Comparative Analysis of Data Persistence Technologies for Large-Scale Models

Konstantinos Barmpis
Department of Computer Science
University of York
York, United Kingdom
kb@cs.york.ac.uk

Dimitrios S. Kolovos
Department of Computer Science
University of York
York, United Kingdom
dimitris.kolovos@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. Benchmarking of these technologies is then performed, 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 dramatically increased in the past decade as it provides several benefits compared to traditional software engineering practices, such as improved productivity and reuse [1], which allow for systems to be built faster and cheaper, two of the important factors in industrial environments. Nevertheless, certain limitations of MDE such as scalability concerns which prevent its wider use in industry [2, 3] 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.
3. Collaborative work: multiple developers checking out a part of their model and querying or editing it, as well as being able to commit their changes successfully.

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. The remainder of the paper is organized as follows. Section 2, introduces current persistence technologies used in the Eclipse Modeling Framework (EMF) and Morsa, a NoSQL prototype for scalable persistence of EMF models. Section 3 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 4 the produced prototypes are compared with existing solutions in terms of performance. Finally, section 5 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 and provides an overview of the current state of practice in model persistence and of proposed approaches for managing large-scale models efficiently.

2.1 Model-Driven Engineering

In MDE models are first class artefacts of the software engineering process. They are 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). EMF is a framework that facilitates the definition and instantiation of metamodels. In EMF, metamodels are defined using the Ecore metamodeling language, a high level overview of which is illustrated in Figure 1.

Metamodel elements are created using *EClass*es. These elements contain *EReference*s to other metamodel elements and *EAttribute*s, which are used to define the attributes any model element of that class can contain. Model elements are created from the *EModelElement* class. They can contain *EAttribute*s, which store information about the element

XM'12, October 01 2012, Innsbruck, Austria

(such as the ISBN number of a book model element) and can have *EReference*s to other model elements.

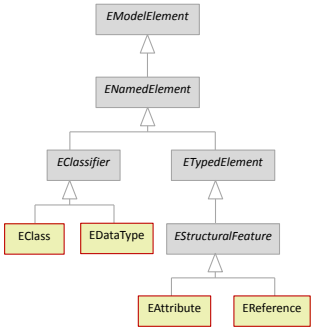

Figure 1: Simplified Diagram of the Ecore Metamodeling Language

By default, models in EMF are stored in a standard XML-based representation called XML Metadata Interchange that is an OMG-standardized format that was 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. To address these limitations of XMI, persisting models in relational databases has been proposed.

Examples of such approaches include the Connected Data Objects (CDO)[1] project and Teneo-Hibernate[2]. 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 inefficient. Due to the highly interconnected nature of most models, complex queries require multiple expensive table joins to be executed and hence do not scale well for large models. Even though Teneo-Hibernate tries 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 all EClasses made their own table of EObjects) the fact that the database is made up 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 [4] 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

[1]http://www.eclipse.org/cdo/
[2]http://wiki.eclipse.org/Teneo/Hibernate

The NoSQL movement is a contemporary approach to data persistence using novel storage approaches. NoSQL databases provide flexibility and performance as they are not limited by the traditional relational approach to data storage [5]. Each type of NoSQL database is tailored for storing a different and specific 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 [6]. 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 are not of the relational model. Such companies are for example Amazon (Dynamo database [7]), Google (Bigtable database [8]) and Facebook (Cassandra database [9]).

There are four widely accepted types of NoSQL databases, which use distinct approaches in tackling data persistence, three of which are described by [10] 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 conventional storage. 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 huge 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 [8] and Hbase [11].
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 [12] and OrientDB [13].
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 [14] and the graph layer (OGraphDatabase) of OrientDB [13].

NoSQL databases have a loosely defined set of characteristics and properties [15]:

- 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 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 [16]:

1. Lack of a 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 *Morsa: NoSQL Back-end Prototype*

Morsa [4] is a prototype that attempts to address the issue of scalable model persistence by 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 a key-value pair, alongside its persistence metadata (such as reference to its metaclass). 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) to an array of references to the documents that represent its root objects. A high level overview of the architecture is displayed in Figure 2 by [4].

