Improving Memory Efficiency for Processing Large-Scale Models

Gwendal Daniel
AtlanMod team (Inria, Mines Nantes, LINA)
gwendal.daniel@etu.univ-nantes.fr

Gerson Sunyé
AtlanMod team (Inria, Mines Nantes, LINA)
gerson.sunye@inria.fr

Amine Benelallam
AtlanMod team (Inria, Mines Nantes, LINA)
amine.benelallam@inria.fr

Massimo Tisi
AtlanMod team (Inria, Mines Nantes, LINA)
massimo.tisi@inria.fr

ABSTRACT

Scalability is a main obstacle for applying Model-Driven Engineering to reverse engineering, or to any other activity manipulating large models. Existing solutions to persist and query large models are currently inefficient and strongly linked to memory availability. In this paper, we propose a memory unload strategy for Neo4EMF, a persistence layer built on top of the Eclipse Modeling Framework and based on a Neo4j database backend. Our solution allows us to partially unload a model during the execution of a query by using a periodical dirty saving mechanism and transparent reloading. Our experiments show that this approach enables to query large models in a restricted amount of memory with an acceptable performance.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Performance, Algorithms

Keywords

Scalability, Large models, Memory footprint

1. INTRODUCTION

The Eclipse Modeling Framework (EMF) is the *de facto* standard for the Model Driven Engineering (MDE) community. This framework provides a common base for multiple purposes and associated tools: code generation [4, 12], model transformation [9, 13], and reverse engineering [17, 6, 5].

These tools handle complex and large-scale models when manipulating important applications, for example, during reverse-engineering or software modernization through model transformation. EMF was first designed to support modeling tools and has shown limitations in handling large models. A more efficient persistence solution is needed to allow for partial model loading and unloading, which are key points when dealing with large models.

While several solutions to persist EMF models exist, most of them do not allow partial model unloading and cannot handle models that exceed the available memory. Furthermore, these solutions do not take advantage of the graph nature of the models: most of them rely on relational databases, which are not fully adapted to store and query graphs.

Neo4EMF [3] is a persistence layer for EMF that relies on a graph database and implements an unloading mechanism. In this paper, we present a strategy to optimize the memory footprint of Neo4EMF. To evaluate this strategy, we perform a set of queries on Neo4EMF and compare them against two other persistence mechanisms, XMI and CDO. We measure performances in terms of memory consumption and execution time.

The paper is organized as follows: Section 2 presents the background and the motivations for our unloading strategy. Section 3 describes our strategy and its main concepts: *dirty saving*, *unloading*, and *extended on-demand loading*. Section 4 evaluates the performance of our persistence layer. Section 5 compares our approach with existing solutions and finally, Section 6 concludes and draws the future perspectives of the tool.

2. BACKGROUND

2.1 EMF Persistence

As many other modeling tools, EMF has adopted XMI as its default serialization format. This XML-based representation has the advantage to be human readable, but has two drawbacks: (i) XMI sacrifices compactness for an understandable output and (ii) XMI files have to be entirely parsed to get a readable and navigational model. The former drawback reduces efficiency of I/O access, while the latter

increases the memory needed to load a model and limits on-demand loading and proxy uses between files. XMI does not provide advanced features such as model versioning or concurrent modifications.

The CDO [8] model repository was built to solve those problems. It was designed as a framework to manage large models in a collaborative environment with a small memory footprint. CDO relies on a client-server architecture supporting transactional accesses and notifications. CDO servers are built on top of several persistence solutions, but in practice only relational databases are used to store CDO objects.

2.2 Graph Databases

Graph databases are one of the NoSQL data models that have emerged to overcome the limitations of relational databases with respect to scale and distribution. NoSQL databases do not ensure ACID properties, but in return, they are able to handle efficiently large-scale data in a distributed environment.

Graph databases are based on nodes, edges, and properties. This particular data representation fits exactly to EMF models, which are intrinsically graphs (each object can be seen as a node and references as edges). Thus, graph databases can store EMF models without a complex serialization process.

