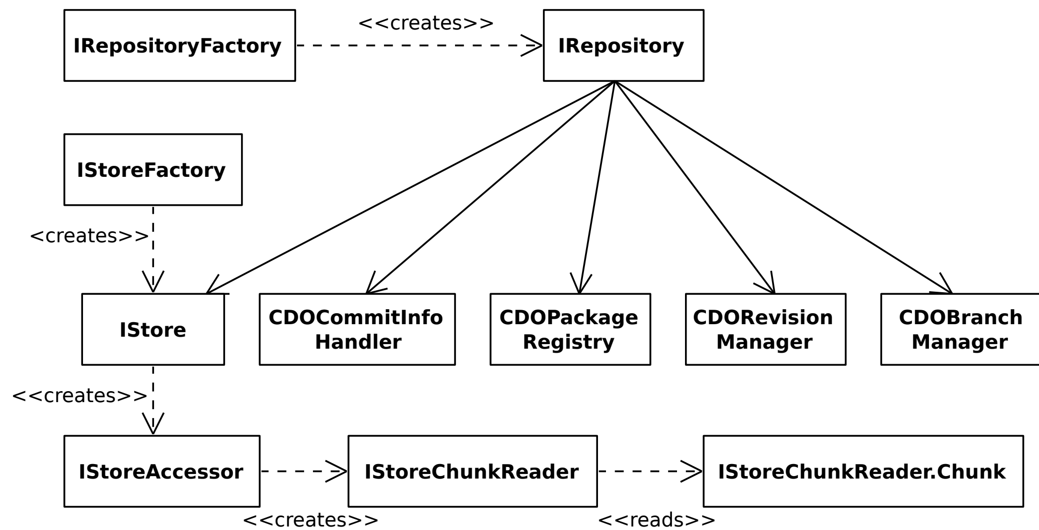

Figure 4.6. Overview of the high level class structure

tory combines several components like a `CDOCommitInfoHandler`, a `CDOPackageRegistry`, a `CDOBranchManager`, a `CDORevisionManager`, and CDO `IStore` that is created by a `IStoreFactory`. A configuration file can be specified by containing store specific settings e. g., the location of the database. The read/write access to the backend is provided by a `IStoreAccessor`. The store accessor will create a `IStoreChunkReader` if the CDO store supports partial collection loading. This means that only `IStoreChunkReader.Chunks` are read.

In order to develop a custom CDO store, three of these interfaces must be implemented.

Chapter 5

A CDO Store with Graph Database Backend

This chapter deals with the development of a CDO store based on a graph database backend, concretely Neo4j. Section 5.1 briefly describes the features that are supported respectively not supported by the our **Neo4jStore**. Then Section 5.2 discusses general design decisions, e. g., the mapping from the CDO revisions to nodes and relationships in the graph database. After that, Section 5.3 gives a top-level overview of the repository schema. Finally, this chapter deals with the concrete storage of meta-models in Section 5.4 and revisions in Section 5.5.

5.1 Supported Features

As depicted in the previous chapter, some features of the CDO model repository must be provided by the CDO store.

ChangeFormat

The suggested Neo4jStore only supports the *REVISION* format since this increases the performance for read operations. As explained below, the complete state of a revision is stored within one node, so the Neo4jStore must only read one node from the database in order to restore the values of a requested revision. If the Neo4jStore accepted revision deltas, the treatment would be more complex. By using the *REVISION* format, the Neo4jStore can transform the revision directly into a database node. So only one write operation is executed. If the *DELTAS* format was used, the Neo4jStore reads the unchanged parts of the revision from the last stored version in the database. Besides the write operation, an additional read operation is required.

The disadvantage of the chosen approach is the higher load on the network, which is accepted since the considered use cases do not involve many update operations.

Audits

As proposed earlier, versioning is an important capability of model repositories. The proposed Neo4jStore supports multiple versions despite the more complex treatment. If versioning was not supported, the old values of the revision can simply be overwritten with new values. But if versioning is supported, the access to different versions must be maintained. Furthermore, the latest version should be fast accessible since it is the most requested version.

Branches

The considered use cases do not require branches. Hence, the proposed Neo4jStore will not support branches. Nevertheless, this can be future work. Hence, the chapter **??** proposes extensions to the current implementation in oder to support branches.

Chunk Reader

This feature is essential in order to make queries faster. So, the Neo4jStore supports it. The implementation of the DBStore was adopted.

5.2 General Design Desicions

As mentioned before, models are represented as object graphs. So, it would be intuitive to map this graph directly to the database, such that a model object is represented as a node and a reference is represented as a relationship. This approach will not be suitable for a CDO store since CDO pursues the strategy of decoupled revisions. The object graph is flattened down into a list of revisions and references are represented as ID objects. Nevertheless, it would be possible to reconstruct the object graph, but creating new model objects or accessing existing ones would result in many read queries before the requested operation can be executed. On top of this, the CDO server requests only one model element simultaneously. This request is based on the CDO internal ID of the model

element. So there is no advantage of reconstructing the object graph, although the CDO server supports a cache which allows to store the children (in the sense of containment references) preventively. The creation of model objects is an expensive operation which makes it necessary to trade off model creation against read performance.

So, a CDO store requires another approach. The revisions must be fast retrievable. Many database managements systems already provides such a facility which are called indices. All CDO internal IDs will be stored in an index. If many model objects would be stored, a global index could become very large and the retrieval of the revisions could be slowed down. In order to mitigate this, the fact that each model element belongs to exactly one resource will be exploited. Each resource gets its own index on the IDs for its containing revisions. So, small resources can be treated faster than larger resources. Assumed that a user who works with large models would excuse more easily slower model access than users who work with small models.

Another important design issue concerns the access to graph database. There is a trade-off between the readability of the database queries and the performance. Since the normal API is faster than Cypher and the traversal API, it was made heavy use of the normal API despite its worse readability. We paid attention to the enclosure of the database access which allows to change the database access API.

A last minor decision concerns the storage of utility class objects like `java.util.Date` or `java.util.List`. Neo4j is only able to store primitive data types e. g., Boolean, Integer, or String. So, the utility class objects must be transformed into representations based on primitives. An object of type `java.util.Date` will be transformed into the long value of milliseconds since 01.01.1970 and a `java.util.List` is transformed into its string representation "[val1, val2, ...]".

5.3 General Overview of the Repository Schema

Although NoSQL databases do not need to rely on a schema like relational databases, a structure is necessary in order to work with the data. Figure 5.1 shows the six top-level nodes of the database. The node *properties* references a set of persistent properties that belong to the repository. Each persistent property is stored as a node with two values. The first value is the key of the property and the second is its actual value. In the case of the Neo4jStore, the only stored property

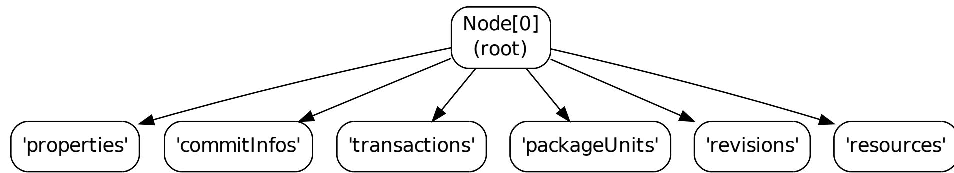

Figure 5.1. Overview of the database schema

is the creation time of the repository. If a property is requested, the neo4j store searches for the suitable key and returns its value.
The children of the node *commitInfos* represent meta information for each executed commit. It persists a comment, the user ID, and two timestamps. The first timestamp informs when the commit was executed, and the second timestamp indicates when the previous commit was executed.
The children of the next node *transactions* represent incomplete transactions. Each child references one or more nodes that were created within the corresponding transaction. More details are given in Section 5.5.
The node *packageUnits* references all persisted meta-models. By default, three meta-models will be created: the Ecore meta-model and two meta-models required by CDO. If a model element is persisted that conforms to a currently unknown meta-model, a new node for the meta-model will be created. A more detailed view will be given in Section 5.4.
The children fo the last two nodes represent the CDO internal revisions. The Neo4jStore distinguishes between revisions that represent resources respectively resource folders and revisions that represent model objects. The former are referenced by the node *resources* while the latter are referenced by the node *revisions*. The distinction is not inevitably necessary but it can be useful since the resources need also be retrievable through its name, while the revisions are only identified by its ids. So, for the resources only the path underneath the *resources* node must be considered in order to find the matching resource.

5.4 Persisting Meta-Models

The meta-models must be persisted in order to recover model elements from graph database nodes, because a node itself do not contain any information about its type. As already mentioned, the node *packageUnits* references all persisted meta-models. Each meta-model gets its own node which contains a set of properties. The first property is a unique id needed by CDO. The second property is the original type of the meta-model. The most important property is the package data property. It is used to store the ecore file as byte array. This byte array can be used to create a model object after the CDO server was shut down and the package registry is empty as mentioned in the previous chapter. Meta-models are organized into one or more packages which are represented by their packageURIs. Each package contains a list of classes. A class is represented by its name. The combination of the class name and the packageURI ensures that each class can be uniquely identified. A model element belongs to exactly one class, moreover, we can say it is an instance of exactly one class. Each revision node references exactly one meta-class.

Figure 5.2 shows the structure of the schema in the graph database. It contains references to three meta-models. The first meta-model contains two packages which contain meta classes. As shown in Figure 5.3, each revision has got a *INSTANCE_OF* relationship to a meta class. Section **??** discusses how this can be used in order to restore model objects.

5.5 Persisting CDO Revisions

5.5.1 General Approach

Model objects are represented as CDO revisions. A CDO revision consists on the one hand of meta information, like its CDO internal id and on the other hand of its properties. Figure 5.3 shows two objects and their related nodes in the database. Both, the meta information and the properties are stored in the same node. Another approach would be to create an additional node that conains the properties. This would result in more human-readable graphs but decreases the performance since more nodes must be created and commited and on the other side, more nodes must be read for retrieving the object. Associations between objects are represented by their CDO internal id which will be converted into a value of type long. Hence, the nodes do not have a direct relationship that indicates the association. A revision node itself does not know which kind of class it represents.

