Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2014
Online at http://www.jot.fm.

Evaluation of Contemporary Graph Databases for Efficient Persistence of Large-Scale Models

Konstantinos Barmpis[a] Dimitrios S. Kolovos[a]

a. Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK
http://www.cs.york.ac.uk/

Abstract Scalability in Model-Driven Engineering (MDE) is often a bottleneck for industrial applications. Industrial scale models need to be persisted in a way that allows for their seamless and efficient manipulation, often by multiple stakeholders simultaneously. This paper compares the conventional and commonly used persistence mechanisms in MDE with novel approaches such as the use of graph-based NoSQL databases; Prototype integrations of Neo4J and OrientDB with EMF are used to compare with relational database, XMI and document-based NoSQL database persistence mechanisms. It also compares and benchmarks two approaches for querying models persisted in graph databases to measure and compare their relative performance in terms of memory usage and execution time.

Keywords scalability, persistence, model-driven engineering

1 Introduction

The popularity and adoption of MDE in industry has increased substantially in the past decade as it provides several benefits compared to traditional software engineering practices, such as improved productivity and reuse [MFM+09], which allow for systems to be built faster and cheaper. However, certain limitations of supporting tools such as poor scalability which prevent wider use of MDE in industry [KPP08, MDBS09] will need to be overcome. Scalability issues arise when large models (of the order of millions of model elements) are used in MDE processes.

When referring to scalability issues in MDE they can be split into the following categories:

1. Model persistence: storage of large models; ability to access and update such models with low memory footprint and fast execution time.
2. Model querying and transformation: ability to perform intensive and complex queries and transformations on large models with fast execution time.

Konstantinos Barmpis, Dimitrios S. Kolovos. *Evaluation of Contemporary Graph Databases for Efficient Persistence of Large-Scale Models*. In Journal of Object Technology, vol. 13, no. 3, 2014, pages 3:1–26. doi:10.5381/jot.2014.13.3.a3

3. Collaborative work: multiple developers being able to query, modify and version control large-scale shared models in a non-invasive manner.

Previous works have suggested using relational and document NoSQL databases to improve performance and memory efficiency when working with large-scale models. This paper contributes to the study of scalable techniques for large-scale model persistence and querying by reporting on the results obtained by exploring two graph-based NoSQL databases (OrientDB and Neo4J), and by providing a direct comparison with previously proposed persistence mechanisms. This paper is an extended version of [BK12] with further analysis of the databases presented in Section 5 and results in Section 7. This work is used as the foundation for [BK13], which integrates scalable persistence with reliable versioning. The remainder of the paper is organized as follows. Section 2 introduces MDE and NoSQL databases. Section 3 discusses other projects aiming at providing scalable model persistence. Section 4 introduces the Grabats query used for evaluating the technologies. Section 5 presents the design and implementation of two further prototypes for scalable model persistence based on the OrientDB and Neo4J graph-based NoSQL databases. In section 6 we discuss two approaches for navigation and querying of models stored in such databases. In section 7 the produced prototypes are compared with existing solutions in terms of performance. Finally, section 8 discusses the application of these results and identifies interesting directions for further work.

2 Background

This section discusses the core concepts related to models, Model Driven Engineering and NoSQL databases that will be used in the remainder of the paper.

2.1 Model-Driven Engineering

Model Driven Engineering is an approach to software development that elevates *models* to first class artefacts of the software engineering process. In MDE, models are living entities used to describe a system and (partly) automate its implementation through automated transformation to lower-level products. In order for models to be amenable to automated processing, they must be defined in terms of rigorously specified modeling languages (metamodels). The Eclipse Modeling Framework[1] (EMF) is one of the most widely-used frameworks that facilitate the definition and instantiation of metamodels, and a pragmatic implementation of the OMG Essential Meta Object Facility (EMOF) standard.

In EMF, metamodels are defined using the Ecore metamodeling language, a high level overview of which is illustrated in Figure 1. In Ecore, domain concepts are represented using *EClass*es. *EClass*es are organized in *EPackage*s and each *EClass* can contain *EReference*s to other *EClass*es in the metamodel and *EAttribute*s, which are used to define the primitive features of instances of the *EClass*. Ecore also provides mechanisms for defining primitive types, enumerations, inheritance between *EClass*es and operation signatures (but not implementations). EMF metamodels can be instantiated both reflectively, and through code generated through a 2-stage transformation. The code generation process involves a model-to-model transformation where the Ecore metamodel is transformed into an intermediate platform-specific model (*GenModel*) that enables engineers to define low-level details of the metamodel

[1] http://www.eclipse.org/emf

implementation, and a model-to-text transformation from the intermediate *GenModel* to Java. Under both the reflective and the generative approach, at runtime EMF models comprise of one or more *Resources* containing nested model elements (*EObject*s) that conform to the *EClass*es of the Ecore metamodel.

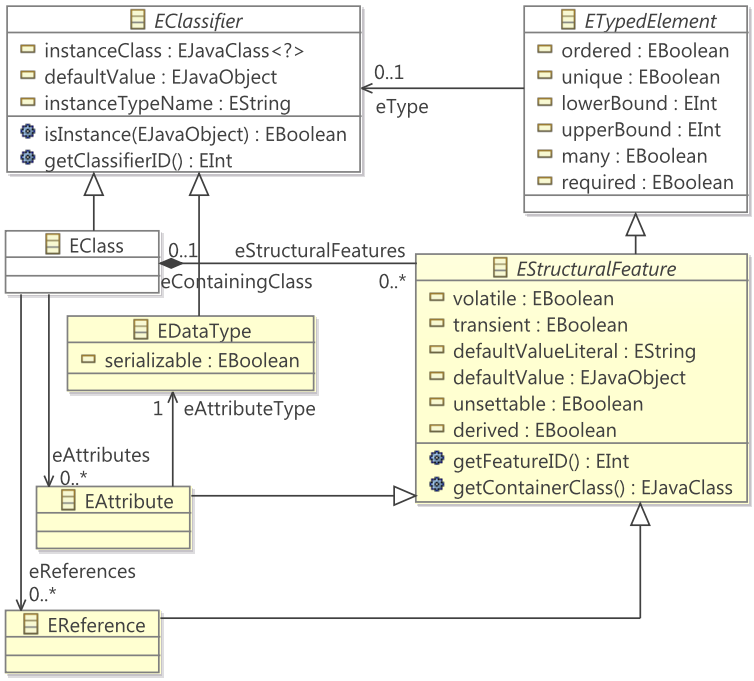


Figure 1 – Simplified Diagram of the Ecore Metamodeling Language

By default, models in EMF are stored in a standard XML-based representation called XML Metadata Interchange (XMI) that is an OMG-standardized format designed to enhance tool-interoperability. As XMI is an XML-based format, models stored in single XMI files cannot be partially loaded and as such, loading an XMI-based model requires reading this entire document using a SAX parser, and converting it into an in-memory object graph that conforms to the respective Ecore metamodel. As such, XMI scales poorly for large models both in terms of time needed for upfront parsing and resources needed to maintain the entire object graph in memory (the performance issues of XMI are further illustrated in Section 7). To address these limitations of XMI, persisting models in relational databases has been proposed.

Examples of such approaches include the Connected Data Objects (CDO)[2] project and Teneo-Hibernate[3]. In this class of approaches, an Ecore metamodel is used to derive a relational schema as well as an object-oriented API that hides the underlying database and enables developers to interact with models that conform to the Ecore metamodel at a high level of abstraction. Such approaches eliminate the initial overhead of loading the entire model in memory by providing support for partial and on-demand loading of subsets of model elements. However, due to the nature of relational databases, such approaches, while better than XMI, are still largely inefficient, as demonstrated in Section 4. Due to the highly interconnected nature of most

[2]http://www.eclipse.org/CDO/
[3]http://wiki.eclipse.org/Teneo/Hibernate

models, complex queries require multiple expensive table joins to be executed and hence do not scale well for large models. Even though Teneo-Hibernate attempts to minimize the number of tables generated (all subclasses of an *EClass* are in the same table as the *EClass* itself, resulting in a fraction of the tables otherwise required if a separate table was created for each *EClass*, the fact that the database consists of sparsely populated data results in increased insertion and query time as demonstrated in the sequel.

To overcome the limitations of relational databases for scalable model persistence, recent work [PCM11] has proposed using a NoSQL database instead. In the following paragraphs we provide a discussion on NoSQL databases and their application for scalable model persistence.