3. NEO4EMF

Neo4EMF is a persistence layer built on top of the EMF framework that aims at handling large-models in a scalable way. It provides a compatible EMF API and a graph-database persistence backend based on Neo4j [16].
Neo4EMF is open source and distributed under the terms of the (A)GPLv3 [1].

In previous work [3], we introduced the basic concepts of Neo4EMF : *model change tracking* and *on-demand loading*. Model change tracking is based on a global changelog that stores the modifications done on a model during an execution (from creation to save). Tracking the modifications is done using EMF notification facilities: the changelog acts as a listener for all the objects and creates its entries from the received notifications. Neo4EMF uses an on-demand loading mechanism to load object fields only when they are accessed. Technically, each Neo4EMF object is instantiated as an empty container. When one of its fields (**EReferences** and **EAttributes**) is accessed, the associated content is loaded. This mechanism presents two advantages: (i) the entire model does not have to be loaded at once and (ii) unused elements are not loaded.

Neo4EMF does not use the **EStore** mechanism. Indeed, **EStore** allows the **EObject** data storage to be changed by providing a stateless object that translates model modifications and accesses into backend calls. Every generated accessor and modifier delegates to the reflexive API. As a consequence, **EObjects** have to fetch through the store each time a field is requested, engendering several database queries. On the contrary, Neo4EMF is based on regular **EObjects** (with in-memory fields) which are synchronized with a database backend.

In this paper we focus on Neo4EMF memory footprint. We introduce a strategy to unload some parts of a processed model and save memory during a query execution. In the previous implementation, the on-demand loading mechanism allows us to load only the parts of the model that are needed, but there is no solution to remove unneeded objects from memory, especially when they were changed but not saved yet.

A reliable unload strategy needs to address two main issues:

- **Accessibility:** Contents of unloaded objects (attributes and referenced objects) have to remain accessible through standard EMF accessors.
- **Transparency:** The management of the object life cycle has to be independent from users, but customizable to fit specific needs, e. g., size of the Java virtual machine, requirements on execution time, etc.

Our strategy faces these issues by providing a *dirty-saving* mechanism, which provides temporary and transparent model persistence. The object life cycle has also been modified to include unloading of persisted elements.

In this next sections, we provide an overview of the changelog used to record the modifications of the processed model. Then, we present *dirty saving*, based on the basic Neo4EMF save mechanism, and we describe the Neo4EMF object life cycle. Finally, we describe the modifications done on the *on-demand loading* feature to handle this new strategy.

3.1 Neo4EMF Changelog

Neo4EMF needs a mechanism to ensure synchronization between the in-memory model and its backend representation, avoiding systematic unnecessary calls to the database.

Despite the existence in EMF of a modification tracking mechanism, the **ChangeRecorder** class, we decided to develop an alternative solution that minimizes memory consumption.

Neo4EMF tracks model modifications in a changelog, a sequence of entries of five types:

Object creation: A new object has been created and attached to a Neo4EMF resource.

Object deletion: An object has been deleted or removed from a Neo4EMF resource.

Attribute modifications: Attribute setting and unsetting.

Reference addition: Assignment of a new single-valued reference or addition of a new referenced object in a multi-valued one.

Reference deletion: Unsetting a single-valued reference or removing a referenced object in a multi-valued one.

We distinguish *unidirectional* and *bidirectional* reference modifications for performance reasons (they are not serialized the

same way during the saving process).
Figure 1 summarizes our changelog model. All changelog entries are subclasses of **Entry**, which defines some shared properties: the object concerned by the modification (for instance the object containing a modified attribute or reference, or the new object in case of a CreateObject entry) and a basic serialization method.

Attribute and reference modification entries (**SetAttribute**, **AddLink**, **RemoveLink** and their subclasses) have three additional fields to track fine-grained modifications: the updated feature (attribute or reference identifier) which corresponds to the modified field of the concerned object, the new and old values of the feature (if available).

This decomposition provides a direct access to the information required during the serialization process, without accessing the concerned objects. The fine-grained entry management also decreases memory consumption. For instance modifications on bidirectional references correspond to a single changelog entry, while they needed two basic entries before. Serialization of those entries is also more efficient since it reduces the number of database accesses.

