Neo4EMF, a Scalable Persistence Layer for EMF Models

Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, David Launay

Neo4EMF, a Scalable Persistence Layer for EMF Models

Amine Benelallam[1], Abel Gómez[1], Gerson Sunyé[1], Massimo Tisi[1], and David Launay[2]

[1] AtlanMod, INRIA, Mines de Nantes, & LINA, France
{amine.benelallam|abel.gomez-llana|gerson.sunye|massimo.tisi}@inria.fr
[2] Mia-Software Nantes, France dlaunay@mia-software.com

Abstract. Several industrial contexts require software engineering methods and tools able to handle large-size artifacts. The central idea of abstraction makes model-driven engineering (MDE) a promising approach in such contexts, but current tools do not scale to very large models (VLMs): already the task of storing and accessing VLMs from a persisting support is currently inefficient. In this paper we propose a scalable persistence layer for the de-facto standard MDE framework EMF. The layer exploits the efficiency of graph databases in storing and accessing graph structures, as EMF models are. A preliminary experimentation shows that typical queries in reverse-engineering EMF models have good performance on such persistence layer, compared to file-based backends.

1 Introduction

With large-scale software engineering becoming a compelling necessity in several industrial contexts, companies need tools that are capable to scale efficiently. One of such companies is MIA Software, part of the group Sodifrance, working in the field of software modernization.

The emergence of new techniques and tools for building complex, adaptive and distributed systems has raised a need for the modernization of existing software. A software modernization process follows a systematic approach by first building high level abstractions from source code through reverse engineering, and then using these abstractions to understand, evaluate the quality, extract enterprise architectures and finally, improve the system. A natural approach to reverse engineering is to use Model-Driven Engineering (MDE) tools and in particular those based on the Eclipse Modeling Framework (EMF).

Indeed, EMF has become a *de facto* standard for building MDE tools, providing a common base for different purposes: reverse engineering [6, 26], model transformation [14,19], and code generation [5,18]. However, EMF was designed to support modeling activities in the first place and has shown clear limits when dealing with large models, which is often the case of automatically generated models.

While several solutions to persist EMF models exist, they are limited for two reasons. First, most of them do not allow partial model load and unload, and

hence, the size of the models they can handle is limited by the memory size; and second, models are structurally graphs and most of the existing solutions are based on relational databases, which are not fully adapted to store graphs.

In this paper we identify specific large-model requirements, discuss the limitations of EMF with this respect, and present a scalable persistence layer for EMF models that meets these requirements. Our persistence layer, Neo4EMF, is built on top of the popular graph database Neo4j. Neo4EMF is open-source, publicly available at [3] and it can be immediately used by existing EMF-based tools, without modifying them, to improve their applicability to complex industrial contexts.

Neo4EMF provides two main benefits to the state-of-the-art MDE tools: (i) a scalable access to very large models, with on-demand loading of model elements, (ii) the possibility to exploit the enterprise features of Neo4j, like online backups, horizontal scalability and advanced monitoring. To evaluate this aspect we perform a set of queries in the domain of software modernization, and we compare the execution performance of these queries with the *de facto* standard persistence layers for EMF: XMI and CDO [13].

The paper is organized as follows. Section 2 introduces the concept of persistence layer and graph database, Section 3 describes our proposed persistence layer, Section 4 experimentally evaluates the performance of our layer. Section 5 compares our proposal to existing related work, and finally Section 6 concludes and draws the future perspectives of the tool.

2 Background

2.1 Persistence layers

Software developers often need to persist the state of one or more objects using an existing storage support: relational databases, XML files, etc. There are two main approaches to achieve object persistence. The first one is to hard code the persistence behavior in the class. This approach is efficient and adapted to small applications, but increases the coupling between the class and the storage support. The second approach is to use a persistence layer [2], i. e., a robust and adaptable mechanism that hides storage details from developers and reduces coupling between the storage support and classes. The adaptability of this approach is ensured by a mapping that binds the object model, composed of classes, references, and attributes to the storage model: tables, columns, etc. The object and the storage models can evolve independently, provided a mapping between their concepts is possible. The mapping reduces the development cost of persistent classes, but has a significant impact on the performance.