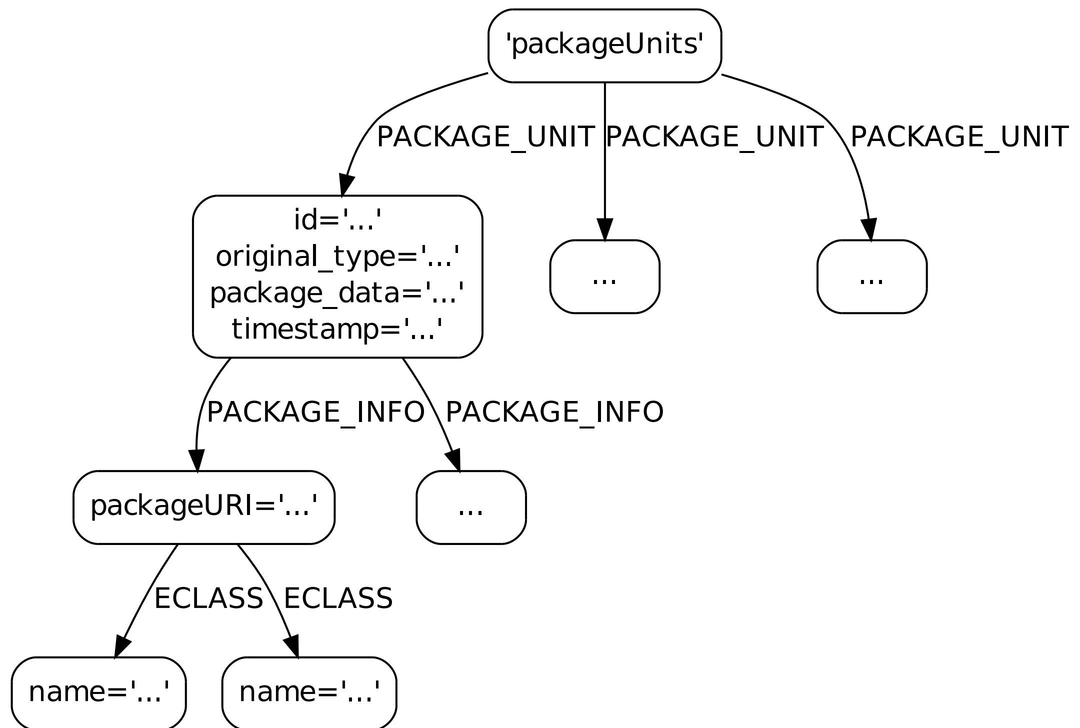

Figure 5.2. A meta-model stored in the database

This is ensured by assigning a meta class to the revision node. This is represented by the *INSTANCE_OF* relationship between the meta class node and the id node.

Updates on model objects implicate that multiple versions of the object exists. Since CDO allows the access to older versions, these information must be stored in the database. Nevertheless, the latest version must be faster accessible than older versions. The versioning in the Neo4jStore is implemented as follows. Each revision is represented by an *id node*. This node contains the CDO internal id and, furthermore, it contains a *VERSION* relationship to its latest version. If older versions exists, then, the latest version will contain another *VERSION* relationship to its predecessor verion. If this version also has earlier versions it also contains a *VERSION* relationship to that. Figure 5.4 shows three different revisions. Depending on how often the revisions were updated, they have different numbers of versions.

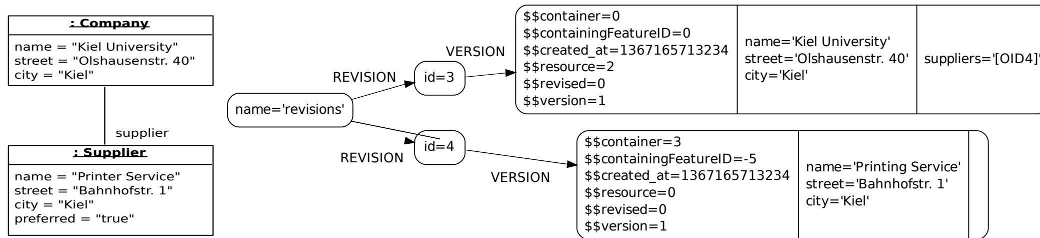

Figure 5.3. Mapping between object graph and neo4j store representation

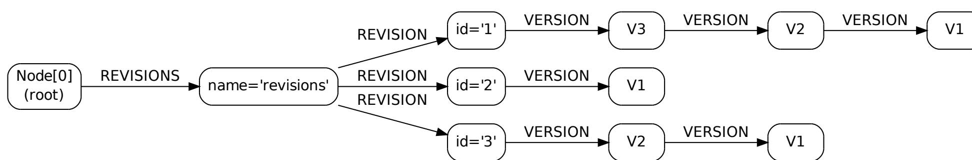

Figure 5.4. Three model elements with different versions

At this point, we want to talk about the evolution of meta-models. Normal CDO does not support the evolution because this would cause some problems since some cdo stores can support it e. g., the DBStore uses tables that represent the meta-model. If the meta class changes, the table must be changed. This would not work. Nevertheless, the Neo4jStore could support meta-model evolution. As you can see, the id node has got the relationship to the meta class. This relationship must be deleted and instead, each version node must get its own relationship to the meta class.

5.5.2 Consistency

After the general overview of the repository schema, this section discusses how the consistency of the models can be maintained. Two different approaches are possible. The first approach relies on the transaction management that is provided by Neo4j. This means that a CDO transaction is mapped to a Neo4j transaction. The Neo4j transaction will be committed when the CDO transaction is committed. This approach works well only for small models since Neo4j is not designed for large data insertion within one transaction as mentioned before. Hence, we propose

another approach that will be discussed now. First, the approach be described in detail and after that its correctness will be discussed.

The solution for large transactions it to cut the large transaction in small chunks of a few thousand revisions. Each chunk will be commited within its own transaction. This can lead to inconsistencies as the following example will show. Lets consider two chunks. The first chunk was committed correctly but during the second chunk, an error occurs and the transaction must be rolled back. The first chunk remained unaffected by the roll back. So, we have got an inconsistent database state. In order to maintain consistency, this approach must be extended. The small transactions must be glued to one large transaction in the graph database. The Neo4jStore creates a transaction node in the graph database which maps to the CDO transaction. Each revision that is created or updated in the transaction gets a relationship to the transaction node. This allows the Neo4jStore to commit or roll back the CDO transaction.

In order to get a clearer understanding, the procedure of the storage and update of revisions will be shown in detail. We assume that there is a Neo4jStore with a given database state shown in Figure 5.5. Only one revision is currently stored. Furthermore, there is only one version for this revision. Now, a CDO transaction is committed. Within this transaction, the revision with id=1 will be updated, i. e., a version 2 will be added and a new revision with id=" will be created.

writeRevisions

When CDO calls the writeRevisions method, the Neo4jStore creates a transaction node. Each revision that will be created within the global CDO transaction will get a relationship to this node. Two cases must be distinguished. The first case considers a new created revision. Remember that each CDO revision belongs to one id node and its corresponding version nodes. So, if we received a new revision, two nodes will be created. Both nodes get a relationship to the transaction node. The type of the relationship differ since in the case of a roll back, the id nodes must also be removed from the index. The second case considers revisions that are only updated. This means, that there already exists an id node and a corresponding version node. The target is now to insert the new version node between the id node and the, so far, latest version node in order to maintain the order of the versions proposed in the last section. Therefore, we create a new version node with a relationship to the transaction node. Furthermore, we create two relationships starting from the new version node. The first relationship models the order of

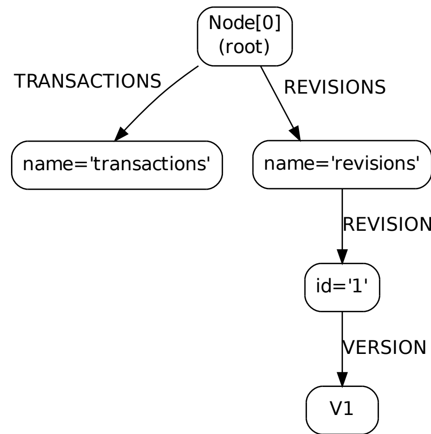

Figure 5.5. Database state before changes

versions for the model element. The second relationship goes to the id node. This allows to get the id node in a further step.

Figure 5.6 shows the state of the database after the writeRevisions method finished its work.

doCommit

When CDO invokes the doCommit method, the transaction node must be marked as committed first. Then, all outgoing relationships will be considered. If the node is a version node, then the pointer for the corresponding id node will be redirected. If the new revision is not the first version, then the VERSION relationship will be removed from the id node and after that the ID_NODE relationship will be reversed and the type changes to VERSION. Then the relationship to the transaction node will be removed. If the outgoing relationship points to an id node, then only the relationship will be removed an no further action is necessary. Finally the transaction node will be removed. This inticates the Neo4jStore that the transaction is completed.

Listing 5.1. Write revisions

```
CREATE node tx WITH tx.id = generateID();

FOR EACH chunk c FROM "committed chunks" DO
    BEGIN Neo4j TRANSACTION;
    FOR EACH revision r FROM c DO
        CREATE version_node WITH version_node.values = r.values;
        CREATE <TX_NODE>-relationship FROM tx TO version_node;
        IF r IS NEW IN REPOSITORY THEN
            CREATE id_node WITH id_node.id = r.id;
            ADD id_node TO INDEX;
            CREATE <TX_ID>-relationship FROM tx TO id_node;
        ELSE
            LET id_node = findNode(r.id);
            LET last_v_node = findLastVersion(id);
            CREATE <VERSION>-relationship FROM version_node TO last_v_node;
        END
        CREATE <ID_NODE>-relationship FROM version_node TO id_node;
    END
    COMMIT Neo4j TRANSACTION;
END
```

Figure 5.7 shows the database state after the doCommit method finished its work. The transaction node is not present anymore. Furthermore, a new revision with id=2 exists and the revision with id=1 has got a new version.

doRollback

After the case of a successful transaction, we will consider now what must happen if a transaction is aborted. When the CDO server calls the doRollBack method, first, the transaction node must be marked as rolled back. Then all outgoing relationships will be considered. If the relationship points to an id node, then this node must be removed from the index. For all referenced nodes, the outgoing relationship must be removed and after that the node itself will be removed. Finally, the transaction node will be removed. That indicates the Neo4jStore, that the transaction is committed as before.