In the previous version of Neo4EMF, we used the EMF notification framework to create changelog entries. This implementation had a major drawback: notifications were handled in a dedicated thread, and we could not ensure that all the notifications were sent to the changelog before its serialization. This behavior could create an inconsistency between the in-memory model and the saved one. This is another reason we do not use the EMF **ChangeRecorder** facilities, which relies on notifications.

In this new version, changelog entries are directly created into the body of the generated methods. This solution removes synchronization issues and is also more efficient, because entries are created directly, and all the information needed to construct them is available in the method body (current object, feature identifier, new and old values). We also do not have to deal with the generic notification API, which was resulting in a lot of casts and complex processing to retrieve this information. Synchronizing the changelog brings another important benefit: the causality between model modifications and entries order is ensured and there is no need to reorder the entry stack before its serialization.

Finally, we modify the changelog life cycle. In the previous version, the changelog was a global singleton object, containing the record of a full execution, mixing modifications of multiple resources. This solution is not optimal because saving is done per resource in EMF, and to save a single resource the entire modification stack needed to be processed to retrieve the corresponding entries. We choose to create a dedicated changelog into each Neo4EMF resource that handles modifications only for the objects contained in the associated resource. This modification reduces the complexity of the save processing: the resource changelog is simply iterated and its entries are then serialized into database calls. The synchronized aspect of the changelog allows us to process the entries in the order they are added, which was not possible in the previous version.
Furthermore, associating a changelog with a resource ensures that, when the resource is deleted, all the related entries are also deleted. In the previous version, entries could not be deleted from the global changelog, and were kept in memory during the execution.

Figure 2: Excerpt of MoDisco Java Metamodel

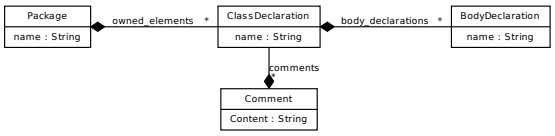
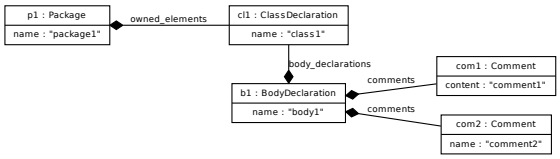


Figure 3: Sample instance of Java Metamodel



3.2 Dirty Saving

Neo4EMF relies on a mapping between EMF entities and Neo4j concepts to save its modifications. Figure 2 shows an excerpt of the Java metamodel, used in the MoDisco [17] project. This metamodel describes Java applications in terms of **Packages**, **ClassDeclarations**, **BodyDeclarations**, and **Comments**. A **Package** is a named container that gathers a set of **ClassDeclarations** through its **owned_elements** composition. A **ClassDeclaration** is composed of a name, a set of **Comments** and a set of **BodyDeclarations**.
Figure 3 shows a simple instance of this metamodel: a **Package** (package1), containing one **ClassDeclaration**, (class1). This *ClassDeclaration* contains two **Comments** (comment1 and comment2) and one single **BodyDeclaration** (body1). Figures 2, 3, and 4 show that:

Model elements are represented as nodes. Nodes with identifier `p1`, `cl1`, `b1`, and `com1` are examples corresponding to `p1`, `cl1`, `b1`, and `com1` in Figure 3. The ROOT node represents the entry point of the model (the resource directly or indirectly containing all the other elements) and is not associated to a model object.

Elements attributes are represented as node properties. Node properties are $\langle name, value \rangle$ pairs, where *name* is the feature identifier and *value* the value of the feature. Node properties can be observed for `p1`, `cl1`, and `b1`.