2.2 NoSQL Databases

The NoSQL (Not Only SQL) movement is a contemporary approach to data persistence using novel, typically non-relational, storage approaches. NoSQL databases provide flexibility and performance as they are not limited by the traditional relational approach to data storage [Sto10]. Each type of NoSQL database is tailored for storing a particular type of data and the technology does not force the data to be limited by the relational model but attempts to make the database (as much as is feasible) compatible with the data it wishes to store [Ore10]. The NoSQL movement itself has become popular due to large widely known and successful companies creating database storage implementations for their services, all of which do not follow the relational model. Such companies include for example Amazon (Dynamo database [DHJ+07]), Google (Bigtable database [CDG+08]) and Facebook (Cassandra database [LM10]).

There are four widely accepted types of NoSQL databases, which use distinct approaches in tackling data persistence, three of which are described by [PPS11] and a fourth, more contemporary one, that is of increasing popularity:

1. Key-value stores consist of keys and their corresponding values, which allows for data to be stored in a schema-less way. This allows for search of millions of values in a fraction of the time needed by relational databases. Inspired by databases such as Amazon's Dynamo, such stores are tailored for handling terabytes of distributed key-value data.
2. Tabular stores (or Bigtable stores - named after the Google database) consist of tables which can have a different schema for each row. It can be seen as each row having one large extensible column containing the data. Such stores aim at extending the classical relational database idea by allowing for sparsely populated tables to be handled elegantly – as opposed to needing a large amount of null fields in a relational database, which scales very poorly when the number of columns becomes increasingly large. Widely used examples of such stores are Bigtable [CDG+08] and Hbase [Hba12].
3. Document databases consist of a set of documents (possibly nested), each of which contains fields of data in a standard format like XML or Json. They allow for data to be structured in a schema-less way as such collections. Popular examples are MongoDB [Mon12] and OrientDB [Ori12].
4. Graph databases consist of a set of graph nodes linked together by edges (hence providing index-free adjacency of nodes). Each node contains fields of data and querying the store commonly uses efficient mathematical graph-traversal algorithms to achieve performance; As such, these databases are optimized for

traversal of highly interconnected data. Examples of such stores are Neo4J [Neo12] and the graph layer (OGraphDatabase) of OrientDB [Ori12].

NoSQL databases have a loosely defined set of characteristics / properties [Cat11]:

- They scale horizontally by having the ability to dynamically adapt to the addition of new servers.
- Data replication and distribution over multiple servers is used, for coping with failure and achieving eventual consistency.
- Eventual consistency; a weaker form of concurrency than ACID (Atomicity, Consistency, Isolation, Durability) transactions, which does not lock a piece of data when it is being accessed for a write operation but uses data replication over multiple servers to cope with conflicts. Each database will implement this in a different way and will allow the administrator to alter configurations making it either closer to ACID or increasing the availability of the store.
- Simple interfaces for searching the data and calling procedures.
- Use of distributed indexes to store key data values for efficient searching.
- Ability to add new fields can be added to records dynamically in a lightweight fashion.

The CAP theorem defines this approach and states that a (NoSQL) database can choose to strengthen only two of the three principles: consistency availability and partition tolerance, and has to (necessarily) sacrifice the third. Popular NoSQL stores chose to sacrifice consistency; BASE (Basically Available, Soft-state, Eventually consistent) defines this approach.

NoSQL stores are seen to have the following limitations [Lea10]:

1. Lack of a standard querying language (such as SQL) results in the database administrator or the database creator having to manually create a form of querying.
2. Lack of ACID transactions results in skepticism from industry, where sensitive data may be stored.
3. Being a novel technology causes lack of trust by large businesses which can fall back on reliable SQL databases which offer widely used support, management and other tools.

2.2.1 Graph Databases

As this paper presents an approach using graph-based NoSQL databases, we will present them in more detail here.

Figure 2 describes the basic terminology used for property graphs, like the ones used in graph databases. As such, graph stores describe their constructs in this way, with nodes containing properties and relationships between them. Below we go into more depth about two specific graph databases, Neo4J and (the graph layer of) OrientDB.

Neo4J Neo4J is a popular, commercial Graph Database released under the GNU Public License (GPL) and the Affero GNU Public License (AGPL) licenses. Neo4J is implemented in the Java programming language and provides a programmatic way to insert and query embedded graph databases. Its core constructs are *Nodes* (which contains an arbitrary number of properties, which can be dynamically added and removed at will) and *Relationships*, whereby *Node* represents a mathematical graph node and *Relationship* an edge between two nodes.

[4]http://www.infoq.com/articles/graph-nosql-neo4j

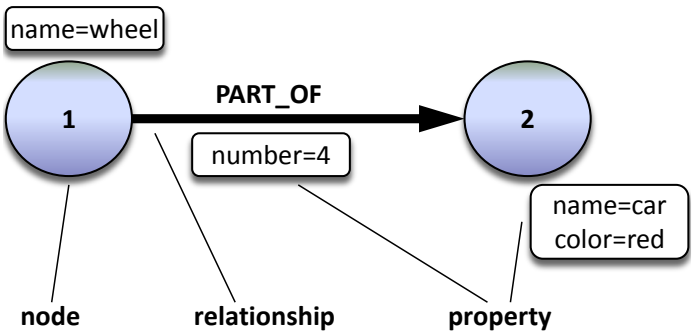


Figure 2 – Diagram of property graph terminology[4]

OrientDB OrientDB is a novel document-store database released under the Apache 2 License. It is also implemented using the Java programming language and provides a programmatic way to insert and query from a document database in Java. OrientDB also has a graph layer which allows for documents to have edges between them (edges are backed by documents themselves), emulating the property of index-free adjacency of documents and hence effectively being a graph database. Its core constructs are *ODocuments* (which can contain an arbitrary number of properties that can be dynamically added and removed).

3 Related Work

While the approach of persisting large models in NoSQL is a novel concept, it has been already done, mainly by use of document-based databases. Below we briefly present a popular model repository, in which a NoSQL store (MongoDB document store) can be used to persist models. We also present Morsa, the first published work of using a NoSQL store to tackle scalable model persistence (also MongoDB) and its prototype tool.

3.1 The Connected Data Objects Repository (CDO)

CDO allows users to store and access models in repositories supported by a range of back-end stores. CDO's API is an extension of EMF's and allows for a seamless use of a remote store for accessing and manipulating models. CDO supports multiple different back-ends such as relational databases and non-relational stores such as the MongoDB NoSQL store.

Object-Relational Mapping CDO handles *EObject*s as *CDOObject*s that extend the *EObject* class by adding CDO-specific metadata. To store an EMF model on CDO, there are three main paths to pursue[5]: The first is to migrate a Resource (for example an *XMIResource*) to a *CDOResource* (by copying all its contents to a new *CDOResource*). The second is to use a GenModel to create *CDOObjectImpl* objects by migrating the *.gen-model* file using the CDO Model Migrator. The third is to use *DynamicCDOObjectImpl* that result from new dynamic model elements added to a CDO session's package registry. The model files used by CDO can be annotated

[5]http://wiki.eclipse.org/Preparing_EMF_Models_for_CDO

(such as with JPA – *EAnnotations*) in order to provide a tool for customization in the storage of the model; such annotations are not directly needed by CDO and are only useful if the back-end store supports them (such as Teneo/Hibernate, for example).

Furthermore, it is worth noting that CDO's architecture, which is directly based on such mapping, limits how much it can benefit from using other technologies as it fundamentally represents model and metamodel data by means of tables.

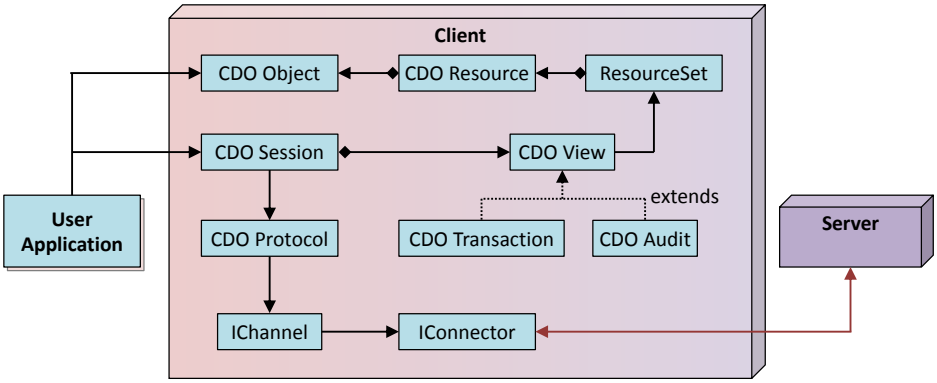


Figure 3 – CDO Client high-level architecture, adapted from the CDO Wiki[6]

Client From the client-side, the regular EMF API can be directly used after a connection (session) has been established, but for using advanced CDO-specific functionality (such as *CDOView* that allows queries directly to the CDO store, or *CDOTransaction* that allows for savepoints and rollbacks), additional dependencies to CDO have to be included. Furthermore, a native CDO User Interface (UI) is provided for accessing, manipulating and querying models stored in the repository. This client architecture is seen in Figure 3.