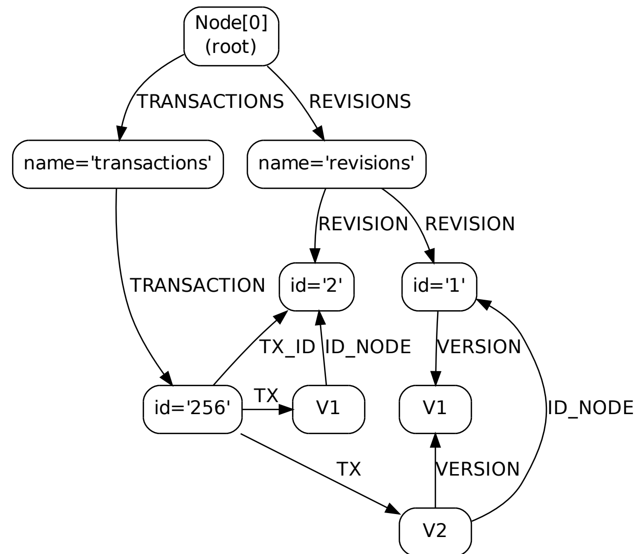

Figure 5.6. Database state after write revision

After the method finished its work, the database has got the same state as before the CDO transaction was started which is the intended behavior.

5.5.3 Correctness

After the procedure was explained, it remains the question whether this procedure behaves correct also in the case of system failures and concurrent access to the revisions.

Crash Tolerance

First, let us investigate what happens if the CDO server and the embedded graph datbase crashes? If the server crashes while the writeRevisions method is called, the problem will arise that maybe some chunks are committed and others are not. Since all new versions and object elements are marked by a relationship to the transaction node, the server can handle the crash after a restart. The transaction

Listing 5.2. Revision commit

```
LET tx = "transaction node";
MARKS tx AS "committed";
FOR EACH relationship r FROM tx DO
    LET node = r.EndNode;
    LET id = "corresponding id node for 'node'";
    IF node IS OF TYPE "version node" THEN
        IF node.version != 1 THEN
            REMOVE <VERSION>-relationship FROM id;
        END
        CREATE <VERSION>-relationship FROM idNode to node;
        REMOVE <ID_NODE>-relationship FROM node;
    END
    REMOVE r FROM tx;
END
REMOVE tx;
```

Listing 5.3. Revision rollback

```
LET tx = "transaction node";
FOR EACH relationship r FROM tx DO
    LET node = r.EndNode;
    IF node IS OF TYPE "id node" THEN
        REMOVE node FROM INDEX;
    END
    REMOVE ALL relationships from node;
    REMOVE node;
END
REMOVE tx;
```

node was not marked as committed or rolled back. So all nodes adjacent to the transaction node will be deleted as seen for the doRollback method. If the server crashes during the doRollback method was called, the server can handle the crash after restart. Since the transaction node was already marked as rolled back, the Neo4jStore knows that all adjacent nodes must be treated as in the doRollback method. If the server crashes during the doCommit method is called, the server

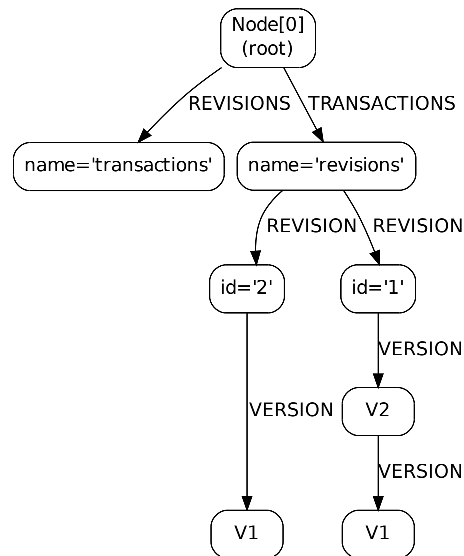

Figure 5.7. Database state after commit

can also handle the crash after restart. Since the transaction node was marked as committed, the Neo4jStore knows that maybe some nodes were already definitely committed and the relationships were removed. All nodes that still are adjacent to the transaction node can also be committed. So, the database state will be the same as the doCommit method was normally executed.

Concurrent Access to Revisions

Another threat is the concurrent access of revisions that are not finally committed. Lets assume that there exists a revision with one version. Now, transaction t_1 makes an update on this revision and adds the version V_2. Before the doCommit method is called for this transaction, another transaction also tries to add a new version V_2 to this element. This situation leads to *lost updates* known from database theory. This means that it is non-deterministic how the database state looks like after all transactions have finished. In order to avoid this problem, the Neo4jStore implements pessimistic locking. This means that no other transaction can modify a revision unless As Neo4j maintains the ACID properties, we can assume, that the new version and the relationship from the new version to the id node exist at the same time. Hence, before a new version will be created, we first check if there is

already a replacing version. If so, the server must be rolled back and the global CDO transaction is aborted.

5.5.4 Summary

As explained, the use of transaction nodes allows the Neo4jStore to cut large transactions into small chunks of transactions. This speeds up the Neo4jStore and allows more throughput. The last section comments on problems that could appear because of system failures or concurrent access. It was argued why the given procedure works correct regarding the consistency.

Chapter 6

Evaluation

This chapter describes the evaluation of our Neo4jStore. Section 6.1 deals with the statistical foundations. Section 6.2 deals with the machines on which the benchmark were executed. Finally, Section 6.3 presents the benchmarks used for evaluation.

6.1 Statistical Foundations

When benchmarks are applied to software systems, there are various sources of non-determinism, e. g., thread scheduling or garbage collection [Georges et al. 2007]. This means that if an experiment is repeated, the results will likely differ. Hence, it is not useful to take the first result but in the contrary, it is necessary to repeat experiments multiple times. Researcher work with samples from the population of possible results. When comparing alternative software sytems in relation to their performance, the samples must be compared. The question that arises is how to derive the actual performance of the system under test, with respect to a certain confidence? Some researchers answer this question by choosing the best or the second best result from the sample. Other researchers compute the mean of the sample. Georges et al. [2007] show that these approaches can lead to wrong interpretations. Firstly, Section 6.1.1 introduces a graphical representation of a sample which gives the opportunity to get a first impression of the distribution of the underlying population. Afterwards, they propose a statistical rigorous performance evaluation, which will be discussed in Section 6.1.2.

6.1.1 Box-And-Whisker Plots

Let $x_1, ..., x_n$ be a sample taken from a repeated experiment and let $x_{(1)} \leqslant ... \leqslant x_{(n)}$ the sample with ascending order such that $x_{(1)}$ is the **minimum** of the sample and $x_{(n)}$ is the **maximum**. There are three values called **quartiles** which separates the sample into four equal parts [Hedderich and Sachs 2012]. The first quartile Q_1 means that 25% of the values are smaller than Q_1. The second quartile Q_2, also

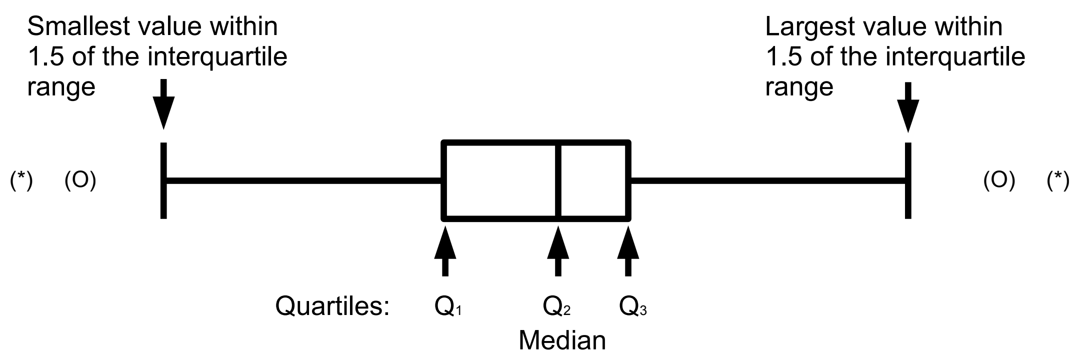

Figure 6.1. Structure of a Box-and-Whisker plot [Kähler 2010]

called **median**, separates the sample in two halfs, i. e., 50% are larger and 50% are smaller than this value. The third quartile Q_3 separates the sample such that 25% are larger than Q_3. The values can be computed as follows [Hedderich and Sachs 2012]:

1. $Q_1 = x_{(k)}$ with $k = \lfloor (n+1) \cdot 0.25 \rfloor$

2. $Q_2 = x_{(l)}$ with $l = \lfloor (n+1) \cdot 0.50 \rfloor$

3. $Q_3 = x_{(m)}$ with $m = \lfloor (n+1) \cdot 0.75 \rfloor$

The difference of the third and the first quartile is called the **interquartile range** (IQR). It consists of the centralized 50% of the sample.

Box-and-Whisker plots represent these values graphically and and offer a first opportunity to decide whether a sample could be normally distributed. Figure 6.1 shows the structure of a Box-and-Whisker plot. The first and third quartiles will be drawn as a rectangle. The median is represented as a line within the rectangle. The smallest value $x_{(i)} \geqslant Q_1 - 1.5 \cdot IQR$ respectively the largest value $x_{(j)} \leqslant Q_3 + 1.5 \cdot IQR$ are marked with so-called whiskers. All values which are smaller than $x_{(i)}$ respectively larger than $x_{(j)}$ are so-called outliers which are marked with (o). Extreme outliers ($3.0 \cdot IQR$ instead of $1.5 \cdot IQR$) will be marked with (*).