Metamodel elements are also represented as nodes and are indexed to facilitate their access. Metamodel nodes have two properties: the metaclass name and the metamodel unique identifier. `P`, `Cl`, `B` and `Com` are examples of metamodel element nodes, they correspond to **PackageDeclaration**, **ClassDeclaration**, **BodyDeclaration**, and **Comment**, respectively in Figure 2

Figure 1: Changelog Metamodel

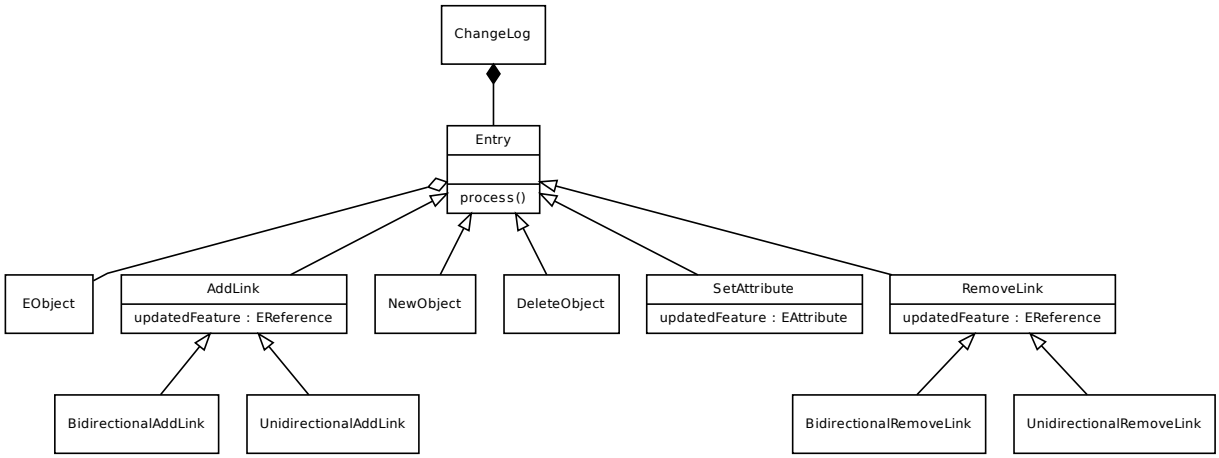
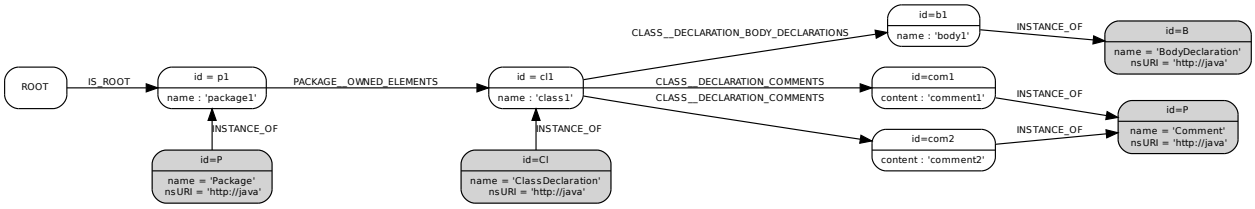


Figure 4: Sample instance database representation



InstanceOf relationships are outgoing relationships between the elements nodes and the nodes representing metaclasses. They represent the conformance of an object instance to its class definition

References between objects are represented as relationships. To avoid naming conflicts relationships are named using the following convention:
CLASS_NAME__REFERENCE_NAME.

When a save is requested, changelog entries are processed to update the database backend. Each entry is serialized into a database operation. The **CreateObject** entry corresponds to the creation of a new node and its meta-information (INSTANCEOF to its meta-class, ISROOT if the object is directly contained in the resource). All the fields of the object are also serialized and directly saved in the database. A **SetAttribute** entry corresponds to an update of the related node's property with the corresponding name. **AddLink**, **RemoveLink**, and their subclasses respectively record the creation and removal of a relationship, storing the containing **class** and **feature** name.

We decide to serialize at the same time a created object and all its references and attributes. New objects need to be entirely persisted, and there is no reason to record their modifications before their first serialization (the final state of the object is the one that needs to be persisted). This full serialization behavior has the advantage of generating only one single entry for a new object, independently from the number of its modified fields.

This approach works well for small models, but has issues when a large modification set needs to be persisted: the changelog grows indefinitely until the user decides to save it. This is typically the case in reverse engineering, where the extracted objects are first all created in memory and only afterwards they are saved.