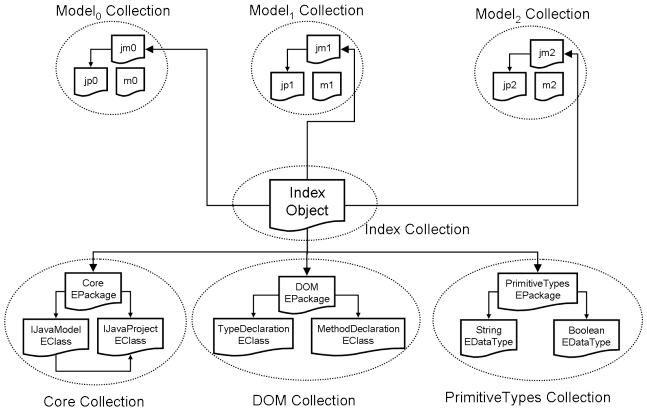

Figure 2: Persistence back-end structure excerpt for Morsa

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 replacement policy that chooses which objects must be unloaded from the client memory (should the cache be deemed full by the current 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 EReferences (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.

3. DATA PERSISTENCE TECHNOLOGIES FOR LARGE-SCALE MODELS

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. Therefore, in this work prototype model stores based on Neo4J [14] and OrientDB [13] have been created, and their efficiency has been compared against the default EMF XMI text store, a MySQL SQL database (using Teneo-Hibernate to integrate with EMF), and Morsa. This section discusses the design and implementation of the two model stores and briefly discusses the Grabats 2009 case study, a subset of which is used for the evaluation of these stores.

Due to the nature of the data being stored, key-value stores as well as tabular NoSQL data-stores were not considered, as they are tailored for handling a different problem (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 one (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 became comparing two graph databases. The reasoning behind choosing these technologies was that Neo4J is an extremely 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 3.1 and 3.2 below.

The Neo4J and OrientDB stores attempt to solve the aforementioned scalability issues using graph databases to store large model 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.

3.1 Neo4J, Graph Database

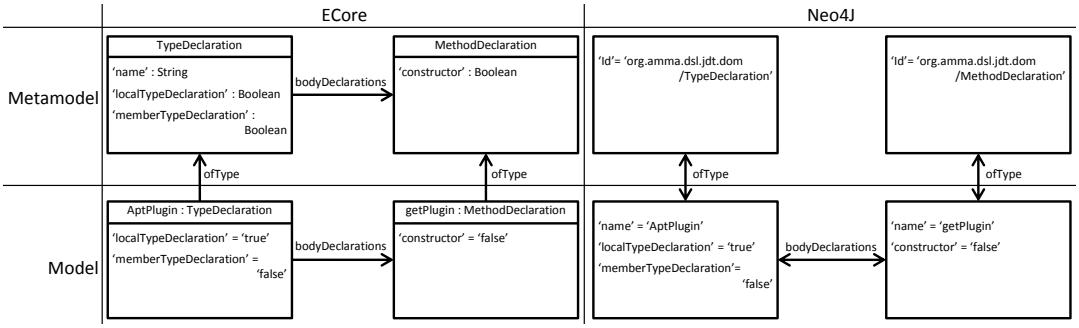

Figure 3: Example mapping from Ecore to Neo4J

Neo4J is a popular, commercial Graph Database released under the GNU Public License (GPL) and the Affero GNU Public License (AGPL) licenses. It is written 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. 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 both its class and its superclasses) that are set.
- *Relationship*s from model element nodes to other model element nodes. These represent the references of the model element to other model elements.
- *Node*s representing EClasses of the metamodel(s) the models stored are instances of. These nodes only have an *id* property denoting the unique identifier of the metamodel they belong to, followed by their name, for example: `org.amma.dsl.jdt.core/IJavaElement` is the id of the EClass *IJavaElement* in the `org.amma.dsl.jdt.core` Ecore metamodel. These lightweight metaclass nodes are used to speed up query time by providing references 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.
- An index containing the ids of the EClasses and their appropriate location in the database. This allows a typical query (such as the Grabats query described below) to have this as a starting point, in order to find all 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 evaluate 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 (only the model information is stored in full) 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 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) and keeping it consistent with its metamodel.