Normal distributions result in symmetric B-n-W plots, but they can also be caused by other symmetric data-sets/samples.

6.1.2 Choosing an Appropriate Statistical Test Method

Georges et al. [2007] propose statistics theory as a rigorous data analysis approach in order to tackle non-determinism. They propose two approaches depending on the number of alternatives that will be compared. If there are only two alternatives it will be sufficient to compare the confidence intervals. This method will be discussed in Section 6.1.3. In the case of more than two alternatives, they propose to use the *Analysis of Variance test* (ANOVA). However, ANOVA is a parametric test which means that there are two preconditions that must be satisfied that the test can be applied. The first condition for ANOVA is that the samples must be normally distributed. The second condition is that the variance of the samples must be equal.

Figure 6.2 shows the tests that must be executed in order to receive a statistical rigorous performance evaluation. At the beginning, it must be decided how many alternatives will be compared as mentioned before. If more than two alternatives will be compared, a test for normal distribution, such as a Shaprio-Wilk Test (see Section 1.1.4), must be applied. If the samples for all alternatives are normally distributed, they must be examined concerning the equality of variance. There are several tests that deal with this question. In this thesis, the *Bartlett Test* was chosen since it is a non-sensitive test for the equality of the variance. It will be considered in Section 6.1.5. If the Bartlett test confirms the equality of variances, the ANOVA can be applied to the alternatives in order to discover performance differences. However, this test can only confirm that differences exist but it is usually interesting which alternatives differ concerning the ranking. In the case of differences, the *Tukey Test* helps to find performance differences between alternatives.

The ANOVA is ineligible for the benchmark evaluations in this thesis because the preconditions were not fulfilled, hence the ANOVA as well as the Tukey test will not be considered in detail.

If one of the preconditions is not valid only a non-parametric test can be applied. A common test is the *Kruskal and Wallis Test*. It has got the same hypothesis as ANOVA but since it is a non-parametric test, it is not as robust as the ANOVA. Nevertheless, it has got an asymptotic efficiency of 95% [Hedderich and Sachs 2012]. Section 6.1.6 considers the test in detail. The Kruskal and Wallis Test can only determine whether there are differences between the alternatives or not, but it cannot determine which alternatives differ. Therefore, the average ranks will be compared pairwise shown in Section 6.1.7.

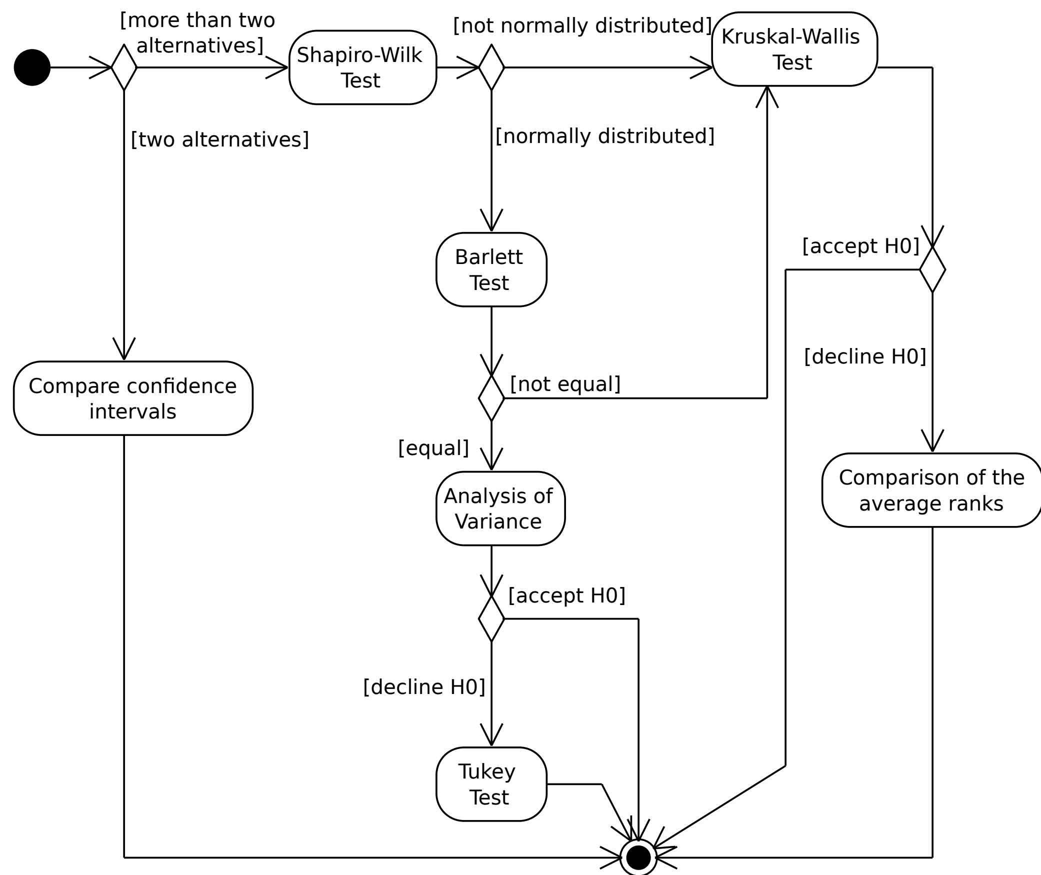

Figure 6.2. How to choose the appropriate statistical test method? (based on [Hedderich and Sachs 2012] and [Georges et al. 2007])

6.1.3 Overlapping Confidence Intervals

This section is based on [Georges et al. 2007]. A 95% confidence interval for a sample is an interval in which the actual distribution mean of the underlying population is contained with a probability of 95%. The size of the confidence interval grows with the given probability. This means that a 99% confidence interval is larger than the 95% confidence interval. Georges et al. [2007] emphasize that the underlying distribution does not have to be normally distributed because of the central limit theorem. Hence, it is not necessary to apply a test of normality like for the ANOVA.

In order to compare two alternatives, their confidence intervals will be compared. Consider two alternatives with n_1 respectively n_2 measurements. The first step is to compute the means $\overline{x}_1$ and $\overline{x}_2$ for both samples:

$$\overline{x}_i = \frac{\sum_{j=1}^{n_i} x_j}{n_i}, i \in \{1,2\}$$

After that, the difference of the means $\overline{x}$ can be computed:

$$\overline{x} = |\overline{x}_1 - \overline{x}_2|$$

The next step is to determine the standard deviations s_1, s_2 for the two samples:

$$s_i = \sqrt{\frac{\sum_{j=1}^{n_i} (x_j - \overline{x})^2}{n_i - 1}}, i \in \{1,2\}$$

Then, the standard deviation s_x of the difference of the means can be computed:

$$s_x = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}$$

Finally, the confidence interval for the difference of the means can be computed:

$$c_1 = \overline{x} - z_{1-\alpha/2} s_x$$
$$c_2 = \overline{x} + z_{1-\alpha/2} s_x$$

The value for $z_{1-\alpha/2}$ is taken from the table of the standard normal distribution, e. g., from [Fahrmeir et al. 2009]. So, with $\alpha = 0.05$ we get $z_{1-\alpha/2} = 1.96$. Finally, it can be concluded that there is not any statistical significant difference if $0 \in [c_1, c_2]$. The means that the performance for both alternatives must be considered to be the same. If $0 \notin [c_1, c_2]$, then it can be assumed that one alternative has got a better performance due to the statistic difference.

6.1.4 Shapiro-Wilk Test

In order to decide whether a sample $x_1, ..., x_n$ is normally distributed, the Shapiro-Wilk Test can be applied. The null hypothesis for the test states that the underlying population is normally distributed, while the alternative hypothesis states that the underlying population is not normally distributed. The test statistic can be computed as follows.

$$\hat{W} = \frac{\left(\sum_{i=1}^{n} a_i x_{(i)}\right)^2}{\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

$x_{(i)}$ are the measurements in ascending order and a_i are constant values taken from a table of a normally distributed variable with sample size n. The null hypothesis must be declined if $\hat{W} < W_{critical}$ which can be taken from corresponding tables, for example in [Pearson and Hartley 1976].

6.1.5 Bartlett Test

This test checks whether the variances of k samples are equal. The null hypothesis states homogeneity of the variances and the alternative hypothesis states that at least two variance are different. The test statistic is computed as follows [Georges et al. 2007]:

$$\hat{\chi}^2 = \frac{1}{c}\left(2.3026(\nu \cdot \lg(s^2) - \sum_{i=1}^{k} \nu_i \cdot \lg(s_i^2))\right)$$

with

$\nu = n - k$, overall degree of freedom
$s^2 = \frac{\sum_{i=1}^{k} \nu_i s_i^2}{\nu}$, estimation of the weighted variance
$\nu_i = n_i - 1$, degree of freedom for the i-th sample
s_i^2, estimation of the variance for the i-th sample

For "large" ν_i, it can be assumed that $c = 1$ and hence, it must only be computed if $\hat{\chi}$ is expected to be statistically significant. Then

$$c = \frac{\sum_{i=1}^{k} \frac{1}{\nu_i} - \frac{1}{\nu}}{3(k-1)} + 1$$

The null hypothesis is declined when $\hat{\chi} > \chi^2_{\nu;\alpha}$ whereby $\chi^2_{\nu;\alpha}$ is taken from the χ^2-distribution.