To address this problem we introduce *dirty-saving*, a periodical save action not requested by the user. The period is determined by the changelog size, configurable through the Neo4EMF resource. Since these save operations are not requested by the user they have to ensure two properties:

- **Reversibility:** if the modifications are canceled or if the user does not want to save a session the database should rollback to an acceptable version. This version is either (i) the previous regularly saved database if an older version exists or (ii) an empty database.

- **Persistability:** if a regular save is requested by the user, the temporary objects in the database have to be definitely persisted. They can then constitute a new acceptable version of the database if a rollback is needed.

We introduce a new mapping for changelog entries with the purpose of temporary dirty saving. This mapping is based on the same entries as the regular mapping but the associated Neo4j concepts allow the system to easily extract dirty objects and regular ones. In addition we create two indexes: `tmp_relationships` and `tmp_nodes` which respectively contain the dirty relationships and nodes (i. e., created in a dirty saving session). Figure 5 summarizes the mapping between changelog entries and *neo4j* concepts:

- **CreateObject**: creation of a new node (as in the regular saving process) and addition to the `tmp_nodes` index.
- **SetAttribute**: creation of a dedicated node containing the dirty attributes. The idea is to keep a stable version (i. e., the previous regularly saved version) to easily reverse it. A **SetAttribute** relationship is created to link the base object and its attribute node
- **AddLink**: creation of a generic **AddLink** relationship, containing the reference identifier as a property. This special relationship format is needed to easily process dirty relationships and retrieve their corresponding image if a regular save operation is requested
- **RemoveLink**: creation of a generic **RemoveLink** relationship, containing the reference identifier as a property. **AddLink** and **RemoveLink** relationships with the same reference identifier and target object are mutually exclusive to limit the number of temporary objects into the database
- **DeleteObject**: creation of a special **Delete** relationship looping on the related node. The base version of the node is kept alive if a rollback is needed.

The objective of this mapping is to preserve all the information contained after a regular save, to easily handle a rollback. That is why object deletion is done using a relationship: if the modifications are aborted it is simpler to remove the relationship than creating a new instance of the node with backup information. We do not use a property to tag deleted objects for performance reasons (access to node properties is slower than edge navigation).
To persist definitely dirty objects in the database into regularly saved ones a serialization process is invoked. As changelog entries, each Neo4j element contains all the information needed to create their regular equivalents: new objects are simply removed from the `tmp_nodes` index, **AddLink** relationships are turned into their regular version using their properties and **RemoveLink** entries correspond to the deletion of their existing regular version.

For example if we update the model given in Figure 3 by removing `com1` and creating a new **BodyDeclaration** `body2` then calling a dirty save, the database will be updated as in Figure 6. Note that a **Delete** relationship has been created because the removed **Comment** is not contained in the resource anymore. Red relationships and nodes are indexed respectively in `tmp_relationships` and `tmp_nodes` indexes.

This example shows that our mapping is built on top of the existing one: there is no modification done on the previous version, represented with black nodes. This simplifies the rollback process, which consists of a deletion of all the temporary Neo4j objects.

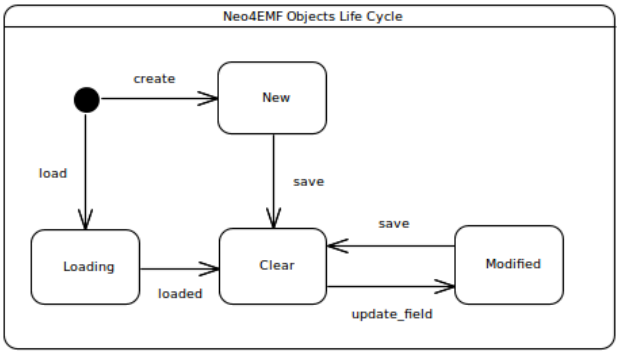
3.3 Object Life Cycle

We modify the Neo4EMF object life cycle to enable unloading. When a *dirty saving* is invoked, all the modifications contained in the changelog are committed to the database. Because of this persistence, persisted objects can be safely released from memory and reloaded using *on-demand loading*, if needed.