Server On the server-side, this repository allows any form of storage to be easily plugged in (such as a MongoDB NoSQL database). It uses a proprietary model-based version control system (Audit Views) and supports collaborative development. More information about model repositories and a comparison thereof with file-based repositories of models can be found in [BK13].

3.2 Morsa: NoSQL Model Persistence Prototype

Morsa [PCM11] is a prototype that attempts to address the issue of scalable model persistence using a document store NoSQL database (MongoDB) to store EMF models as collections of documents. Morsa stores one model element per document, with its attributes stored as key-value pairs, alongside its other metadata (such as a reference to its *EClass*). Metamodel elements are stored in a similar fashion to model elements and are also represented as entries in an index document that maps each model or metamodel URI (the unique identifier of a model or metamodel element in the store) into an array of references to the documents that represent its root objects. A high level overview of the architecture of Morsa is displayed in Figure 4 by [PCM11].

Morsa uses a load-on-demand mechanism which relies on an object cache that holds loaded model objects. This cache is managed by a configurable cache replace-

[6]http://wiki.eclipse.org/CDO

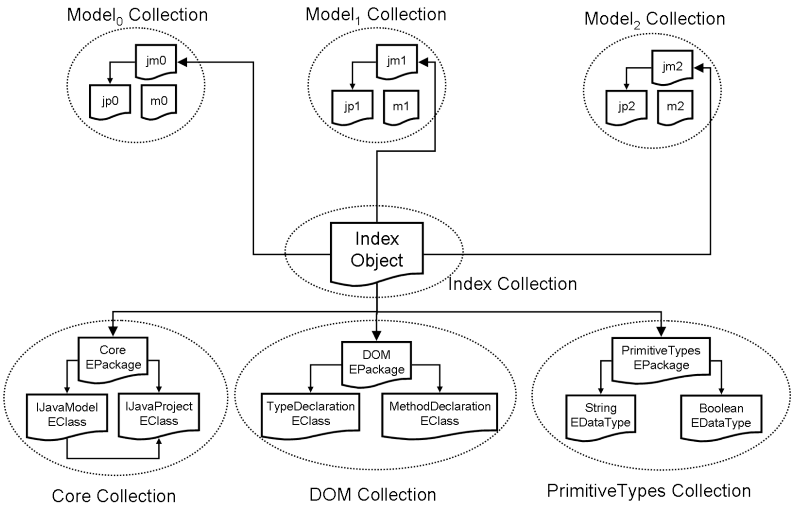


Figure 4 – Persistence back-end structure excerpt for Morsa

ment policy that chooses which objects must be unloaded from the client memory (should the cache be deemed full by the active configuration). While this attempts to use an effective storage technique and succeeds in improving upon the current paradigms, due to using a Document Store database the *EReference*s (which are serialized as document references) are stored inefficiently, which hampers insertion and query speed, as models tend to be densely interconnected with numerous references between them. Nevertheless, the discussions on the various caching techniques and cache replacement policies, as well as the different loading strategies are very effective in conveying the large number of configurations possible in a single back-end persistence example, and how optimizing the storage of different sizes and types of models can be extremely complex. Hence any solution aiming at tackling this challenge needs to be aware of these issues and experiment on the optimal way to handle them in its specific context.

4 The Grabats 2009 Case Study and Query

To obtain meaningful evaluation results, we have evaluated all solutions using large-scale models extracted by reverse engineering existing Java code. For this purpose, we have used the updated version of the JDTAST metamodel used in the *SharenGo Java Legacy Reverse-Engineering* MoDisco use case[7], presented in the Grabats 2009 contest [Gra12] described below, as well as the five models also provided in the contest.

A subset of the Java JDTAST metamodel is presented in Figure 5. In this figure, there are *TypeDeclaration*s that are used to define Java classes and interfaces, *MethodDeclaration*s that are used to define Java methods (in classes or interfaces, for example) and *Modifier*s that are used to define Java modifiers (like static or synchronized) for Java classes or Java methods.

The Grabats 2009 contest comprised several tasks, including the case study used in this paper for benchmarking different model querying and pattern detection tech-

[7] http://www.eclipse.org/gmt/MoDisco/useCases/JavaLegacyRE/

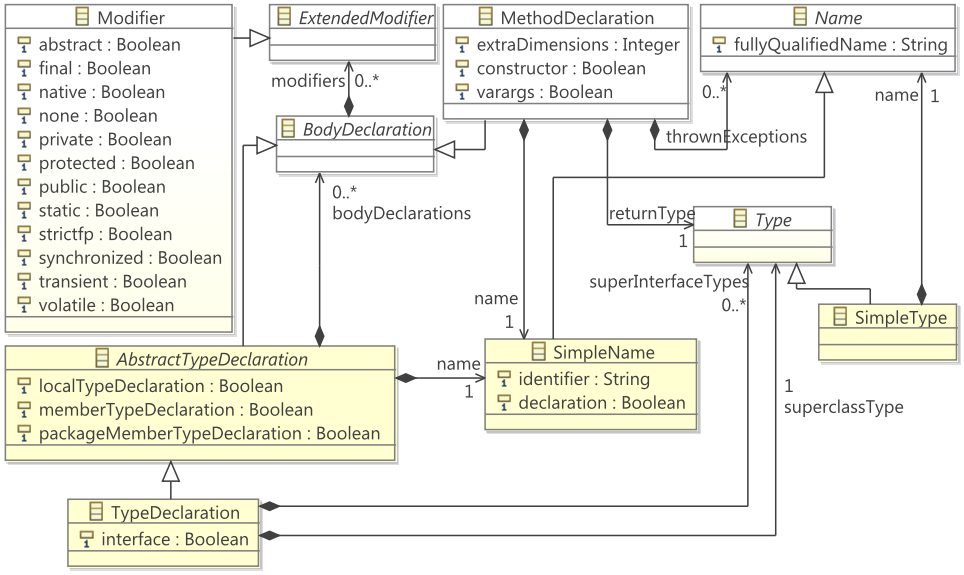


Figure 5 – Small subset of the Java JDTAST metamodel

nologies. More specifically, task 1 of this case study is performed, using all of the case studies' models, set0 – set4 (which represent progressively larger models, from one with 70447 model elements (set0) to one with 4961779 model elements (set4)), all of which conform to the JDTAST metamodel.

These models are injected into the persistence technologies used in the benchmark (insertion benchmark) and then queried using the Grabats 2009 task 1 query (query benchmark) [SJ09]. This query requests all instances of *TypeDeclaration* elements which declare at least one *MethodDeclaration* that has static and public modifiers and the declared type being its returning type (i.e. singleton candidates).

In the following sections we use the JDTAST metamodel as a running example to demonstrate our approach for persisting large-scale models in the Neo4J and OrientDB graph databases.

5 Persisting and Querying Large-Scale Models using Graph Databases

As discussed above, NoSQL databases have been shown to be a promising alternative that overcomes some of the limitations of relational databases for persistence of large-scale models, briefly summarized in Section 2.1. Extending the study of the suitability of NoSQL databases for persistence of large-scale models, in this work prototype model stores based on Neo4J [Neo12] and OrientDB [Ori12] have been created, and their efficiency has been compared against the default EMF XMI text store and a relational database (using CDO with its default H2[8] store as well as a MySQL[9] store, to integrate with EMF). This section discusses the design and implementation of the

[8] http://www.h2database.com/html/main.html
[9] http://www.mysql.com/

two model stores; to our knowledge, this is the first time graph databases are used to persist large models.

Due to the highly-interconnected nature of typical models, key-value stores as well as tabular NoSQL data-stores were not considered, as they target a different class of problems (as explained above in Section 2.2). Hence the decision was made to experiment with a document based (and hybrid-graph) database (OrientDB) and a pure graph database (Neo4J). After initial trials, the document layer of OrientDB lagged behind the graph layer (as can be expected with the nature of the data being stored) so the focus shifted on comparing two graph databases. The rationale behind choosing these technologies was that Neo4J is a particularly popular, stable and widespread graph database while OrientDB not only provides a document layer and a graph layer, but also has a flexible license, which Neo4J does not, as detailed in their respective subsections 5.1 and 5.2 below.