The emergence of code generation techniques allows developers to adopt a third approach that combines the advantages of the two others. It consists on automatically generating an efficient code for persistence using the correspondence mapping as a generation parameter. Contrary to a persistent layer, the adaptability is not ensured at runtime, but at generation time.

Persistence layers for EMF Since the publication of the XMI standard [20], XML-based serialization has been the preferred format for storing and sharing models and metamodels. Some tools, such as EMF [12], have even adopted it as their canonical representation. However, XMI-based serialization in EMF results to be extremely inefficient: (i) XMI files sacrifice compactness in favor of human-readability and (ii) XMI files need to be completely parsed to obtain a navigational model of their contents. The first factor greatly reduces the efficiency in I/O accesses, while the second greatly increases the memory required to load and query models and limits the use of proxies and on-demand loading to inter-document relationships. Moreover, XMI-based implementations do not provide advanced features such as concurrent modifications, model versioning, or access control out-of-the-box.

The design of CDO [13], built on top of EMF, solves most of these problems. CDO was initially envisioned, among other things, as a framework to manage large models in a collaborative environment with a low memory footprint. CDO implements a client-server architecture with transactional and notification facilities where model elements are loaded on demand. CDO servers (usually called *repositories*) are built on top of different data storage solutions. However, in practice, only relational databases are commonly used. Indeed, only *DB Store* [8], which uses a proprietary Object/Relational mapper, supports all the features of CDO and is regularly released in the *Eclipse Simultaneous Release* [9–11].

2.2 Graph databases

The volume of data that organizations gather has grown explosively in recent years, showing a need for solutions that scale-out, as well as the limits of relational databases. To overcome these limits, new technologies for data management have raised, the so-called NoSQL databases [25]. Despite their non-respect of the ACID properties, these database are able to manage large-scale data on highly distributed environments.

Among the different data models used on NoSQL databases (e. g.column, document, or key-value), graph databases are particularly adapted to store EMF models. The graph data model uses graph structures with nodes, edges, and properties to store data and provides index-free adjacency. Although this data model is not new—the research on graph databases was popular back in the 1990s—it became again a topic of interest due to the large amounts of graph data introduced by social networks and the web in general.

3 Neo4EMF

Neo4EMF is our proposal for scalable model persistence built on top of the EMF framework. Neo4EMF is an open source project that aims at providing a compatibility layer between the EMF API and a graph-based storage subsystem. Specifically, Neo4EMF is built on top of Neo4j [23], a NoSQL database which is distributed under the terms of the (A)GPLv3.

EMF-based models can easily be described in terms of graph concepts, since there is a natural mapping between the two representations. This natural translation is the main motivation that lead us to choose a native graph database instead of another NoSQL database. Since graph databases like Neo4j have shown good performance for connected data operations, we argue that they are a promising platform for model manipulation.

In this section we first briefly provide an overview of the underlying mapping between EMF models and Neo4j artifacts through a running example, then we describe the main design principles of Neo4EMF.

3.1 Mapping EMF models and Neo4j graphs

Figure 1 shows a small excerpt of the *Java* metamodel provided by MoDisco [26]. This metamodel describes *Java* programs in terms of *Packages*, *ClassDeclarations*, *BodyDeclarations* and *Modifiers*. A *Package* is a named container that groups a set of *ClassDeclarations* through the *ownedElements* composition. A *ClassDeclaration* contains a *name* and a set of *BodyDeclarations*. Finally, a *BodyDeclaration* contains a *name*, and its *visibility* is described by a single *Modifier*.

Figure 2 shows a simple instance of this metamodel. This instance contains a single *Package* (`package1`), containing only one *ClassDeclaration* (`class1`). The *Class* contains only the `bodyDecl1` *BodyDeclaration*, which is `public`. Figures 1, 2, and 3 show that:

- Model elements are represented as *nodes*. *Nodes* `p1`, `c1`, `d1` and `m1` are examples of this, and correspond to the elements `p1`, `c1`, `d1` and `m1` shown in Figure 2. A `ROOT` element denotes the model element(s) that directly or indirectly references all the other elements in the model.
- Element attributes are represented as *node properties* – a pair of ⟨*property name*, *property value*⟩ contained in the corresponding *node*. This can be observed in *nodes* `p1`, `c1`, `d1`, and `m1` again.
- Metamodel elements are also represented as *nodes*. *Nodes* representing metamodel elements are indexed to ease their access. These kind of *nodes* also contain two node properties. As it can be seen in *nodes* `P`, `C`, `B`, and `M` (which correspond to *Package*, *ClassDeclaration*, *BodyDeclaration*, and *Modifier* on Figure 1), the first property holds the name of the metamodel element, and the second property the metamodel unique identifier (also known as *nsURI*).