Figure 7 shows the different life cycle states of a Neo4EMF object. When a Neo4EMF object is created it is **New**: it has not been persisted into the database and cannot be released. When a save is requested or a dirty save is invoked, the new object is persisted into the database and it is tagged as **Clear**: all the known modifications related to the object have been saved and it is fetchable from the database without information loss. In this state the object can be removed from memory without consistency issues. When a modification is done on the object (setting an attribute or updating a reference) then it is tagged as **Modified**.

Modified objects cannot be released, because their database-mapped nodes do not contain the modified information. When a save is processed, the **Modified** objects revert to **Clear** state and can be released again. **Loading** objects also have a particular state that avoids garbage collection of an object when it is loading.

Figure 7: Neo4EMF EObject life cycle



To allow garbage collection of Neo4EMF objects, we use Java Soft and Weak references to store object's fields. Weak and Soft referenced objects are eligible for garbage collection as soon as there is no strong reference chain on them. The

Figure 5: Changelog to Neo4j entity mapping

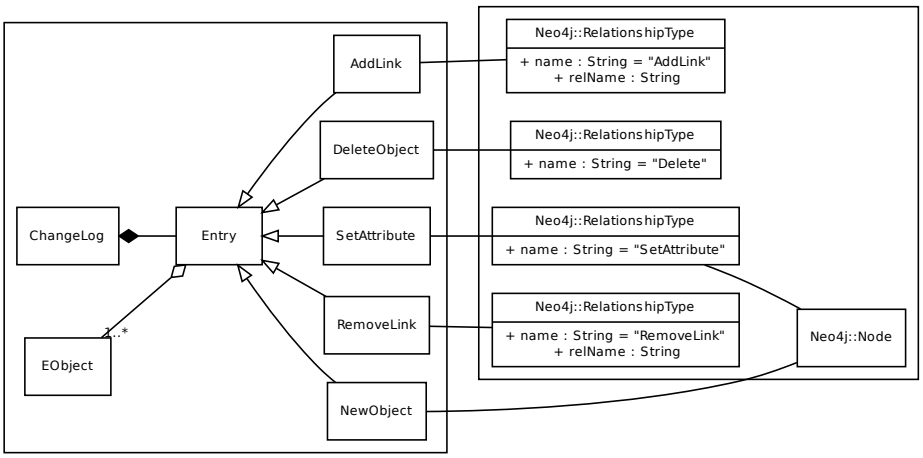
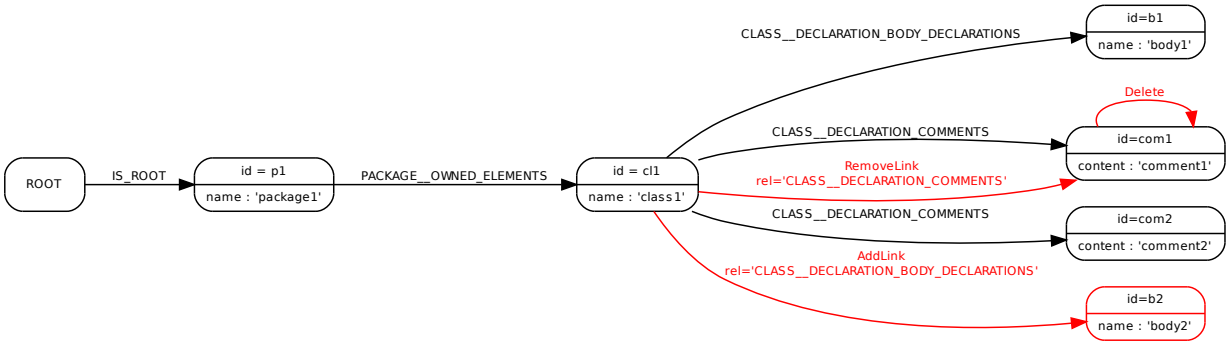


Figure 6: Database state after modifications



difference between the two kinds of references is the time they can remain in memory. Weak references are collected as soon as possible by the garbage collector, whereas Soft references can be retained in memory as long as the garbage collector does not need to free them (i.e., as long as there is enough available memory). This particular behavior is interesting for cache implementation and to optimize execution speed in a large available memory context. Reference type (Weak or Soft) can be set through Neo4EMF resource parameters.