The Neo4J and OrientDB stores attempt to solve the aforementioned scalability issues using graph databases to store large models. As such stores have index-free adjacency of nodes, we anticipate that retrieving subgraphs or querying a model will scale well. The main differences between the two prototypes lie in the fact that OrientDB's core storage is in documents (and uses a graph layer to handle the data as a graph) while Neo4J's core storage is as a graph. In the following sections we present our approaches for persisting and querying models that conform to Ecore metamodels using Neo4J and OrientDB databases, and we then evaluate the performance of our two prototypes against XMI and CDO.

5.1 Neo4J

In our prototype, a Neo4J-based model store consists of the following:

- *Node*s representing model elements in the model stored. These nodes contain as properties all of the attributes of that element (as defined by the *EClass* it is an instance of) that are set.

- *Relationship*s from model element nodes to other model element nodes. These represent the *EReference*s of the model element to other model elements.

- *Node*s representing *EClass*es of the metamodel(s) the models stored are instances of. These nodes only have an *id* property denoting the unique identifier (URI) of the metamodel they belong to, followed by their name, ie: `org.amma.dsl.jdt.core/IJavaElement` is the id of the *EClass IJavaElement* in the `org.amma.dsl.jdt.core` Ecore metamodel. These lightweight nodes are used to speed up querying by providing references to model elements that are instances of this *EClass* (*ofType* reference) as *EClass*es that inherit from it (*ofKind* reference) – as such types of queries are very common in model management programs (e.g. model transformations, code generators etc.). This is the only metamodel information stored in the database, as explained below.

- An index containing the ids of the *EClass*es and their appropriate location in the database. This allows typical queries (such as the Grabats query described above) to use an indexed *EClass* as a starting point, in order to find all model elements of a specific type, and then navigate the graph to return the required results.

The above data contains all of the information required to load a model and evaluate any EMF query, provided that the metamodel(s) of the model is also available (e.g. registered to the EMF metamodel registry[10]) during the evaluation of the query, as detailed metamodel information is not saved in the database (only the model information is stored in full). This is due to the fact that metamodels are typically small and sufficiently fast to navigate using the default EMF API. Thus any action that may require use of such metamodel data, like querying whether this element can have a certain property (as it may be unset, and therefore not stored in the database) or whether a reference is a containment one for example (as the database only stores the reference's name) will need to access the metamodel (e.g. through the EMF registry), retrieve the *EClass* in question and extract this information from there. Note that the database supports querying of a model, insertion of a new model (from a file-based EMF model) as well as updating a model (adding or removing elements or properties).

Figure 6 shows how a model conforming to the Java metamodel described above is stored in Neo4J. *EClass*es only store their full name (including their *EPackage*) and have *ofType* and *ofKind* relationships to their instances (we note that in EMF such relationships are not references but results of applying the *.eClass()* operation on the model element). Model elements store all their properties and relationships to other model elements (as well as to their *EClass* and superclass(es)). Note that even though all Neo4J relationships are bi-directional, they have a starting and an end node so can be treated in the same way as references internally. Opposite references are similarly created (but with the starting and end node reversed), linked to one another internally, and treated as usual.

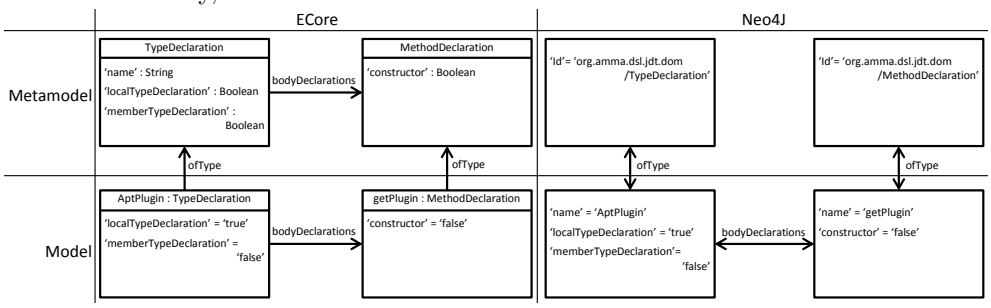


Figure 6 – Example high-level mapping from Ecore to Neo4J

5.1.1 Transactions and I/O

The default mechanism for querying the database uses ACID transactions in a similar manner to SQL databases, and any number of operations can be carried out per transaction (such as creation of a node, creation of an edge, creation of a property of a node). The database uses the relevant operating system's Memory Mapping for its I/O (MMIO) in order to increase performance. Hence if a transaction has too many operations in it (more than the allocated MMIO (or even the maximum Java heap) can handle) then the performance of the transaction will suffer. On the other hand if transactions perform too few operations then there will be a lot more transactions needed for performing a task and hence the overall run time of this task will be longer. Hence an equilibrium needs to be found around the size of transactions and dependent on the total memory allocated to the process. A further balance needs to be found

[10] http://www.eclipse.org/epsilon/doc/articles/epackage-registry-view/

between the Java heap and the MMIO which will add up to be the total memory used by the process. Even further the memory available for inserts is hindered by the XMI resource being loaded (which uses a considerable amount of memory) and for large models needs to be taken into consideration. Empirical tests have shown that using:

$$\frac{Runtime.getRuntime().maxMemory() - xmiResourceMemory}{15000}$$

to be the number of operations per transaction works efficiently. The value of *xmiResourceMemory* is calculated dynamically after the relevant file(s) have been loaded into memory by using the before and after delta in memory consumption. This number is obtained by considering information given in the documentation[11] and by testing various values for the denominator (aka the size of each object to be committed in the transaction). The tests performed demonstrated that other values were less performant than this; the value assumes that, on average, each object in the transaction will be of size 15000 bytes. If this value is not reasonable for the specific context, either it is too large (for example someone storing a large model consisting of elements containing only a single boolean attribute and a single reference), or it is too small (for example someone storing a large model consisting of elements containing large strings inside them and multiple references to one another) it should be altered by the administrators.

Table 1 – Configuration used for MMIO for Neo4J

key	value
nodestore.db.mapped_memory	$3 \times x +$ "M"
relationshipstore.db.mapped_memory	$14 \times x +$ "M"
propertystore.db.mapped_memory	$x +$ "M"
propertystore.db.strings.mapped_memory	$2 \times x +$ "M"
propertystore.db.arrays.mapped_memory	$x +$ "M"

Concerning the Heap vs. MMIO the following ratios described in Table 1 works efficiently as well, using:

$$x = \frac{Runtime.getRuntime().maxMemory()}{60 \times 1000000};$$

whereby x is a constant calculated from the maximum heap memory given to the JVM, config is the map of configuration options Neo4J uses for MMIO for each of its files and "M" denotes to Neo4J that values are in megabytes. In this case, the values reflect considering the suggestions given in the Neo4J documentation[12], tailored for the specific machine the server is running on (using the current JVM heap), and to our expected need for high-efficiency traversal of the store. Different sets of values were tested (such as the exact ones shown in the documentation[12]) and displayed poorer performance in our test cases (such as in the Grabats query example).

5.2 OrientDB

An OrientDB-based model store consists of the following:

[11] http://docs.neo4j.org/chunked/stable/performance-guide.html#_configuring_neo4j
[12] http://docs.neo4j.org/chunked/milestone/configuration-io-examples.html

- *ODocument*s representing (the nodes of the) model elements in the model stored. These nodes contain as fields all of the attributes of that element (as defined by both its *EClass* and its superclass(es)) that are set.

- *ODocument*s representing (the references (edges)) from model element nodes to other model element nodes. These represent the *EReference*s of the model element to other model elements.

- *ODocument*s representing metamodel *EClass*es of the metamodel the model stored adheres to. These nodes have an 'id' field denoting the URI of their package followed by their Name, ie: `org.amma.dsl.jdt.core/IJavaElement` is the id of the *EClass* IJavaElement in the *EPackage*: `org.amma.dsl.jdt.core`. They also have a 'class' and a 'superclass' field which contain lists of the database ids of their *ofType* and *ofKind* elements (the ones they are a class or a superclass of).These lightweight class nodes are used to speed up queries by providing direct links (in the form of lists) to model elements that have this *EClass* as their class (*ofType* reference) or superclass (*ofKind* reference). This is the only metamodel information actually stored in the database, as explained below. The reason references are not used (like in Neo4J) is that they are too heavyweight (in a similar manner to how the ones in Morsa are), and empirical evidence shows that execution time using references instead of lists is much slower. We acknowledge that keeping a dynamic list of model instances in the *EClass*es has a significant effect in model alteration performance but in the absence of a more efficient way to handle such information in OrientDB (such as the bi-directional references in Neo4J), we chose to use this in order to significantly enhance query performance (for example when a model transformation requires access to all instances of a class to work with).

- An index containing the ids of the *EClass*es and their appropriate location in the database. This allows a typical query to use an indexed *EClass* as starting point, in order to find all of the model elements of a specific type, and will then navigate the graph to return the appropriate results.