6.1.6 Kruskal-Wallis Test

This test will be applied in order to decide whether $k \geqslant 3$ samples are derived from the same population. So, the null hypothesis is that the k distribution functions are equal and the alternative hypothesis states that at least two distribution functions are different. Assuming that each sample i consists of n_i measurements $x_{i1}, ..., x_{in_i}$, there are altogether $n = \sum_{i=1}^{k} n_i$ measurements. It can now be assumed that $n_i = n_j$ for all $i, j \in \{1, ..., k\}$. These measurements will be sorted in ascending order so that each measurement can be ranked. Then, the ranks will be added up for each sample. Let R_i the sum for the i-th sample. The test statistic can be computed as follows:

$$\hat{H} = \left(\frac{12k}{n^2(n+1)}\right)\left(\sum_{i=1}^{k} R_i^2\right) - 3(n+1)$$

As stated in [Hedderich and Sachs 2012], if $n_i \geqslant 5$ and $k \geqslant 4$ then $\hat{H}$ is χ^2-distributed with $k-1$ degree of freedom, i. e., H_0 is declined when $\hat{H} > \chi^2_{k-1;\alpha}$. This value can be taken from a table of the χ^2 distribution [Hedderich and Sachs 2012].

6.1.7 Multiple Pairwise Comparison of Average Ranks

The Kruskal-Wallis-Test can only determine whether there are differences between the populations. For this purpose the average ranks will be compared. The average rank for a sample i is defined as $\overline{R_i} = R_i/n_i$. The null hypothesis for this test states that the samples i, j are derived from the same population and the alternative hypothesis states that the underlying populations are different. The null hypothesis will be declined when

$$|\overline{R}_i - \overline{R}_j| > \sqrt{d \cdot \chi^2_{k-1;\alpha} \cdot \left(\frac{n(n+1)}{12}\right)\left(\frac{1}{n_i} + \frac{1}{n_j}\right)}$$

In the context of the thesis, it can be assumed that $d = 1$, since there are not many repetitions of the same measurements.

6.2 Execution Environment

All tests were performed under repeatable and controlled conditions. The "Software Performance Engineering Lab" (SPEL) [1] has been used to assure these conditions. The tests were executed with the following hardware and software:

1. 2x Intel Xeon E5540 (2.53 GHz; 4 cores)
2. 24 GB RAM
3. Debian 6.0.7
4. OpenJDK Runtime Environment (IcedTea6 1.8.13)

For all benchmarks, the CDO model repositories receive 8GB RAM and the clients receive 4GB RAM.

6.3 Benchmark for CDO

6.3.1 The Models

The benchmarks are based on a meta-model that is shipped with the CDO sources. Models that relate to the meta-model describe a company and its relationships to suppliers and customers. Figure 6.3 shows the meta-model in detail. The root of a model is the `Company`. The company is `Addressable` and can have several suppliers and customers. `Supplier` and `Customer` are also `Addressable`. A company offers different products that are organized in different categories. So, each `Category` can contain further categories or products. Each `Product` has got several properties and can be bought from suppliers or sold to customers. The company has two types of orders, purchase orders and sales orders. Both are a specializations of an `Order`. A `PurchaseOrder` belongs to a supplier and a `SalesOrder` belongs to a customer. An `Order` can be comprised of order details. Each `OrderDetail` belongs to a product of the company.

6.3.2 Benchmark Design

We defined the following four scenarios for the benchmark: "model import", "model export", "small query", and "large query". Each category considers another use case

[1] `http://se.informatik.uni-kiel.de/research/spel/` (last visit: June 3, 2013)

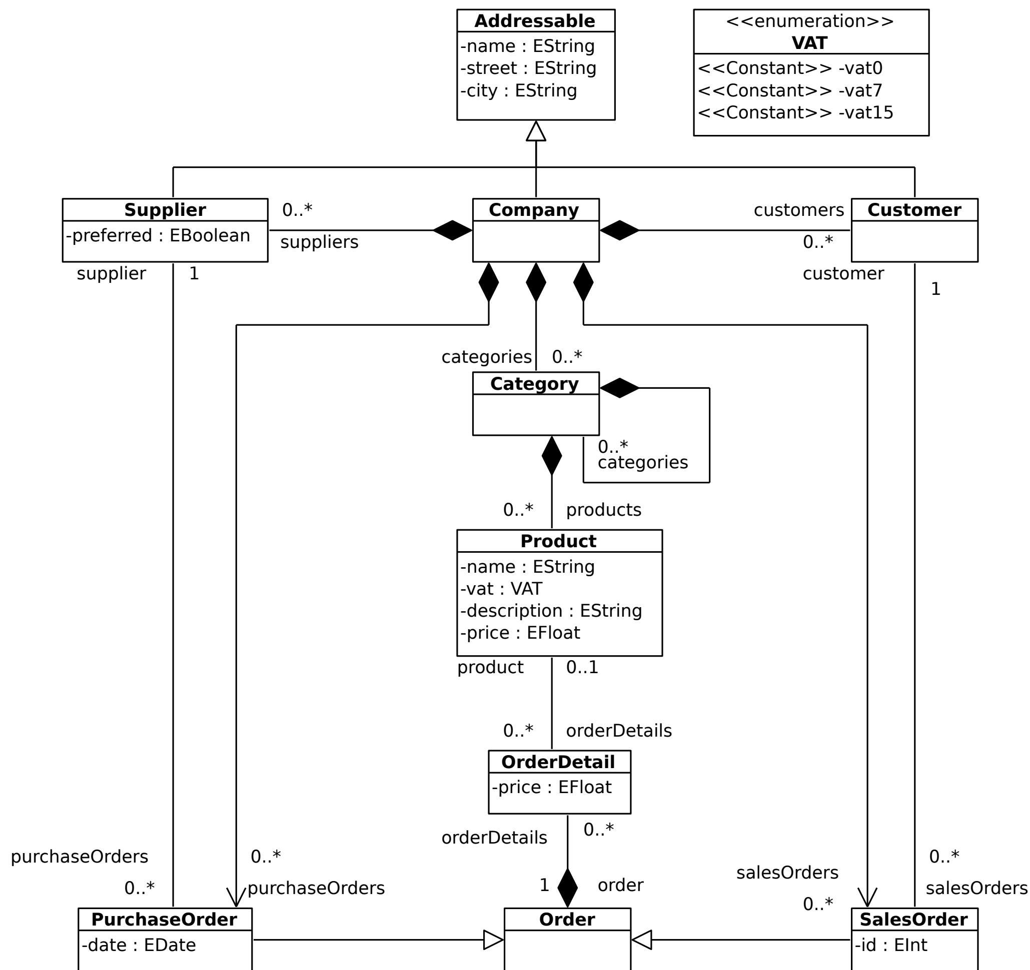

Figure 6.3. Used Meta-Model for Evaluation

of the model repository. "**Model import**" deals with the performance when storing models within one transaction. This means that XMI files will be loaded from the file system and stored into the model repository. "**Model export**" considers the performance when loading large models from the model repository and writing it to a XMI file on the file system. The category "**Small query**" considers the performance when models are queried and only small parts of the model are

Table 6.1. Models considered in the benchmark

	XS	S	M	L	XL	XXL
Categories	1	3	7	14	27	54
Products	5	44	219	614	2058	4675
Suppliers	7	15	29	37	67	98
Customers	9	14	29	47	63	81
Sales orders (SO)	99	200	665	1443	2929	3627
SO details	1056	2875	14,769	42,723	130,365	161,328
Purchase orders (PO)	250	812	2135	3041	5911	8532
PO details	9373	45,257	158,884	244,375	514,469	741,972
Overall size	10,800	49,220	176,737	292,294	665,889	920,367

affected. On the other side, "**large query**" considers the performance when nearly the complete model is affected.

All scenarios are applied to six different models which are shown in Table 6.1. These models were created randomly and hence the underlying model tree is not balanced. The smallest model consists of only 10,800 objects while the largest model consists of 920,367 objects.

The "model import" scenario compares our Neo4jStore with the MEMStore, DBStore, DB4OStore, and the MongoDBStore. The "model export", "small query" and "large query" scenarios compare only the Neo4jStore with the DBStore. It is not useful to take the other CDO stores into account since the results are not meaningful. The MEMStore loads the model from the cache while the other CDO stores must perform I/O-operations. The DB4OStore has got a bug which makes it impossible to load models after a restart of the CDO server. The MongoDBStore was not compared since it can not handle large models which is the core of this thesis. It has not got the ability to handle models larger than the defined size "m". The reason is that the store writes one transaction to one document. A document has got a maximum size of 16MB, but this size will be exceeded for large transactions.

6.3.3 Benchmark Results

Scenario 1: Model Import

Figure 6.4 gives a general overview of the distributions for all model sizes. The box-and-whisker plots are grouped by the CDO stores. For each store, the distribution on the left side relates to the model XS and then, they are ordered by model size