In Section 3.1, we describe that changelog entries contain all the information related to the serialization of the concerned object. This information constitutes the strong reference chain on the related object fields. When a save is done, entries are processed and deleted, breaking the strong reference chain and making objects eligible for garbage collection.

Neo4j's objects are not impacted by this new life-cycle. The database manages its objects life cycle through a policy defined at the resource creation (memory or performance preferences).

3.4 Extended On-Demand Loading

To handle the new architecture of our layer, we have to extend the *on-demand loading* feature to support temporary persisted objects. On-demand loading uses two parameters: (i) the object that handles the feature to load and (ii) the identifier of the feature to load. This behavior implies that a Neo4EMF object is always loaded from another Neo4EMF object.

Figure 6 shows our Java metamodel instance state after a dirty save. The database content is a mix between regularly saved objects (in black) and dirty-saved ones (in red). Loading referenced **Comments** instances from **ClassDeclaration** `cl1` is done in three steps to ensure the last dirty-saved

operations have been considered.
First, CLASS_DECLARATION__COMMENTS relationships are processed and their end nodes are saved. Second, the **AddLink** relationships containing the corresponding **rel** property are processed and their end nodes are added to the previous ones. This operation retrieves all the associated nodes for the given feature, regular ones and dirty ones. Third, **RemoveLink** relationships are processed the same way and their end nodes are removed from the loaded node set.

Attribute fetching behavior is a bit different: if a node representing an object has relationships to a dedicated attribute node, then the data contained in this node is returned instead of the base node property.

To improve the performances of our layer, we create a cache that maps Neo4j identifiers to their associated object. When *on-demand loading* is performed, the cache is checked first, avoiding the cost of a database access. This cache is also used to retrieve released objects.

4. EVALUATION

In this section, we evaluate how the memory footprint and the access time of Neo4EMF scale in different large model scenarios, and we compare it against CDO and XMI. These experiments are performed over two EMF model extracted with the MoDisco Java Discoverer [17]. Both models are extracted from Eclipse plug-ins: the first one is an internal tool and the second one is the Eclipse *JDT* plugin. The resulting XMI files are 20 MB and 420 MB, containing respectively around 80 000 and 1 700 000 elements.

4.1 Execution Environment

Experiments are executed on a computer running Windows 7 professional edition 64 bits. Interesting hardware elements are: an Intel Core I5 processor 3350P (3.5 GHz), 8 GB of DDR3 SDRAM (1600 MHz) and a Seagate barracuda 7200.14 hard disk (6 GB/s). Experiments are executed on Eclipse 4.3 running Java SE Runtime Environment 1.8.

To compare the three persistence solutions, we generate three different EMF models from the MoDisco Java Metamodel: (i) the standard EMF model, (ii) the CDO one and (iii) the Neo4EMF one. We import both models from XMI to CDO and Neo4EMF and we verify they contain the same data after the import.

Neo4EMF uses an embedded Neo4j database to store its objects. To provide a meaningful comparison in term of memory consumption we choose to use an embedded CDO server.

Experiment 1: Object creation. In this first experiment, we execute an infinite loop of object creation and simply count how many objects have been created before a **OutOfMemoryException** is thrown. We choose a simple tree structure of three classes to instantiate from the MoDisco Java metamodel: a parent **ClassFile** containing 1000 **BlockComment** and **ImportDeclaration**. The resulting model is a set of independent element trees. For this experiments we choose a 1 GB Java virtual machine and an arbitrarily fixed changelog size of 100 000 entries. Table 1 summarizes the results.

Persistence Layer	XMI	CDO	Neo4EMF
#Created Elements	22 939 780	4 378 990	>40 000 000[1]

Table 1: Number of Created Elements Before Memory Overhead

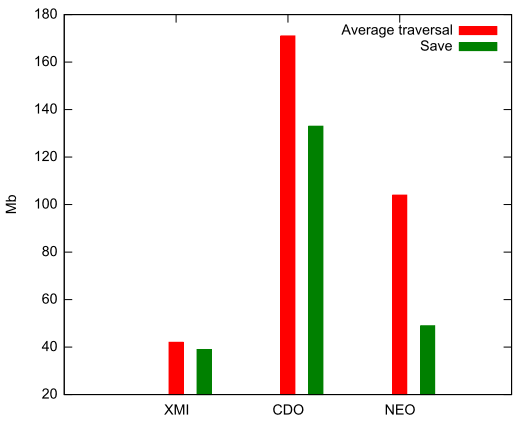


Figure 8: Memory Consumption: Model Traversal and Save (20 MB)