The above data contains all of the information required to load a model and provide answers to any EMF query, provided that the metamodel(s) of the model is registered to the EMF registry before any actions can be performed, as detailed metamodel information is not saved in the database. Thus any action which may require use of such metamodel data will go to the EMF registry, retrieve the *EClass* in question and get this information from there. Note that the database supports querying of a model, insertion of a new model (from an Ecore - XMI document) as well as updating a model (adding or removing elements or properties).

Figure 7 shows how a model conforming to the Java metamodel described above is stored in OrientDB. *EClass*es only store their full name (including their *EPackage*) and lists of the id's of their model instances (to whom they are *ofType* or *ofKind*). Model elements store all their properties (as well as the id of their *EClass* and a list of the id's of their possible superclass(es)) and relationships to other model instances. We note that in EMF such relationships are not actual references but results of applying the *.eClass()* operation on the model element.

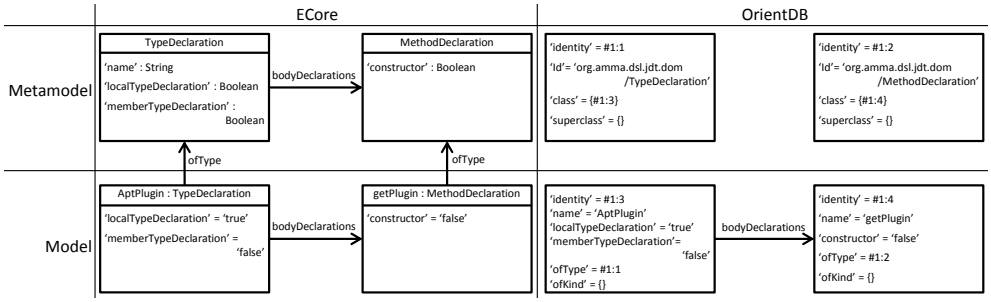


Figure 7 – Example high-level mapping from Ecore to OrientDB

5.2.1 Transactions and I/O

The default mechanism for querying the database does not use transactions but internal mechanisms to update the database when appropriate and ensure eventual consistency. The database uses the relevant operating system's Memory Mapping for its I/O (MMIO) in order to increase performance. In this paradigm the MMIO is not embedded in the Java heap hence the programmer needs to keep enough RAM available after executing the program for the MMIO in order not to overflow the system and make it go into swap. Like in the previous example, a balance needs to be found between the Java heap and the MMIO which will add up to be the total memory used by the process. Empirical tests have shown that using the default configuration for 64 bit systems works efficiently; this configuration sets the MMIO to:

$$(maxOsMemory - maxProcessHeapMemory) \times 50\%$$

whereby *maxOsMemory* is the maximum memory used by the operating system and *maxProcessHeapMemory* is the maximum heap given to the OrientDB server. In some situations (such as if running OrientDB on a dedicated server) using $\times 85\%$ instead of the above $\times 50\%$ is beneficial, but since the tests we ran on a standard desktop computer this was not possible.

6 Model Querying

Sections 5.1 and 5.2 have presented two graph-based NoSQL solutions proposed to tackle the problem of persisting large-scale models. This section will discuss two distinct ways in which persisted models can be queried, as well as their benefits and drawbacks. It is worth noting that the current implementation does not support any other forms of non-native querying other than the Epsilon (EOL) layer seen below.

6.1 Native Querying

The default way of querying these stores is using their native API. In both the Neo4J and the OrientDB examples, this involves (in most cases) using the index discussed above as a starting point of the query, and then navigating the object graph through its relationships (which represent EMF references) in order to calculate the required results. This method has the important advantage of being the most efficient way to retrieve data from the databases. Nevertheless, it also demonstrates certain shortcomings which should be considered:

- *Query Conciseness* Native queries can be particularly verbose and, consequently, difficult to write, understand and maintain. For example the equivalent of the query presented in Listing 2 spans over 200 lines of Java code (without comments) when expressed using the native Neo4J API.
- *Query Abstraction Level* Native queries are bound to the specific technology used; they have to be engineered for that technology and cannot be used for a different back-end without substantial alteration in most cases. Table 2 summarizes several key methods available in the Neo4J API and Listing 1 shows some of the code used to implement the Grabats query in Neo4J. Line 1 iterates through all of the outgoing references of the *typeDeclaration* of relationship type *bodyDeclaration*s. Line 3 keeps only the *methodDeclaration*s from these relationships. Lines 7 to 11 get the *name* of the *methodDeclaration*. Line 12 iterates through all of the outgoing references of the *methodDeclaration* of relationship type *modifier* (in order to find if the *methodDeclaration* has public and static *modifier*s). A similar approach is used in OrientDB, omitted in the interest of space.

Table 2 – Example methods available in the Neo4J API

Method	Return Type	Description of Functionality
AbstractGraphDatabase .getNodeById(long id)	Node	Returns the node with id 'id' from the AbstractGraphDatabase
PropertyContainer .getProperty(String prop)	Object	Returns the value of the property 'prop' in the PropertyContainer (for example in a *Node*)
Node.getRelationships (Direction dir, RelationshipType rType)	Iterable<Relationship>	Returns an iterable with all of the relationships of type 'rType' with direction 'dir' with respect to the Node
Relationship.getEndNode()	Node	Returns the node at the end of the Relationship (all relationships have a start and an end node)

```
   ...
1  for (Relationship outEdge : typeDeclaration.getRelationships(
       Direction.OUTGOING, new RelationshipUtil().
       getNewRelationshipType("bodyDeclarations"))) {
2    Node methodDeclaration = outEdge.getEndNode();
3    if (new MetamodelUtils().isOfType(methodDeclaration,new
         MetamodelUtils().eClassNSURI(methodDeclarationClass))) {
4      boolean isPublic;
5      boolean isStatic;
6      String currMethodName;
7      for (Relationship methodDeclarationOutEdge :
           methodDeclaration.getRelationships(Direction.OUTGOING,
```

```
        new RelationshipUtil().getNewRelationshipType("name")))
          {
 8      Node name = methodDeclarationOutEdge.getEndNode();
 9      currMethodName = name.getProperty("fullyQualifiedName").
           toString();
10      //break;
11    }
12    for (Relationship methodDeclarationOutEdge :
         methodDeclaration.getRelationships(Direction.OUTGOING,
         new RelationshipUtil().getNewRelationshipType("
         modifiers"))) {
...
```

Listing 1 – Code excerpt for the Grabats query implemented in Neo4J

6.2 Back-end independent navigation and querying

A common way to access and query models (such as the ones stored in Neo4J or OrientDB, shown above) is using commonly agreed upon interfaces such as the EMF *Resource* and using a higher-level query language that is independent of the persistence mechanism. Examples of such languages include OCL and EOL [KPP06] (from the Epsilon [Ric09] platform), which abstract over concrete model representation and persistence technologies using the *OCL pivot metamodel* [Edw11] and *Epsilon Model Connectivity* [Dim08] layer respectively. In this section we illustrate using the facilities provided by Epsilon/EOL to support back-end independent querying – but in principle a similar approach could also apply to OCL. We avoid presenting any OCL queries as they only support EMF-based models (using the EMF Resource interface to go from their persisted form to an in memory representation), a feature that our current prototypes do not support. Our aim here is to measure the impact of introducing a higher-level query language in terms of conciseness and performance.

Epsilon The Epsilon platform [Ric09] is an extensible family of languages for common model management tasks and includes tailored languages for tasks such as model-to-text transformation (EGL), model-to-model transformation (ETL), model refactoring (EWL), comparison (ECL), validation (EVL), migration (Flock), merging (EML) and pattern matching (EPL). All task-specific languages in Epsilon build on top of a core expression language – the Epsilon Object Language (EOL) – to eliminate duplication and enhance consistency.

As seen in Figure 8, EOL – and as such all languages that build on top of it – is not bound to a particular metamodeling architecture or model persistence technology. Instead, an intermediate layer – the Epsilon Model Connectivity layer – has been introduced to allow for seamless integration of any modeling back-end, using a driver-based approach where integration with a particular modeling technology is achieved by implementing a *driver* that conforms to a Java interface (*IModel*) provided by EMC. Table 3 displays some of the important methods in this interface and a short explanation of their functionality. For a more detailed discussion on EMC and the *IModel* interface, the reader can refer to Chapter 3 of [Dim08].