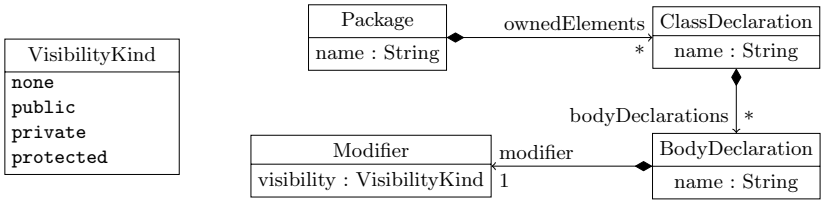

Fig. 1: Excerpt of the *Java* metamodel

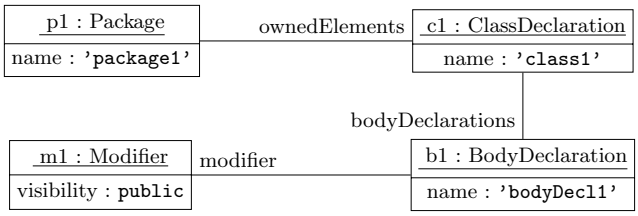

Fig. 2: Sample instance of the *Java* metamodel (nsURI: `http://java`)

- Conformance relationships are represented as an outgoing *relationship* of type `INSTANCE_OF` pointing to the node representing the corresponding metamodel element, as exemplified by the horizontal arrows of Figure 3.
- References are represented as *relationships*. To avoid naming conflicts in *relationships*, we use the following convention for assigning names: `CLASS_NAME__REFERENCE_NAME`. Vertical arrows in Figure 3 are examples of references. Bidirectional references would be represented with two separate directed graph *relationships*.

3.2 Neo4EMF design principles

Figure 3 shows the high-level architecture of Neo4EMF. In this section we introduce the different design principles that we respected in the development of Neo4EMF.

Compliance with standard APIs. In order to keep compliance with EMF, Neo4EMF provides a feature to generate an adapted Java code implementation

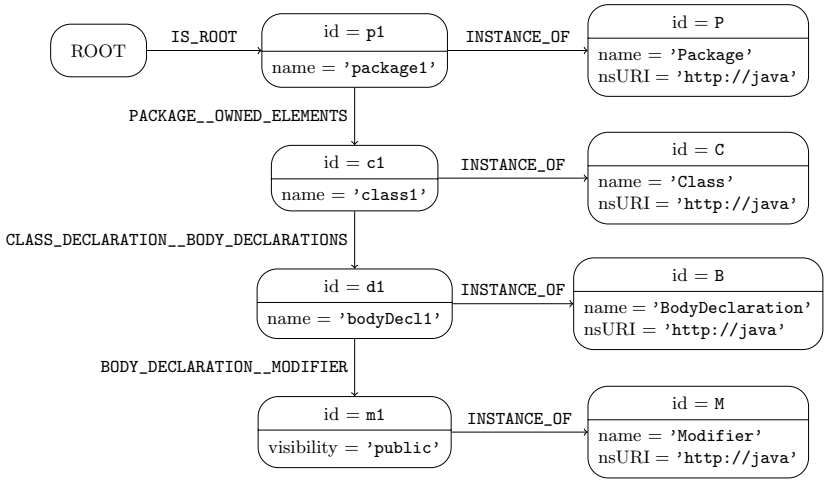

Fig. 3: Representation of the sample instance of the *Java* metamodel in Neo4j

allowing a refined on-demand loading. To allow for a fine-grained on-demand load mechanism even when using the Java generated API, Neo4EMF provides an adapted code generator supporting all the kinds of EMF generation (reflective, virtual, and dynamic). Neo4EMFObject extends the EMF *EObject* class with additional metadata such as the *id*. In addition to the default package organization, we generate a Java class containing a map from the model references to the Neo4j Relationships.