Figure 3 shows how a simple Ecore metamodel and model are stored in Neo4J. As seen above, metamodel EClasses are only stored as lightweight nodes containing only their NsURI as a property.

3.2 OrientDB, Hybrid Document Store, Graph Database

OrientDB is a novel document-store database released under the Apache 2 License. It is also written in 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, emulating the property of index-free adjacency of documents. Its core constructs are *ODocuments* (which can contain an arbitrary number of properties, which can be dynamically added and removed).

An OrientDB-based model store consists of the following:

- *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 class and its superclasses) that are set.
- *ODocument*s representing (the references (edges)) from model element nodes to other model element nodes. These represent the references of the model element to other model elements.
- *ODocument*s representing metamodel EClasses of the metamodel the model stored adheres to. These nodes have an 'id' field denoting the NsUri of their package followed by their Name, for example: `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 metaclass nodes are used to speed up query time 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

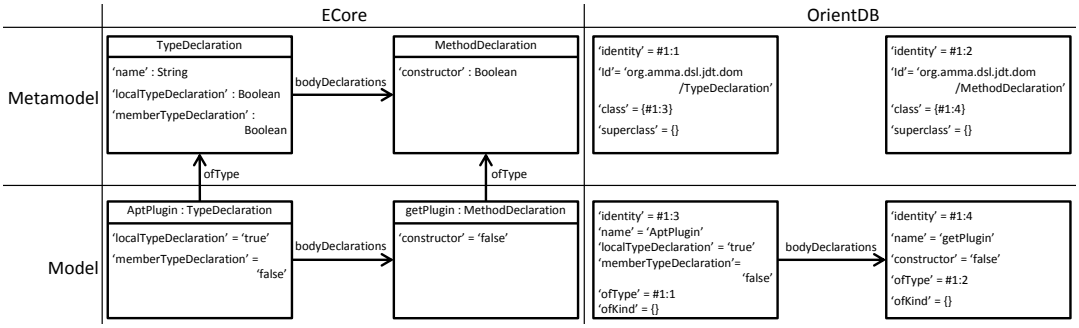

Figure 4: Example mapping from Ecore to OrientDB

manner to how the ones in Morsa are) so are avoided, and empirical evidence shows that execution time using references instead of lists is much slower.

- An index containing the ids of the metaclasses and their appropriate location in the database. This allows a typical query to have this 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.

Figure 4 shows how a simple Ecore metamodel and model are stored in OrientDB.

3.3 The Grabats 2009 Case Study and Query

The 5th International Workshop on Graph-Based Tools ran the Grabats 2009 contest [17]. The contest comprised several tasks, including the case study used in this paper for benchmarking the different technologies. 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)). These models all conform to the JDTAST metamodel, which is a metamodel used to model Java source code.

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) [18]. 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.

4. EVALUATION

In this section, XMI, Teneo/Hibernate (using a MySQL database), Morsa and the two prototypes implemented in this work are compared to assess their performance and efficiency in terms of memory use. 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 repetition of each measure. Results are in seconds and Megabytes, where appropriate. Note that all results for the Morsa test cases were **not** performed under these conditions as the released prototype could not be run successfully. Hence the results are taken from the published benchmarks performed by Pagan, Cuadrado, and Molina [4], who used a Intel CoreI5 760 PC at 2.80GHz with 8GB of physical RAM running 64-bit Linux 2.6.35. This setup mostly coincides with the one used for this paper's benchmarks but the results cannot be directly related due to this discrepancy and are included only for reference.

Table 1 shows the configurations that have been used for the JVM and for the relevant databases to optimize execution time and have all been obtained empirically. As the Morsa prototype was not ran, the exact configurations used by [4] are not known.