To enable querying models persisted in the graph databases discussed in this paper using EOL, we have implemented two EMC *drivers*: one for Neo4J and one for OrientDB. Listing 2 shows how using the two new EMC drivers, the Grabats

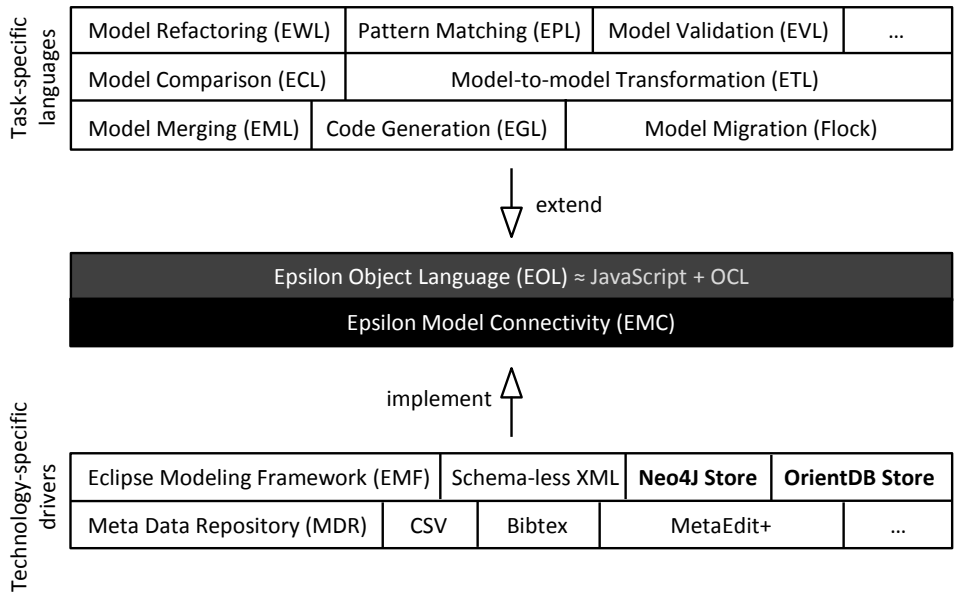


Figure 8 – The Epsilon Model Connectivity Layer

query – which as discussed above, spanned more than 200 lines of Java code – can be expressed using a few lines of, arguably more understandable and easier to maintain, EOL code.

In the following section, we use these drivers to evaluate the impact of using EOL as a higher-level query language in terms of performance.

```
1  TypeDeclaration.all
2    .collect(td|td.bodyDeclarations
3      .select(md:MethodDeclaration|
4        md.modifiers.exists(mod:Modifier|mod.public=="true")
5        and md.modifiers.exists(mod:Modifier|mod.static=="true")
6        and md.returnType.isTypeOf(SimpleType)
7        and md.returnType.name.fullyQualifiedName ==
8          td.name.fullyQualifiedName
9      )
10   )
11   .flatten()
12 );
```

Listing 2 – The Grabats 2009 query expressed in EOL

7 Evaluation

In this section, XMI, Teneo/Hibernate using a MySQL server, CDO (using its default H2 SQL database as well as with a MySQL server) and the two prototypes implemented in this work are compared to assess their performance and efficiency in terms of memory use.

Table 3 – Subset of Methods defined in the IModel interface

Method	Return Type	Summary of Functionality
allContents()	Collection<?>	Returns a collection containing all of the objects contained in the model.
getAllOfType(String type)	Collection<?>	Returns a collection containing all of the objects whose *EClass* is 'type'
getAllOfKind(String type)	Collection<?>	Returns a collection containing all of the objects whose superclass is 'type'
getTypeOf(Object instance)	Object	Returns the *EClass* of the 'instance' element
isOfType(Object instance, String type)	boolean	Returns True iff the 'instance' object has *EClass* 'type' NB: throws EolModelElementTypeNotFoundException if the *EClass* 'type' does not exist
knowsAboutProperty (Object instance, String property)	boolean	Returns True iff the 'instance' object can have the property 'property'
getPropertyGetter()	IPropertyGetter	Returns the IPropertyGetter used by this model. An IPropertyGetter is a class which defines how properties and references are found in the persisted model and has to also be implemented.
getPropertySetter()	IPropertySetter	Returns the IPropertySetter used by this model. An IPropertySetter is a class which defines how properties and references are set (updated or initialized) in the persisted model and has to also be implemented.

7.1 Execution Environment

Performance figures that have been measured on a PC with Intel(R) Core(TM) i5-2300 CPU @ 2.80GHz, with 8GB of physical memory, and running the Windows 7 (64 bits) operating system are presented. The Java Virtual Machine (JVM) version 1.6.0_25-b06 has been restarted for each measure as well as for each of the 20 repetitions of each measure. **Results are in seconds and Megabytes, where appropriate**.

Table 4 shows the configurations that have been used for the JVM and for the relevant databases aiming to optimize execution time and were obtained empirically.

7.2 Model Insertion

Tables 5 and 6 show the results for the insertion of an XMI model into the databases. We assume availability of XMI model files so models written to an XMI file are omitted.

Table 4 – Configuration Options for Benchmarks

Config	Persistence Mechanism					
	XMI	Teneo/Hibernate	CDO[H2]	CDO[MySQL]	Neo4J	OrientDB
JVM	-Xmx6G	-Xmx6G	-Xmx5G	-Xmx5G	-Xmx6G	-Xmx5G
Database	n/a	default	default	default	2.2G MMIO	1.5G MMIO

Regarding insertion time, Teneo/Hibernate did not successfully insert set2 – set4 and CDO did not successfully insert set3 – set4 (neither with H2 nor with MySQL), as even with maximum memory allocated to both client and server in both cases, they threw a timeout exception, so values are omitted. For small model sizes, in the order of tens of megabytes (set0, set1), CDO performs the best but for larger ones, in the order of hundreds of megabytes (set2 – set4), Neo4J and OrientDB are not only able to store them successfully, but for set2 do so faster than CDO. It is worth noting that for set3 – set4, due to the sizes of the files, the computer's RAM is exhausted hence the operation is greatly bottlenecked by I/O from the hard disk. This results in a greater variance in the results and hence the averages presented here are influenced by multiple factors such as the physical location of each database on the hard disk.

Table 5 – Model Insertion (Persistent to Database) Size Results

Model	Size					
	XMI	Teneo/Hibernate	CDO[H2]	CDO[MySQL]	Neo4J	OrientDB
Set0	8.75	38.6	26.0	34.8	29.4	53.6
Set1	26.59	83.1	67.0	75.7	85.9	134.0
Set2	270.12	-	539	551	794	1197
Set3	597.67	-	-	-	1750	2591
Set4	645.53	-	-	-	1890	2789

Table 6 – Model Insertion (Persistent to Database) Execution time Results

Model	Time taken					
	XMI	Teneo/Hibernate	CDO[H2]	CDO[MySQL]	Neo4J	OrientDB
Set0	n/a	58.7	11.8	26.2	12.4	19.6
Set1	n/a	218.2	19.2	66.7	32.5	57.1
Set2	n/a	-	778.5	647.5	499.1	590.8
Set3	n/a	-	-	-	2210	2245
Set4	n/a	-	-	-	2432	2397

7.3 Query Execution Time and Memory Footprint

Table 7 shows the results for performing the first Grabats 2009 [Gra12, SJ09] query on the databases. As previously mentioned, the Grabats query finds all occurrences of *TypeDeclaration* elements that declare at least one public static method with the declared type as its returning type.

As Teneo/Hibernate did not insert set2 – set4 and CDO did not insert set3 – set4

Table 7 – GrabatsQuery Results

Model	Metric	Persistence Mechanism					
		XMI	Teneo/Hibernate	CDO[H2]	CDO[MySQL]	Neo4J	OrientDB
Set0	Time	1.20	4.53	0.60	0.61	0.11	0.43
	Mem (Max)	42	248	14	12	15	10
	Mem (Avg)	19	117	12	9	11	10
Set1	Time	2.28	7.34	1.12	1.06	0.62	1.18
	Mem (Max)	111	323	17	20	18	27
	Mem (Avg)	48	176	13	17	13	17
Set2	Time	16.51	-	12.94	12.20	3.10	9.83
	Mem (Max)	813	-	98	120	401	742
	Mem (Avg)	432	-	32	70	195	255
Set3	Time	84.91	-	-	-	6.71	24.41
	Mem (Max)	1750	-	-	-	960	2229
	Mem (Avg)	844	-	-	-	620	881
Set4	Time	145.67	-	-	-	7.16	29.65
	Mem (Max)	1850	-	-	-	1070	2463
	Mem (Avg)	939	-	-	-	866	1314

(neither with H2 nor MySQL), query values are omitted for these test cases. As can be observed, Neo4J demonstrates the best performance in terms of execution time and OrientDB is faster than XMI but also uses a comparable memory footprint. CDO has the lowest memory consumption for the queries it can run (we are not considering memory use of set0 and set1 as it is extremely low and the variance caused by the computer itself is significant) but is also slower to execute than Neo4J and comparable to OrientDB.