On demand loading. Neo4EMF uses an on-demand loading mechanism that reduces memory footprint and allows programs to load and query large models in systems with limited memory. This capabilities are provided for both the Neo4EMF dynamic API and the Neo4EMF generated Java API. These APIs are kept fully compliant with the standard EMF methods to load, navigate, modify, and save models. When a resource is loaded, only the root elements of the model are charged in memory, without any reference to their features. Depending on the user's query, the rest of the model is to be loaded. Thus, when a feature is queried, Neo4EMF checks if the elements already exist in the *cache memory*, if not they are loaded from the backend store.

Lightweight model change tracking. Saving model changes in XMI is time consuming, especially when dealing with in large models. The standard serialization mechanisms must traverse the whole resource to save a file. Neo4EMF uses an event-driven change notification approach to keep track of the model changes. Every Neo4EMFObject contains an adapter that sends a notification for each change to a shared listener. Notifications are stored in a *ChangeLog model*, which is asynchronously analyzed to optimize persistence operations. In this case, instead of traversing the whole resource to save the changes, Neo4EMF queries a *ChangeLog model*, and saves only the modified elements. Here, a model change can either be a creation of a new element, an edition of feature(s) of an existing one, or a deletion. Figure 4 shows the metamodel of the *ChangeLog model*.

Lightweight first time loading. Neo4EMF Java code generation separates objects data from their objects, in the sense that, every generated class references to an inner class holding all the class features. This allows a light-weight first time loading of Neo4EMF Objects.

4 Experimental evaluation

In this section, we evaluate how the access time of Neo4EMF scales in increasingly large scenarios, and we compare it against CDO (with H2 as relational database backend) and XMI. These experiments are performed over 3 EMF models that conform to the Java Metamodel proposed in MoDisco [26] and reverse-engineered from existing code using the MoDisco Java Discoverer. As

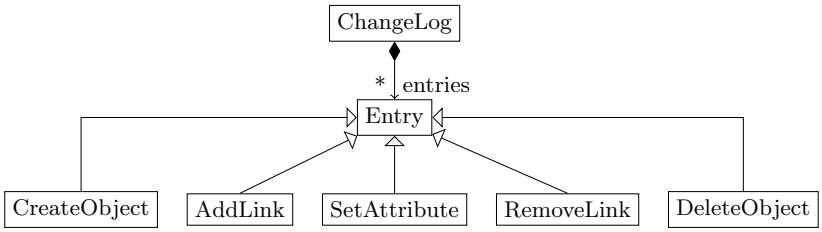

Fig. 4: Neo4EMF ChangeLog

starting code we used 3 sets of Eclipse plugins, of increasing size. Table 1 details how the experiments vary in size and thus in the number of elements:

4.1 Execution environment

Experiments are executed in a laptop computer running Windows 7 Enterprise 64. The most significative hardware elements are: an Intel Core i7 processor 3740QM (2.70GHz), 16 GB of DDR3 SDRAM (800MHz), and a Samsung SM841 SATA3 SSD Hard Disk (6GB/s). Experiments are executed on Eclipse version 4.3.1 running Java SE Runtime Environment version 1.7 (specifically, build 1.7.0_40-b43).

In order to compare the three technologies, we generate three different EMF access APIs, starting from the Java MoDisco Metamodel respectively with 1) EMF standard parameters, 2) CDO parameters, and 3) Neo4EMF generator. We import the 3 experimental models, originally in XMI format to CDO and Neo4EMF, and we verify that all the imported models contain the same data.

Experiment I : Model traversal. In a first experimentation we execute a model visitor that starting from the root of the model traverses the full containment tree in a depth-first order. At each step of the traversal the visitor loads the element content from the backend, and modifies the element (changing its name). Only the standard EMF interface methods are used by the visitor, that is hence agnostic of which backend he is running on. During the traversal we measure the execution times for covering 0.1%, 1%, 10% 50% and 100% of the model. Fig. 5 shows the results of this experimentation over the two largest test models (org.eclipse.jdt.core and org.eclipse.jdt.*).