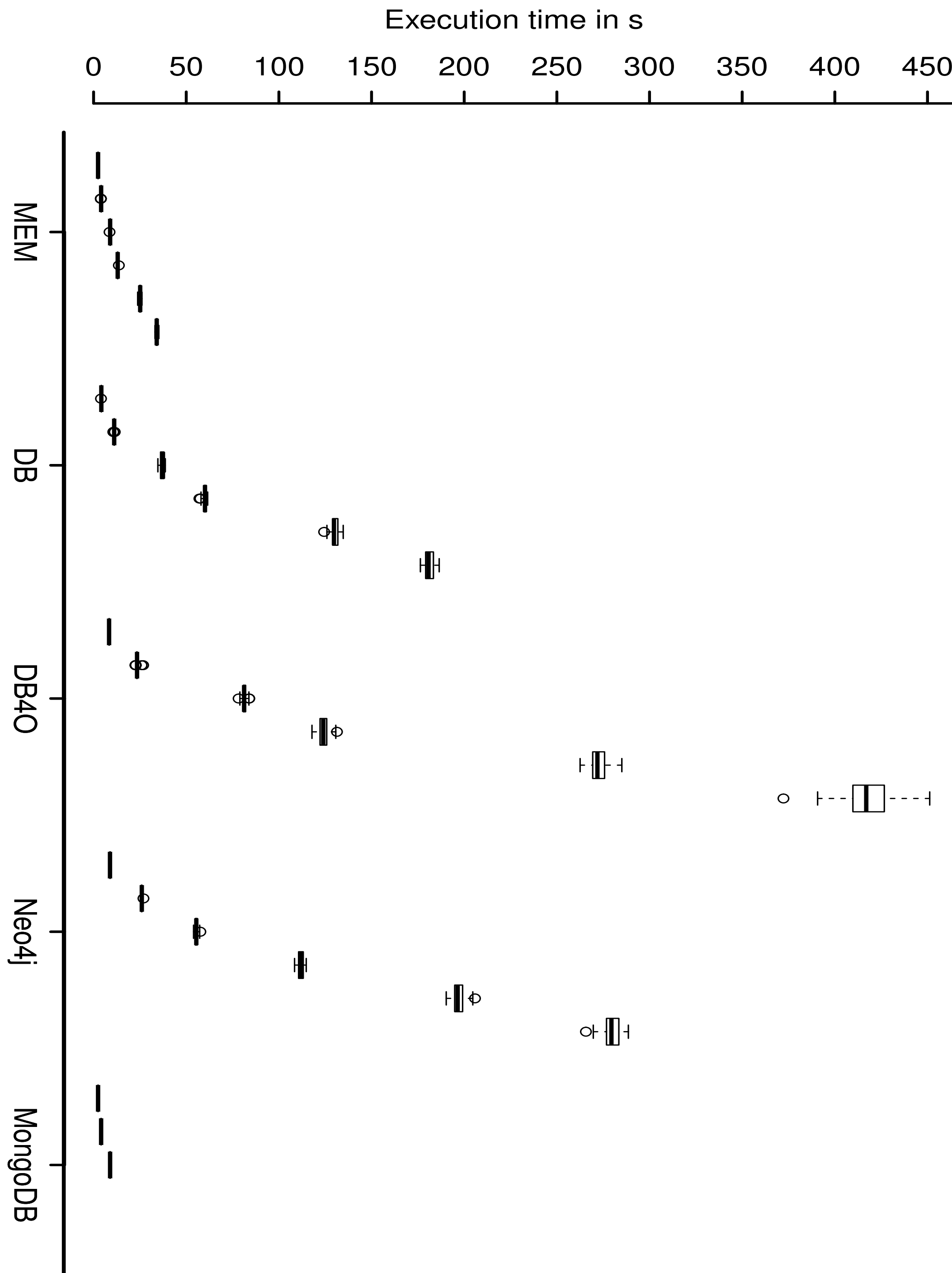

Figure 6.4. Box-and-Whisker plots for the "model import" benchmark

Table 6.2. Test statistics for the "model import" scenario

	XS	S	M	L	XL	XXL
MEM	0.9601	**0.9100**	0.9818	0.9832	0.9806	0.9838
DB	0.9642	0.9710	**0.9027**	**0.9500**	0.9698	0.9718
DB4O	0.9630	**0.6044**	0.9679	0.9844	0.9790	0.9832
Neo4j	0.9811	0.9716	0.9778	0.9724	0.9766	0.9762
MongoDB	**0.9370**	0.9626	0.9753	-	-	-
Normally distributed?	X	X	X	$\surd$	$\surd$	$\surd$
Bartlett Test	-	-	-	-	163.24	380.13
Kruskal-Wallis Test	227.11	225.88	227.39	186.57	186.57	186.57

Table 6.3. CDO stores comparison for model size XS

	MEM	DB	DB4O	Neo4j	MongoDB	Ranking
MEM	-	$\surd$	$\surd$	$\surd$	o	(1)
DB	-	-	$\surd$	$\surd$	-	(3)
DB4O	-	-	-	$\surd$	-	(4)
Neo4j	-	-	-	-	-	(5)
MongoDB	o	$\surd$	$\surd$	$\surd$	-	(1)

such that the distribution on the right side relates to the model XXL. All CDO stores have exponential behavior. While the MEMStore and the MongoDB grow moderately, the other stores grow fast. Now, we will consider the performances of the CDO stores per model. Table 6.2 shows the test statistics for the "model import" scenario. The values were computed as explained in Section 6.1. The values in the rows with the CDO store names represent the results of the Shapiro-Wilk test on normality. If all CDO stores are normally distributed for a particular model size, then this model size is marked with $\surd$. In this case the Bartlett test on equality of variance was executed. This concerns the model sized L-XXL. All Bartlett test results indicate that the particular variances are not equal. So, we can conclude that the Kruskal-Wallis test must be applied for all model sizes. Finally this test indicates that there are considerable differences between the CDO stores. The average ranks must be compared for each model size.

Table 6.3 shows the results of the compared average ranks for the model size XS. It reveals a uniquer order of the achieved performances of the CDO stores. The

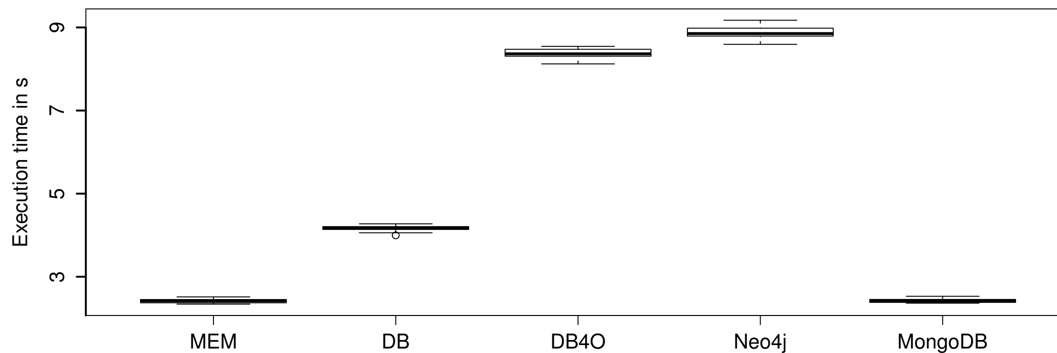

Figure 6.5. "Model import" scenario results for model size XS

Table 6.4. CDO stores comparison for model size S

	MEM	DB	DB4O	Neo4j	MongoDB	Ranking
MEM	-	$\surd$	$\surd$	$\surd$	o	(1)
DB	-	-	$\surd$	$\surd$	-	(3)
DB4O	-	-	-	$\surd$	-	(4)
Neo4j	-	-	-	-	-	(5)
MongoDB	o	$\surd$	$\surd$	$\surd$	-	(1)

best performances were given by the MEMStore and the MongoDBStore. This is not surprising for the MEMStore. The good performance of the MongoDBStore could be arised from the fact, that the MongoDB is started in a separate process. So, the combination of CDO server and MongoDBStore could use more memory than the other combinations. can be considered as equal. The third best CDO store is the DBStore followed by the DB4OStore and the Neo4jStore. Figure 6.5 shows the distributions for the model XS. The order that was computed by the statistical tests can be graphically confirmed.

Table 6.4 shows the results of the compared average ranks for the model size S. It is the same result as for the smaller model XS. Also the gap between the CDO stores are equal as depicted in Figure 6.6.

Table 6.5 shows the results of the compared average ranks for the model size M. The MEMStore and the MongoDBStore still provides the sam performance. Both stores are followed by the DBStore. At this point there is a change compared to the smaller models. the Neo4jStore has got a better performance than the DB4OStore.

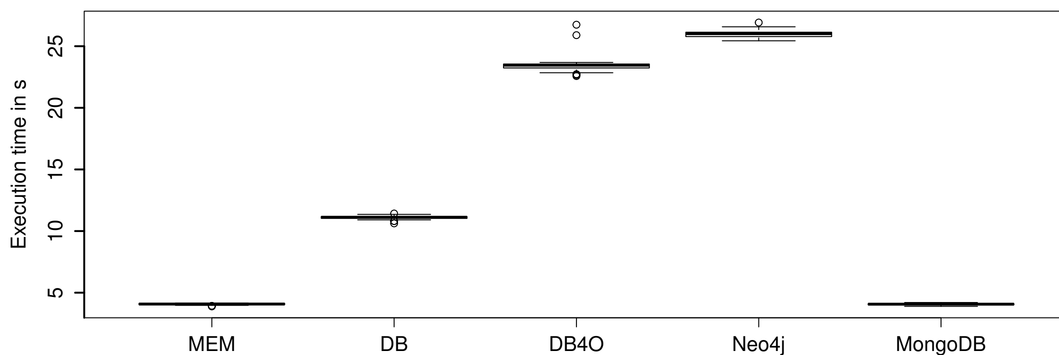

Figure 6.6. "Model import" scenario results for model size S

Table 6.5. CDO stores comparison for model size M

	MEM	DB	DB4O	Neo4j	MongoDB	Ranking
MEM	-	$\surd$	$\surd$	$\surd$	o	(1)
DB	-	-	$\surd$	$\surd$	-	(3)
DB4O	-	-	-	-	-	(5)
Neo4j	-	-	$\surd$	-	-	(4)
MongoDB	o	$\surd$	$\surd$	$\surd$	-	(1)

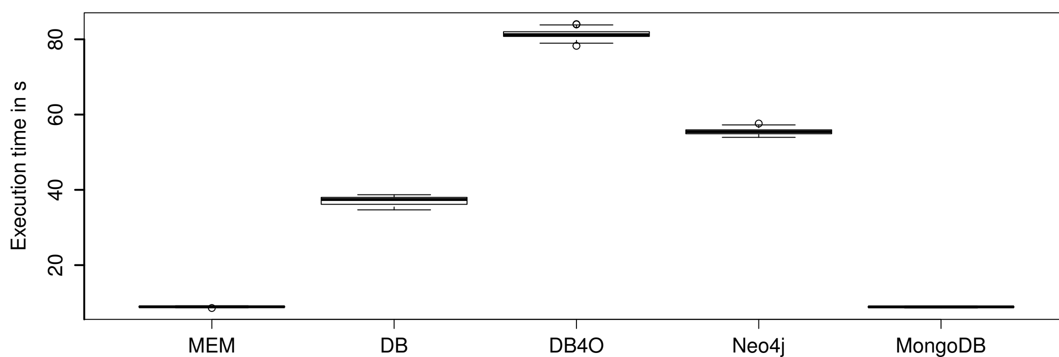

Figure 6.7. "Model import" scenario results for model size M

Table 6.6. CDO stores comparison for model size L

	MEM	DB	DB4O	Neo4j	Ranking
MEM	-	√	√	√	(1)
DB	-	-	√	√	(2)
DB4O	-	-	-	-	(4)
Neo4j	-	-	√	-	(3)