Using this empirical data we can deduce that even though OrientDB's Graph layer is competitive and can be an improvement to XMI even for the largest model sizes in this benchmark, due to the fact that it is built atop a document store causes its performance to be lower than that of Neo4J, which is a pure graph-based database.

Table 8 shows the results for performing the Grabats query using the EMC query on the databases next to their native query results from Table 7.

As can be expected this layer adds an overhead both in memory and execution time, but the results are still greatly superior to XMI persistence.

Figure 10 compares the total time taken for Ecore's XMI loader and our prototypes to answer the query, starting from a model provided in an XMI file. The querying time (at 0 times performed) is the time it takes to insert the model to the store as we assume the availability only of the XMI files.

The total time is calculated assuming that the persistence mechanism is disconnected from the query API each time but the persistence (for the NoSQL) is not deleted (for example if one execution of the query is performed on each working day) and can be used to visualize after how many such runs a NoSQL solution would be beneficial to deploy. It is worth noting that the query execution time for XMI, not counting the loading of the resource is comparable to Neo4J query execution times (seen in Table 7), so if a model only needs to be analyzed very few distinct times, with multiple queries executed, XMI is still the fastest approach, assuming that the

Table 8 – Grabats Query Results – Using Native and EMC

Model	Metric	Persistence Mechanism			
		Neo4J	OrientDB	Neo4J (EMC)	OrientDB (EMC)
Set0	Time	0.11	0.43	0.35	0.99
	Mem (Max)	15	10	12	15
	Mem (Avg)	11	10	11	12
Set1	Time	0.62	1.18	0.61	2.21
	Mem (Max)	18	27	12	23
	Mem (Avg)	13	17	11	14
Set2	Time	3.10	9.83	6.02	15.63
	Mem (Max)	401	742	520	910
	Mem (Avg)	195	255	280	390
Set3	Time	6.71	24.41	12.71	38.92
	Mem (Max)	960	2229	1320	2400
	Mem (Avg)	620	881	520	750
Set4	Time	7.16	29.65	14.99	41.37
	Mem (Max)	1070	2463	1410	2540
	Mem (Avg)	866	1314	810	870

client can handle the immense memory consumption it requires.

Regarding native querying, for set2, Neo4J is preferable to XMI after around 35 repeats while OrientDB after around 90. For set3, Neo4J is around 28 while OrientDB 37. For set4, Neo4J is around 18 while OrientDB 21.

We can observe that for any model size both NoSQL solutions are beneficial after some threshold, the larger the model size the earlier we can use NoSQL solution to persist it and that Neo4J is always more performant than OrientDB.

Regarding EMC querying we observe that for set2, OrientDB has similar gradient to XMI, with the lines only intersecting at around 2500 repeats and with Neo4J still outperforming XMI at around 49 repeats. For set3 and set4, we see similar results to Figure 10, with Neo4J surpassing XMI at 30 and 19 repeats respectively and OrientDB at 53 and 23 respectively.

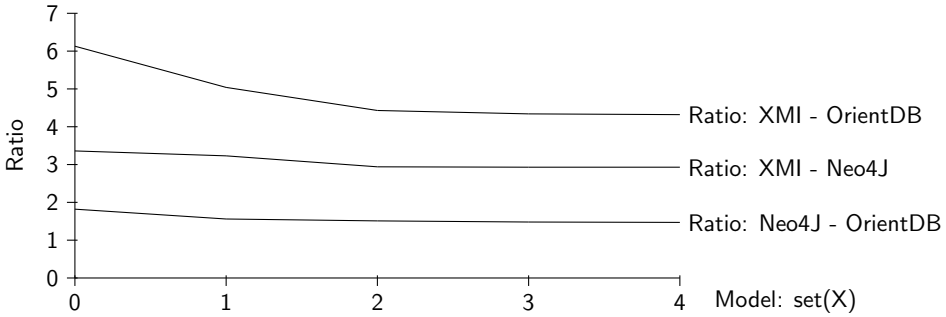
These results seem to show that the overhead of using Epsilon with OrientDB is sufficient enough to cause it to only be negligibly more efficient than XMI for relatively small model sizes (set0 – set2); when working with large enough model sizes (set3 – set4) though, OrientDB's EMC performance starts to reflect that of its native querying with respect to XMI. Regarding Neo4J, Epsilon's overhead only minorly effects its overall performance causing to be quickly surpass that of XMI for any model size.

7.4 Disc Space

As expected, Neo4J and OrientDB require more disk space than XMI. Figure 9 shows the ratios of relative disk space needed to store the different models (set0 – set4) for the different technologies, using the results in Table 5.

All three ratios, for large enough model sizes, tend to a constant. This constant is estimated to be 4.3 for XMI – OrientDB, 2.9 for XMI – Neo4J and 1.45 for Neo4J – OrientDB. For smaller model sizes variables such as database-specific overhead seem

Figure 9 – Ratios of relative disk space used for the different persistence mechanisms



to influence the ratios substantially (hence the larger ratios with respect to XMI for smaller models). Hence, for large enough models, we can expect an OrientDB store to be around 4.3x as large as its XMI file and a Neo4J store around 2.9x as large. Furthermore the results seem to show that Neo4J is more efficient in storing the data relative to OrientDB, which can be expected as it handles references in a more lightweight fashion, as explained in Sections 5.1 and 5.2. The ratio between Neo4J and OrientDB seems to indicate that for both databases their relative overheads discussed above are similar, but OrientDB is less efficient in that regard (with a 19.2% delta in the ratio between Neo4J and OrientDB at set0 and the one at set4).

8 Conclusions

This paper has explored the use of graph-based NoSQL databases to support scalable persistence of large models by exploiting the index-free adjacency of nodes provided by these stores. Prototypes for integrations of both Neo4J and OrientDB with EMF as well as with Epsilon's EMC have been implemented and described in detail. Benchmarks using the Grabats 2009 query have been executed and have shown that:

- Native queries provide performance results which greatly surpass XMI text file based stores and can store models larger than CDO.
- Back-end independent queries (using Epsilon's EMC) provide a maintainable and performant alternative to native querying.

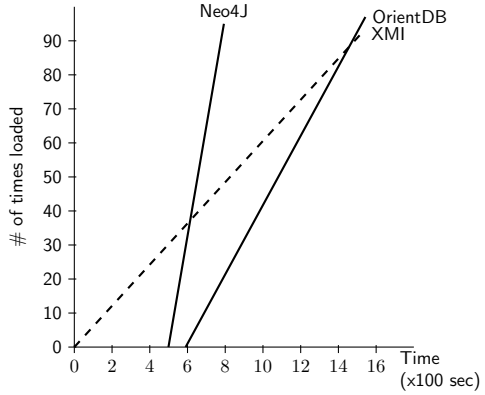
These results can promote further research and development of large-scale model persistence solutions based on graph-based NoSQL databases, a feasible performant alternative to XMI.

Further work in this area would include implementation of features allowing for more memory-efficient client access to the repositories, for scenarios where execution time can be traded for a lower memory footprint, as well as creation of query optimizations that allow for more efficient model navigation, hence eliminating some of the execution time required to run them.

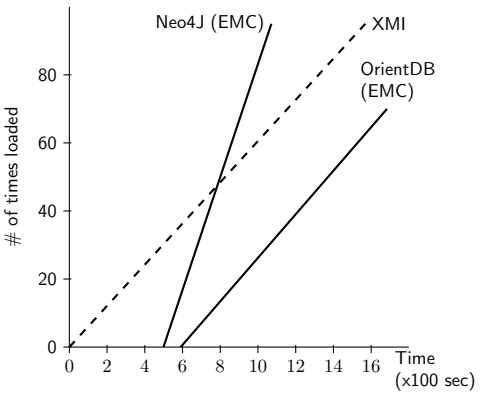
Acknowledgments

The work in this paper was partly supported by the European Commission via the OSSMETER FP7 project (#318736). Information included in this document reflects only the authors views. The European Commission is not liable for any use that may be made of the information contained herein.