Table 1: Configuration Options for Benchmarks

Configuration	Persistence Mechanism				
	XMI	Teneo/Hibernate	Morsa	Neo4J	OrientDB
JVM	-Xmx6G	-Xmx6G	-	-Xmx6G	-Xmx5G
Database	n/a	default	-	2.2G MMIO	1.5G MMIO

Table 2 shows the results for the insertion of an XMI model into the database. We assume availability of XMI model files so tests of models written to an XMI file are omitted. Teneo/Hibernate did not successfully insert set2 - set4 so values are omitted. The Morsa publication does not report on insertion time so is also omitted. Neo4J is the fastest to complete a successful full insert for all model sizes, OrientDB is closely second with comparable execution time and success for all model sizes and Teneo/Hibernate is considerably slower as well as failing to insert past set1. 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 2: Model Insertion (Persistent to Database) Results

Model	Size	Persistence Mechanism (Time taken)				
		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

Table 3 shows the results for performing the first Gra-

bats 2009 [17, 18] query on the databases. As previously mentioned, the Grabats query finds all occurrences of *Type-Declaration* 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 query values are omitted for these test cases. As the Morsa publication included only one memory footprint value, this is taken as an average memory use and a max memory use is omitted.

Table 3: Grabats Query Results

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

As can be observed, Neo4J demonstrates the best performance in terms of execution time, Morsa is the most memory efficient (as it is optimized for memory consumption) and OrientDB is faster than XMI but also uses a comparable memory footprint. Teneo/Hibernate not only has the highest memory consumption for the queries it can run but is also considerably slower to execute.

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.

5. 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 have been implemented and demonstrate performance results which surpass XMI text file based stores as well the Teneo/Hibernate solution. These results can promote further research and development of large-scale model persistence solutions based on graph-based NoSQL databases. Further work in this area would also include integration of these prototypes with model management (e.g. validation, model-to-text and model-to-model transformation) languages, as well as 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.

6. REFERENCES

[1] Mohagheghi, P., Fernandez, M., Martell, J., Fritzsche, M., Gilani, W.: MDE Adoption in Industry: Challenges and Success Criteria. In: Models in Software Engineering. Volume 5421 of Lecture Notes in Computer Science. Springer (2009) 54–59

[2] Kolovos, D.S., Paige, R.F., Polack, F.A.: Scalability: The Holy Grail of Model Driven Engineering. In: Proc. Workshop on Challenges in MDE, collocated with MoDELS '08, Toulouse, France. (2008)

[3] Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform Random Generation of Huge Metamodel Instances. In: Proceedings of ECMDA-FA '09, Berlin, Heidelberg, Springer-Verlag (2009) 130–145

[4] Pagán, J.E., Cuadrado, J.S., Molina, J.G.: Morsa: a scalable approach for persisting and accessing large models. In: Proceedings of MODELS'11, Berlin, Heidelberg, Springer-Verlag (2011) 77–92

[5] Stonebraker, M.: SQL Databases vs NoSQL Databases. Commun. ACM **53(4)** (2010)

[6] Orend, K.: 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/2010/0r10/0r10.pdf`.

[7] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: Proc. 21st ACM SIGOPS symposium on Operating systems principles. SOSP '07 (2007) 205–220

[8] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comp. Syst. (2008)

[9] Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2) (April 2010) 35–40

[10] Padhy, R.P., Patra, M.R., Satapathy, S.C.: RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database's. IJAEST **Vol.11(1)** (2011)

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

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

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

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

[15] Cattell, R.: Scalable SQL and NoSQL data stores. SIGMOD Rec. **39**(4) (May 2011) 12–27

[16] Leavitt, N.: Will NoSQL Databases Live Up to Their Promise? Computer **43**(2) (February 2010) 12 –14

[17] 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/`.

[18] Sottet, J.S., Jouault, F.: Program comprehension. In: Proc. 5th Int. Workshop on Graph-Based Tools. (2009)