Experiment II : Java reverse engineering. In a second experimentation we execute a set of three simple queries on the Java metamodel that originate

Table 1: Overview of the experimental sets

#	Plugin	Size	Number of elements
1	org.eclipse.emf.ecore	24.2MB	121.295
2	org.eclipse.jdt.core	420.6MB	1.557.007
3	org.eclipse.jdt.*	984.7MB	3.609.454

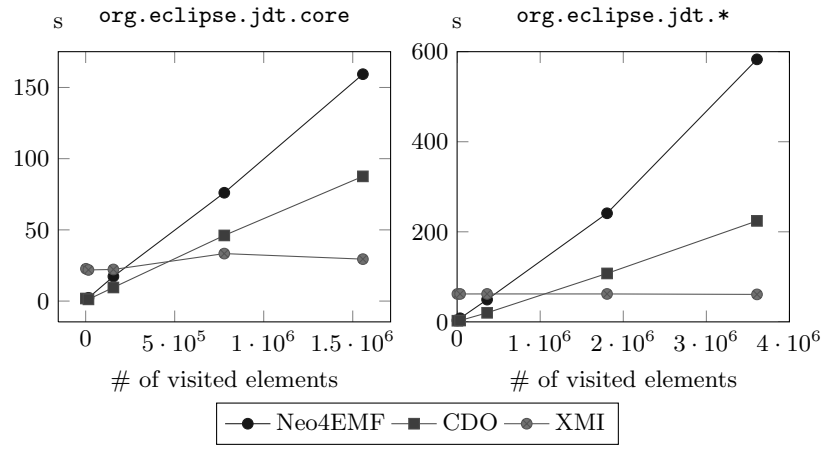

Fig. 5: Results for model traversal on test models 2 and 3.

from the domain of reverse-engineering Java code. While the first of these queries is a well-known scenario in academic literature, the other two have been selected to mimic typical model access patterns in reverse engineering, according to the experience of our industrial partner.

1. Grabats (GB): it returns the set of classes that holds static method declarations having as return type the holding class (e. g., Singleton) [15].
2. Unused Method Declarations (UnM): it returns the set of method declarations that are private and not internally called.
3. Class-Attribute Map (CA-M): it returns a map associating each Class declaration to the set of its attribute declarations.

All these queries start their computation by accessing the list of all the instances of a particular element type, then apply a filtering to this list to

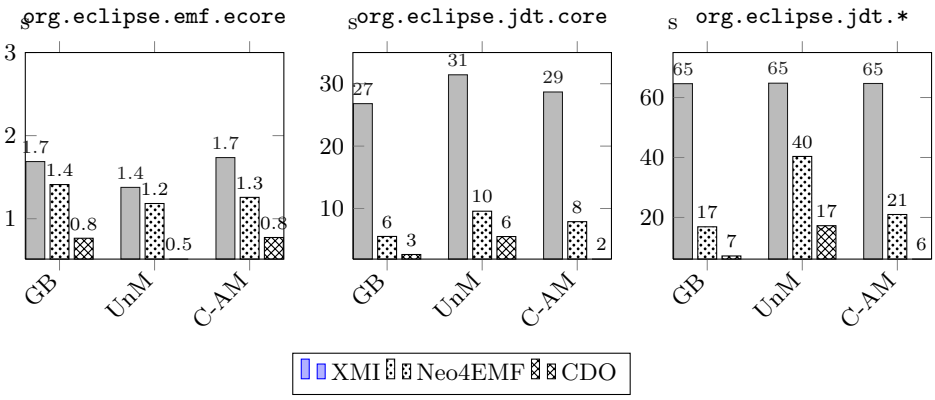

Fig. 6: Results for scenario 2

select the starting points for navigating the model. In the experience of our industrial partner this pattern covers the quasi-totality of computational-demanding queries in the reverse-engineering domain. For this reason we added a method *getAllInstances* to the EMF API and we implemented it in all the three back-ends. In CDO we implemented this method by a native SQL query, achieved through the union of the tables containing elements of the given type and its subtypes. In Neo4EMF the same result is achieved by a native Neo4j query traversing the graph nodes via the relationship `INSTANCE_OF`, for the given type and all of its subtypes. The user-code of each of the three queries uses this method to start the computation in all implementation, hence remaining backend-agnostic. Fig. 6 illustrates the result of the experimentation.