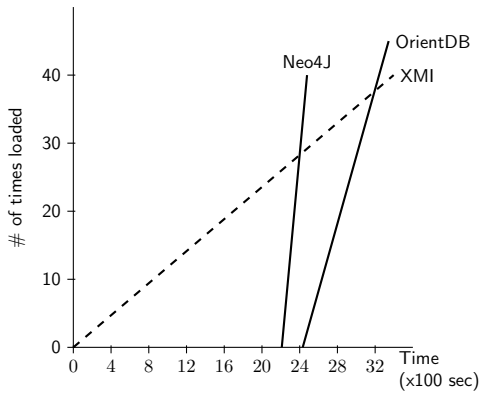
Figure 10 – Performance Comparison for full execution of the Grabats Query from XMI through the relevant persistence mechanism using native and EMC querying



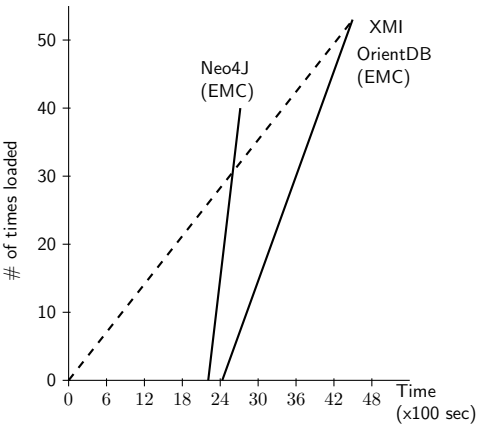
(a) Performance for **set2**



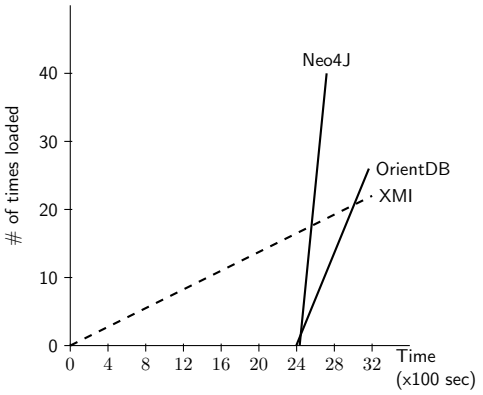
(b) Performance for **set2** (EMC)



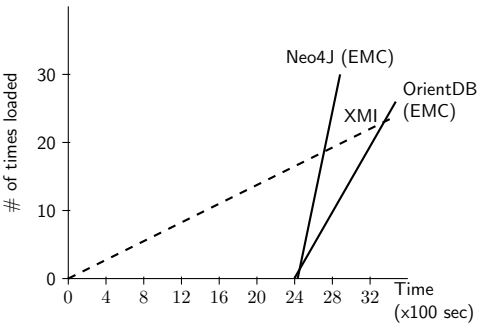
(c) Performance for **set3**



(d) Performance for **set3** (EMC)



(e) Performance for **set4**



(f) Performance for **set4** (EMC)

References

[BK12] Konstantinos Barmpis and Dimitrios S. Kolovos. Comparative analysis of data persistence technologies for large-scale models. In *Proceedings of the 2012 Extreme Modeling Workshop*, XM '12, pages 33–38, New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2467307.2467314, doi:10.1145/2467307.2467314.

[BK13] Konstantinos Barmpis and Dimitris Kolovos. Hawk: towards a scalable model indexing architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 6:1–6:9, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2487766.2487771, doi:10.1145/2487766.2487771.

[Cat11] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011. URL: http://doi.acm.org/10.1145/1978915.1978919, doi:10.1145/1978915.1978919.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comp. Syst.*, 2008. URL: http://doi.acm.org/10.1145/1365815.1365816, doi:10.1145/1365815.1365816.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proc. 21st ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, 2007. URL: http://doi.acm.org/10.1145/1294261.1294281, doi:10.1145/1294261.1294281.

[Dim08] Dimitrios S. Kolovos, Louis M. Rose, Antonio Garcia Dominguez and Richard F. Paige. *The Epsilon Book*. 2008. URL: http://www.eclipse.org/epsilon/doc/book/.

[Edw11] Edward Willink. Aligning OCL with UML. In *Proceedings of the Workshop on OCL and Textual Modelling*, volume 44 of *Electronic Communications of the EASST*, 2011. URL: http://journal.ub.tu-berlin.de/eceasst/article/view/664.

[Gra12] Grabats2009. 5th International Workshop on Graph-Based Tools [online], 2012. [Accessed 1 June 2012] Available at: http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/.

[Hba12] Hbase Developers. Hbase, Tabular NoSQL Database [online], 2012. [Accessed 1 June 2012] Available at: http://hbase.apache.org/.

[KPP06] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon object language (eol). In *Proceedings of the Second European Conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA'06, pages 128–142, Berlin, Heidelberg, 2006. Springer-Verlag. URL: http://dx.doi.org/10.1007/11787044_11, doi:10.1007/11787044_11.

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Scalability: The Holy Grail of Model Driven Engineering. In *Proc. Workshop on Challenges in MDE, collocated with MoDELS '08, Toulouse, France*, 2008. URL: http://ssel.vub.ac.be/ChaMDE08/_media/chamde2008_proceedingsd121.pdf?id=wsorganisation&cache=cache#page=10.

[Lea10] Neal Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2):12 –14, February 2010. doi:10.1109/MC.2010.58.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010. URL: http://doi.acm.org/10.1145/1773912.1773922, doi:10.1145/1773912.1773922.

[MDBS09] Alix Mougenot, Alexis Darrasse, Xavier Blanc, and Michèle Soria. Uniform Random Generation of Huge Metamodel Instances. In *Proceedings of ECMDA-FA '09*, pages 130–145, Berlin, Heidelberg, 2009. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-02674-4_10, doi:10.1007/978-3-642-02674-4_10.

[MFM+09] Parastoo Mohagheghi, Miguel Fernandez, Juan Martell, Mathias Fritzsche, and Wasif Gilani. MDE Adoption in Industry: Challenges and Success Criteria. In *Models in Software Engineering*, volume 5421 of *Lecture Notes in Computer Science*, pages 54–59. Springer, 2009. URL: http://dx.doi.org/10.1007/978-3-642-01648-6_6.

[Mon12] MongoDB Developers. MongoDB, Document-Store NoSQL Database [online], 2012. [Accessed 1 June 2012] Available at: www.mongodb.org/.

[Neo12] Neo4J Developers. Neo4J, Graph NoSQL Database [online], 2012. [Accessed 1 June 2012] Available at: http://neo4j.org/.

[Ore10] Kai Orend. Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace an Object-relational Persistence Layer. *Architecture, p. 100*, April 2010. http://weblogs.in.tum.de/file/Publications/2010/Or10/Or10.pdf.

[Ori12] OrientDB Developers. OrientDB, Hybrid Document-Store and Graph NoSQL Database [online], 2012. [Accessed 1 June 2012] Available at: http://www.orientechnologies.com/.

[PCM11] Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. Morsa: a scalable approach for persisting and accessing large models. In *Proceedings of MODELS'11*, pages 77–92, Berlin, Heidelberg, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=2050655.2050665.

[PPS11] Rabi Prasad Padhy, Manas Ranjan Patra, and Suresh Chandra Satapathy. RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database's. *IJAEST*, Vol.11(1), 2011. URL: http://www.ijaest.iserp.org/archieves/19-Sep-15-30-11/Vol-No.11-Issue-No.1/3.IJAEST-Vol-No-11-Issue-No-1-RDBMS-to-NoSQL-Reviewing-Some-Next-Generation-Non-Relational-Database's-015-030.pdf.

[Ric09] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, Fiona A.C. Polack. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *Proc. 14th IEEE International Conference on Engineering of Complex Computer Systems*, Potsdam, Germany, 2009. URL: http://dx.doi.org/10.1109/ICECCS.2009.14.

[SJ09] Jean-Sebastien Sottet and Frédéric Jouault. Program comprehension. In *Proc. 5th Int. Workshop on Graph-Based Tools*, 2009. URL: http://is.ieis.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2009reverseengineering.pdf.

[Sto10] Michael Stonebraker. Sql databases v. nosql databases. *Commun. ACM*, 53(4):10–11, April 2010. URL: http://doi.acm.org/10.1145/1721654.1721659, doi:10.1145/1721654.1721659.

About the authors

Konstantinos Barmpis is a second year EngD (Engineering Doctorate) student in the Enterprise Systems Research Group at the University of York.

He has a bachelor's degree from Imperial College London in Joint Mathematics and Computer Science and a master's degree from the University of Saint Andrews in Software Engineering. He is now focused on researching scalability issues in MDE.

His e-mail address is kb@cs.york.ac.uk and his web-page is http://www.cs.york.ac.uk/~kb.

Dimitrios S. Kolovos is a lecturer in Enterprise Systems in the Department of Computer Science of the University of York.

To date, he has published more than 70 articles in international journals, conferences and workshops in the field of MDE, and is currently leading the development of the Epsilon open source MDE platform (http://www.eclipse.org/epsilon).

His e-mail address is dimitris.kolovos@york.ac.uk and his web-page is http://www.cs.york.ac.uk/~dkolovos.