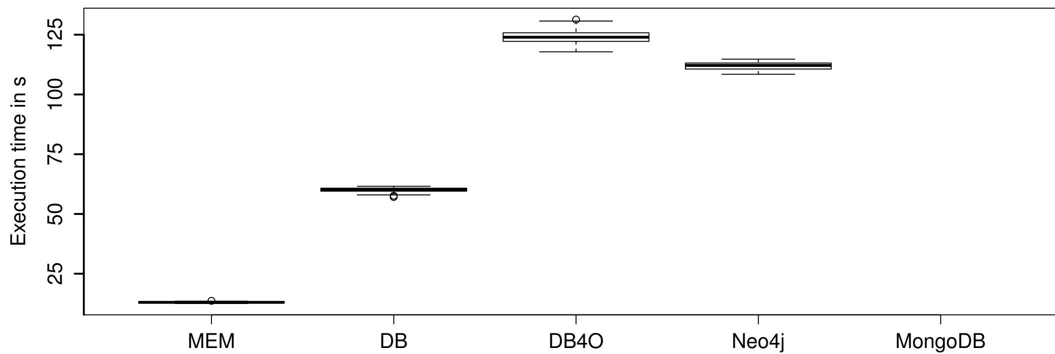

Figure 6.8. "Model import" scenario results for model size L

This is also the result of the graphical representation depicted in Figure 6.7. It seems that the performances of the DBStore and the Neo4jStore move closer together.

Table 6.6 shows the results of the compared average ranks for the model size L. As mentioned before, the MongoDBStore is not able to handle models of this size. Hence, the MEMStore has got the best performance. The rest of the ranking is the same as for the model size M which can also be graphically proved in Figure 6.8.

Table 6.7. CDO stores comparison for model size XL

	MEM	DB	DB4O	Neo4j	Ranking
MEM	-	√	√	√	(1)
DB	-	-	√	√	(2)
DB4O	-	-	-	-	(4)
Neo4j	-	-	√	-	(3)

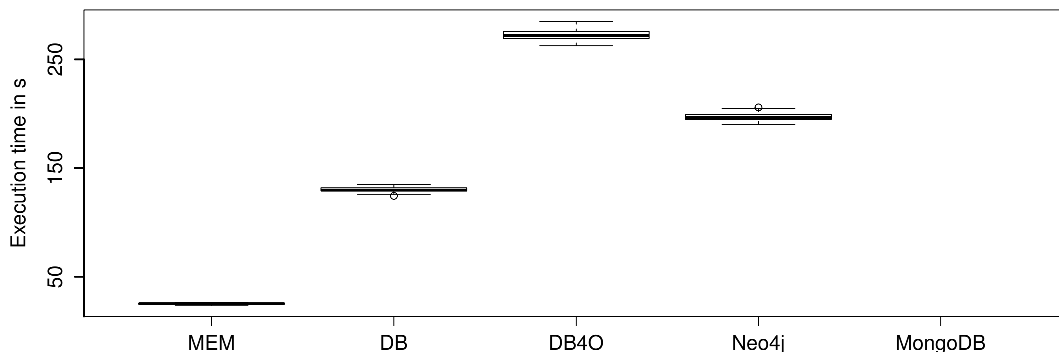

Figure 6.9. "Model import" scenario results for model size XL

Table 6.8. CDO stores comparison for model size XXL

	MEM	DB	DB4O	Neo4j	Ranking
MEM	-	$\surd$	$\surd$	$\surd$	(1)
DB	-	-	$\surd$	$\surd$	(2)
DB4O	-	-	-	-	(4)
Neo4j	-	-	$\surd$	-	(3)

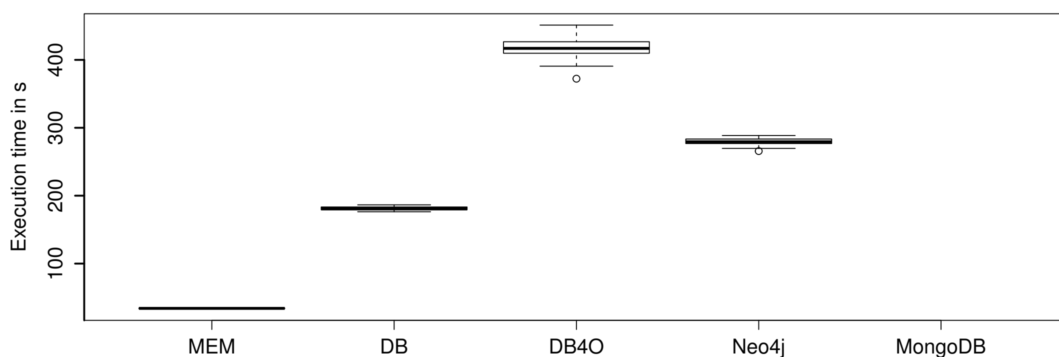

Figure 6.10. "Model import" scenario results for model size XXL

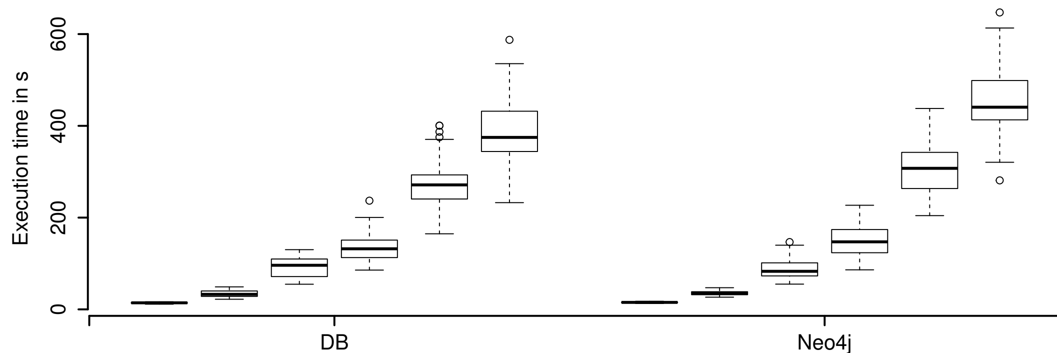

Figure 6.11. Box-and-Whisker plots for the "model export" benchmark

This rankings are the same for the model size XL and XXl as described in Table 6.7 and Table 6.8. There is only a variation of the gaps between the CDO stores as depicted in Figure 6.9 and Figure 6.10.

For the "model import" scenario we can conclude that the model size has not large influence on the ranking of the CDO stores. It is not surprising that the MEM-Store has got the best performance. The good performance of the MongoDBStore can be explained by the additional process for the database that allocates more memory than the other CDO stores. The DBStore has always a better performance compared with the Neo4jStore.

Scenario 2: Model Export

For this scenario, only the DBStore and the Neo4jStore will be considered as explained in Section 6.3.2. Hence, we can compare the confidence intervals in order to get a ranking for both CDO stores. Figure 6.11 shows an overview of the distributions for the different models. Compared to the previous scenario, the ranking can not be directly taken from the box-and-whisker-plot. Table 6.9 shows the results for the confidence intervals. These results show that the DBStore performs better than the Neo4jStore in general. For smaller models (S and M) there are not statistically significant differences between these store since the confidence intervals contain the 0. It seems that the model size has got influence on the performance. The gap between the means of the CDO stores grows with the model size.

Table 6.9. "Model export" performance comparison

	XS		S		M		L		XL		XXL	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
DB	14111	1167	34078	7155	92537	21170	134399	30716	274582	53049	384079	82120
Neo4j	15235	1133	35543	4772	87854	21796	148660	32410	308522	56853	453240	80744
Confidence interval	[673, 1575]		[−918, 3850]		[−3739, 13106]		[1884, 26639]		[12147, 55732]		[37239, 101084]	

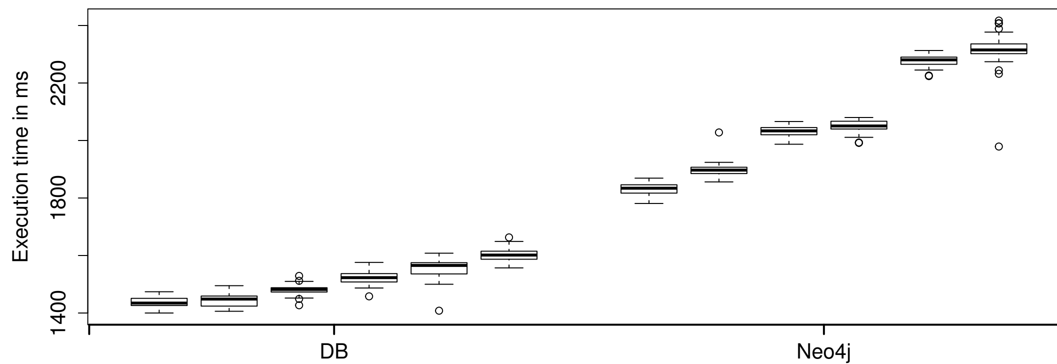

Figure 6.12. Box-and-Whisker plots for the "small query" benchmark

Scenario 3: Small Query

This scenario only compares the DBStore and the Neo4jStore which allows us to compare their intervals as statistical test. Figure 6.12 shows the distributions for the different model sizes. It is obvious that the DBStore provides a much better performance than the Neo4jStore. Nevertheless, a statistical analysis was done and the results a presented in Table 6.10. The confidence intervals confirm the obseravation since they do not contain 0.