4.2 Discussion

The results of the two experimentations are consistent with each other. Fig. 5 shows that while in XMI the access time to each model element is trascurable with respect to the initial model-loading time, the two backends with on-demand loading mechanisms have a constant access time (giving linear complexity to the query). This shows that the backends can scale well for even larger sizes. In both experiments in Fig. 5 the backends with on-demand loading mechanisms outperform XMI when the part of the model that needs to be accessed is lower than a certain ratio of the full model. The graphs show that this ratio is approximately constant, independently of the size of the model and it amounts to 14.12% and 12.46% for Neo4EMF and 29.54% and 27.84%. for CDO. The CDO backend performs better than Neo4EMF, by an approximately constant factor that in the two experiments is respectively of 1.38 and 2.6.

The results from Fig. 6 show that both Neo4EMF and CDO outperform XMI. The test also confirms the previous result, showing execution times from CDO consistently lower than Neo4EMF.

Summarizing, while resulting a better solution than XMI for the industrial use case under study, the current version of Neo4EMF does not exhibit the performance optimizations in caching and prefetching of more mature solutions like CDO. CDO has two complementary ways of caching, one of CDOObjects placed on the client side, and two other caches maintaining CDORevisions (through the revision manager). Moreover CDO supports partial collection loading that gives the possibility to manage the number of elements to be loaded when an elements is fetched for the first time. Likewise, CDO provides a mechanism to decide how and when fetching the target objects asynchronously.

We also remark that the acceptable performances of XMI may be misleading in a real-world scenario: the amount of memory we used allowed to load the whole models in memory, avoiding any swapping in secondary memory that would have made the XMI solution completely unusable for the scenario. Moreover the use of an SSD hard disk significantly improved the loading & saving times from file. On-demand loading allows to use only the necessary amount of primary memory, extending the applicability of MDE tools to these large scenarios.

We did not measure significant differences in memory occupation between CDO and Neo4EMF, but we noticed several problems in importing large models in CDO. For instance CDO failed to import the test model 3 from its initial XMI serialization on a 8Go machine, as a `TimeOutException` was raised.

Finally, the comparison with relational databases backend should also take into account several other features, besides execution time and memory in a single processor configuration. Neo4EMF allows existing MDE tools to make use from now of the characteristics of graph databases like Neo4j, including clustering, online backups and advanced monitoring.

5 Related work

The interest on scalable model persistence has grown significantly in recent years, especially with the advent of new solutions for Model-Driven Reverse Engineering (MDRE) and Software Modernization (MDSM). Tools built on top of the EMF, such as MoDisco [6,17,26] have shown that models obtained from reverse-engineering processes can normally be composed of millions of elements [15]. Existing approaches are not suitable to manage this kind of artifacts both in terms of processing and memory consumption requirements.

CDO is the *de facto* standard solution to handle large models in EMF by storing them in a relational database. However, different experiences have shown that CDO does not scale well to very large models [21,22,24]. Barmpis and Kolovos [4] suggest that NoSQL databases would provide better scalability and performance than relational databases due to the interconnected nature of models.

Morsa [21] was one of the first approaches to provide persistence of large scale EMF models using NoSQL databases. As Neo4EMF, Morsa is based on a NoSQL database. Specifically, Morsa uses MongoDB, a document-oriented database, as its persistence backend. Morsa can be used seamlessly to persist large models using the standard EMF mechanisms. As CDO, it was built using a client-server architecture. Morsa provides on-demand loading capabilities together with incremental updates to maintain a low workload. Performance of the storage backend and their own query language (MorsaQL) has been reported in [21] and [22]. Neo4EMF is similar to Morsa in several aspects (notably in on-demand loading) but it aims at exploiting the optimized navigation performance offered by graph-databases w.r.t. document-oriented databases.

Mongo EMF [7] is another alternative to store EMF models in MongoDB databases. Mongo EMF provides the same standard API than previous approaches. However, according to the documentation, the storage mechanism behaves slightly different than the standard persistence backend (for example, for persisting collections of objects or saving bi-directional cross-document containment references). For this reason, Mongo EMF cannot be used without performing any modification to replace another backend in an existing system.

EMF fragments [16] is another NoSQL-based persistence layer for EMF aimed at achieving fast storage of new data and fast navigation of persisted models. Supported backends are MongoDB, Apache Hbase and regular files on

the file system. EMF fragments principles are simpler than in other similar approaches and those principles are based on the proxy mechanism used by EMF for inter-document relationships. In EMF fragments, models are automatically partitioned in several chunks (fragments). Unlike Neo4EMF, CDO, and Morsa, all data from a single fragment is loaded at a time, and only links to another fragments are loaded on demand. Another difference with other approaches is that artifacts should be specifically adapted: metamodels have to be modified to indicate where the partitions should be made to get the partitioning capabilities. While our approach has the advantage of not requiring metamodel-specific user manipulation or tool adaptation, fragmentation may provide performance benefits that we plan to investigate in future versions of Neo4EMF.