Scenario 4: Large Query

This scenario also compares the DBStore and the Neo4jStore. The results, depicted in Figure 6.13, promise that the performance of the stores are closer than for the "small query"scenario. This can be confirmed by the statistical test. Its result is presented in Table 6.11. The results are very similar to the results of the "model export" scenario. This is not very surprising since the underlying use cases are very similar.

Table 6.10. "Small query" performance comparison

	XS		S		M		L		XL		XXL	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
DB	1437	19	1445	22	1480	18	1522	21	1556	32	1600	24
Neo4j	1832	20	1898	23	2031	18	2049	21	2276	19	2315	60
Confidence interval	[387, 402]		[445, 463]		[544, 558]		[519, 536]		[710, 731]		[696, 732]	

Table 6.11. "Large query" performance comparison

	XS		S		M		L		XL		XXL	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
DB	12680	1310	32305	740	78534	16406	128559	33086	252016	59397	312254	65248
Neo4j	14082	1134	33489	6485	84157	20111	142194	34506	300085	72360	364067	74021
Confidence interval	$[924, 1881]$		$[-1543, 3912]$		$[-1571, 12817]$		$[384, 26886]$		$[21856, 74282]$		$[24613, 79012]$	

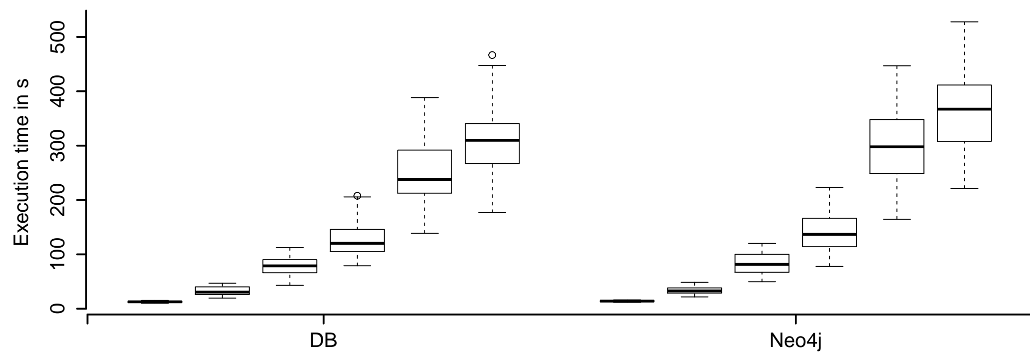

Figure 6.13. Box-and-Whisker plots for the "large query" scenario

Chapter 7

Conclusion

7.1 Summary and Discussion

The thesis provided an overview of the current model repository technologies in the context of the Eclipse Modeling Framework. Five different repositories were investigated and classified with respect to requirements necessary for the MAMBA framework. The thesis showed that many model repository technologies only recreate the default persistence mechanism by adding only a persistence backend. This faces the problem with insufficient memory but do not support collaborative work on models.

At this time, only the Connected Data Objects model repository support the persistence aspect as well as the distributed shared model approach. Since there already exists a model repository with the capabilities that we require in the context of MAMBA, this thesis investigated the use of graph databases by using CDO. We implemented a CDO store based on the Neo4j graph database. Neo4j is not intended to persist large amount of data within one transaction. If models are large enough, they cannot be persisted anymore. Therefore, we proposed an approach to split a large transaction into multiple smaller transactions in order to meet the ability of Neo4j. This split makes it necessary to provide a mechanism for preserving the consistency.

CDO does not provide the object graph to its backend but rather flatten the graph into a list of objects. This makes it difficult to exploit the graph properties of the object graph. In order to exploit the graph properties, it would be necessary to reconstruct the graph which results in expensive insert operations. Furthermore, CDO queries requests only a single object per time from its backend. Because of that, we decided to store the objects by using an index. This is a fast way to look up objects referenced by a unique ID.

The evaluation showed that the developed Neo4jStore with its graph database is slower than the DBStore with a relational database management system. This statement must be considered very carfully. Actually we compared an index with a relational database and not a graph database. As Barmpis and Kolovos [2012] showed, Neo4j seems to be pretty useful in the context of EMF.

7.2 Future Work

There could be some space for improvement of the developed Neo4jStore. For example the optimal chunk size for split transaction must be experimentally investigated. Furthermore, it should be investigated whether the assumption is correct that the recreation of the object graph is useless.

The performance of other CDO stores could be considered, e. g., the OjectivityStore. With the new version of CDO the LissomeStore was introduced. Maybe these CDO stores provides better performances.

The evaluation of the CDO stores showed that, MongoDB is very fast for small models. A mechanism is required that allows to store larger models in the MongoDB. Then, it would be possible that the MongoDB accomplish the best performance.

Finally, there are some improvements for our proposed benchmark. Until now, it does not consider concurrent access of the model repository.

Bibliography

[Barmpis and Kolovos 2012] K. Barmpis and D. S. Kolovos. Comparative Analysis of Data Persistence Technologies for Large-Scale Models. In: *Proceedings of the 15th International Conference on Model Driven Engineering Languages & Systems (MoDELS '12)*. Oct. 2012. (Cited on pages 13–15, and 74)

[Edlich et al. 2010] S. Edlich, A. Friedland, J. Hampe, and B. Brauer. NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken. Hanser Fachbuchverlag, Oct. 2010. (Cited on page 10)

[Espinazo-Pagán et al. 2011] J. Espinazo-Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: a scalable approach for persisting and accessing large models. In: *MoDELS*. Edited by J. Whittle, T. Clark, and T. Kühne. Volume 6981. Lecture Notes in Computer Science. Springer, 2011, pages 77–92. URL: `http://dblp.uni-trier.de/db/conf/models/models2011.html#Espinazo-PaganCM11`. (Cited on pages 1, 2, 13, and 22)

[Fahrmeir et al. 2009] L. Fahrmeir, R. Künstler, I. Pigeot, and G. Tutz. Statistik - Der Weg der Datenanalyse. Springer-Verlag, 2009. (Cited on page 55)

[Frey et al. 2011] S. Frey, A. van Hoorn, R. J. andWilhelm Hasselbring, and B. Kiel. MAMBA: A Measurement Architecture for Model-Based Analysis. Technical report. Department of Computer Science, University of Kiel,Germany, 2011. (Cited on page 1)

[Georges et al. 2007] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. OOPSLA '07. ACM, 2007, pages 57–76. (Cited on pages 51, 53–56)

[Goncalves 2010] A. Goncalves. Beginning Java EE 6 with GlassFish 3. Apresspod Series. Apress, 2010. (Cited on page 12)

[Hedderich and Sachs 2012] J. Hedderich and L. Sachs. Angewandte Statistik - Methodensammlung mit R. Springer Gabler, 2012. (Cited on pages 51–54, and 57)

[Kloos et al. 2012] U. Kloos, N. Martínez, and G. Tullius. Informatics Inside 2012: Reality++ - Tomorrow comes today! Hochschule Reutlingen, 2012. (Cited on pages 29–31)

[Kähler 2010] W.-M. Kähler. Statistische Datenanalyse. Vieweg+Teubner, 2010. (Cited on page 52)

[Lübbe 2012] K. Y. Lübbe. Improving a Transformation of Java Models to KDM. Bachelor's thesis. Kiel University, Sept. 2012. (Cited on page 2)

[MacDonald et al. 2005] A. MacDonald, D. M. Russell, and B. Atchison. Model-driven development within a legacy system: an industry experience report. In: *Australian Software Engineering Conference*. 2005, pages 14–22. (Cited on page 8)

[Mohagheghi 2008] P. Mohagheghi. Evaluating software development methodologies based on their practices and promises. In: *SoMeT*. 2008, pages 14–35. (Cited on pages 1, 7, 8)

[Mohagheghi and Aagedal 2007] P. Mohagheghi and J. Aagedal. Evaluating quality in model-driven engineering. In: *Proceedings of the International Workshop on Modeling in Software Engineering*. MISE '07. Washington, DC, USA: IEEE Computer Society, 2007, pages 6–8. (Cited on page 1)

[*Knowledge Discovery Meta-Model v1.3*] Object Management Group. Knowledge Discovery Meta-Model v1.3. URL: http://www.omg.org/spec/KDM/1.3. (Cited on page 1)

[*Structured Metrics Meta-Model v1.0*] Object Management Group. Structured Metrics Meta-Model v1.0. URL: http://www.omg.org/spec/SMM/1.0. (Cited on page 1)

[Pearson and Hartley 1976] E. Pearson and H. Hartley. Biometrika tables for statisticians. 1. Biometrika Tables for Statisticians. Biometrika Trust, University college, 1976. (Cited on page 56)

[Robinson et al. 2013] I. Robinson, J. Webber, and E. Eifrem. Graph Databases. 1. Aufl. O'Reilly Media, Incorporated, 2013. (Cited on pages 2, 11, 12)

[Staron 2006] M. Staron. Adopting Model Driven Software Development in Industry – A Case Study at Two Companies. In: *Model Driven Engineering Languages and Systems*. Edited by O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio. Volume 4199. Springer Berlin Heidelberg, 2006. Chapter 5, pages 57–72. (Cited on page 7)

[Steinberg et al. 2009] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. EMF: Eclipse Modeling Framework. 2nd. Addison-Wesley, 2009. (Cited on pages 2, 8, 19, and 32)

[*CDO Model Repository Documentation*] E. Stepper. CDO Model Repository Documentation. URL: http://www.eclipse.org/cdo/documentation/. (Cited on page 27)

[*CDO Model Repository Overview*] E. Stepper. CDO Model Repository Overview. last visit: June 3, 2013. URL: http://www.eclipse.org/cdo/documentation/. (Cited on page 27)

[Stepper 2010] E. Stepper. Scale, share and store your models with cdo / dawn. In: Presented on Eclipse Summit Europe 2010, Ludwigsburg/Germany, 2010. (Cited on pages 19, 33, 34)