6 Conclusions and future work

In this paper we present the first version of Neo4EMF, a tool that can improve the applicability of MDE to large-scale scenarios, where on-demand loading, high-performance access and enterprise-level data-management features are needed. Our preliminary experimentation shows that, while Neo4EMF is a beneficial alternative to XMI for these scenarios, its raw performances do not surpass a more mature solution like CDO.

In our future work we plan to improve the tool by implementing performance optimization strategies, starting from a definition of model partitions, i.e., elements that are loaded in a single transaction, to reduce the total number of transactions during execution. We then plan to study the problem of memory unloading, by deriving unloading strategy from a definition of the possible uses of the persisted model. Finally we want to extend the applicability of Neo4EMF to other graph databases by exploiting recent proposals of common APIs among graph-databases [1], making of Neo4EMF a generic graph-database backend like CDO is for relational databases.

References

1. Blueprints, 2014. URL: `https://github.com/tinkerpop/blueprints/wiki`.
2. S. W. Ambler. The design of a robust persistence layer for relational databases. Technical report, 2000.
3. AtlanMod. Neo4EMF, 2014. URL: `http://www.neo4emf.com/`.
4. K. Barmpis and D. 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.
5. L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. 2013.
6. H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. Modisco: A generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM.
7. Bryan Hunt. MongoEMF, 2014. URL: `https://github.com/BryanHunt/mongo-emf/wiki/`.

8. Eclipse Foundation. CDO / DB Store, 2014. URL: http://wiki.eclipse.org/CDO/DB_Store/.
9. Eclipse Foundation. CDO / Hibernate Store, 2014. URL: http://wiki.eclipse.org/CDO/Hibernate_Store/.
10. Eclipse Foundation. CDO / MongoDB Store, 2014. URL: http://wiki.eclipse.org/CDO/MongoDB_Store/.
11. Eclipse Foundation. CDO / Objectivity Store, 2014. URL: http://wiki.eclipse.org/CDO/Objectivity_Store/.
12. Eclipse Foundation. Eclipse Modeling Framework Project (EMF), 2014. URL: http://www.eclipse.org/modeling/emf/.
13. Eclipse Foundation. The CDO Model Repository (CDO), 2014. URL: http://www.eclipse.org/cdo/.
14. INRIA and LINA. ATLAS transformation language, 2014.
15. F. Jouault, J. Sottet, et al. An AmmA/ATL Solution for the GraBaTs 2009 Reverse Engineering Case Study. In *Grabats 2009 5th International Workshop on Graph-Based Tools, Zurich, Switzerland (July 2009)*, 2009.
16. Markus Scheidgen. EMF fragments, 2014. URL: https://github.com/markus1978/emf-fragments/wiki/.
17. Modeliosoft Solutions, 2014. URL: http://www.modeliosoft.com/.
18. J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire. Acceleo user guide, 2006.
19. OMG. MOF 2.0 QVT final adopted specification (ptc/05-11-01), April 2008.
20. OMG. OMG MOF 2 XMI Mapping Specification version 2.4.1, August 2011.
21. J. E. Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: A scalable approach for persisting and accessing large models. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 77–92, Berlin, Heidelberg, 2011. Springer-Verlag.
22. J. E. Pagán and J. G. Molina. Querying large models efficiently. *Information and Software Technology*, 2014. In press, accepted manuscript. URL: http://dx.doi.org/10.1016/j.infsof.2014.01.005.
23. J. Partner, A. Vukotic, and N. Watt. *Neo4j in Action*. O'Reilly Media, 2013.
24. M. Scheidgen, A. Zubow, J. Fischer, and T. Kolbe. Automated and transparent model fragmentation for persisting large models. In R. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 102–118. Springer Berlin Heidelberg, 2012.
25. M. Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4):10–11, 2010.
26. The Eclipse Foundation. MoDisco Eclipse Project, 2014. URL: http://www.eclipse.org/MoDisco/.