Note that the number given for Neo4EMF is an approximation: we stop the execution before any **OutOfMemory** error. The average memory used to create elements was around 500 MB and does not seem to grow. This performance is due to the *dirty-saving* mechanism: created objects generate entries in the changelog. When the changelog is full, changes are saved temporarily in the database, freeing the changelog for next object creations.

Experiment 2: Model traversal. In this experiment, we load a model and execute a traversal query that starts from the root of the model, traverses all the containment tree and modifies the **name** attribute of all **NamedElements**. All the modifications are saved at the end of the execution. During the traversal, we measure the execution time for covering the entire model and the average memory used to perform the query. In addition, we measure the memory needed to save the modifications at the end of the execution. Figures 8 and 9 summarize memory results. As expected, the Neo4EMF traversal footprint is higher than the XMI one because we include the Neo4j embedded database and runtime in our measures. Unloading brings a real interest when comparing the results with CDO: when removing unused (i. e., unreferenced) objects we save space and process the request in a reduced amount of memory. For this experiment we use a 4 GB Java virtual machine, with the `ConcMarkSweepGC` garbage collector, recommended when using Neo4j.

Experiment 3: Time performance. This experiment is similar to the previous one, but we focus on time performances. We measure the time needed to perform traversal and save. Figures 10 and 11 summarize the results. To provide a fair comparison between full and on-demand loading strategies we also include model loading time with the traversal queries.

[1]The execution was stopped before any memory exception.

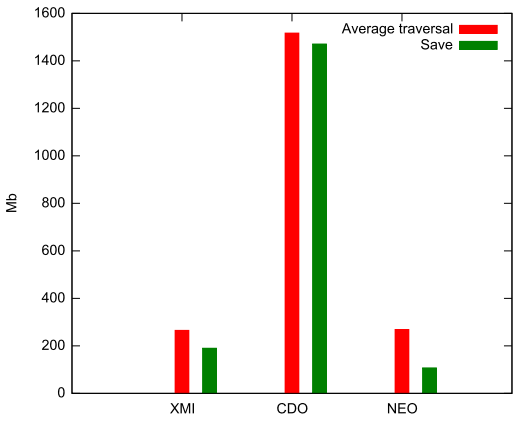


Figure 9: Memory Consumption: Model Traversal and Save (420 MB)

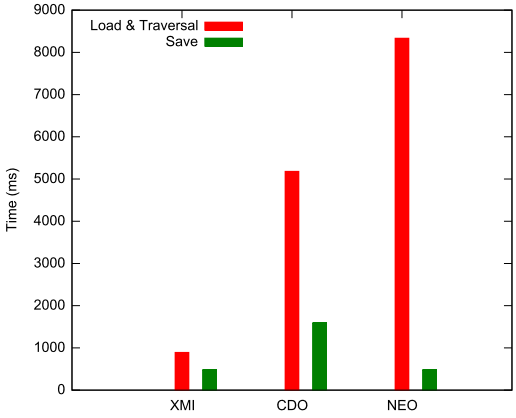


Figure 10: 20 MB **model traversal and save performances**

Neo4EMF save performances can be explained with *dirty-saving*: during the traversal, entries are generated to track the *name* modifications. These entries are then saved in the database when the changelog is full, reducing the final save cost. This behavior also explains a part of the traversal time overhead, when compared to CDO: Neo4EMF traversal implies database write access for *dirty saving* where CDO does not, related I/O accesses considerably impact performance.

4.2 Discussion

The results of these experiments show that dirty-saving coupled with on-demand loading decrease significantly the memory needed to execute a query. As expected, this memory footprint improvement worsens the time performances of our tool, in particular because of dirty-saving, which generates several database calls. That is why we provide *dirty saving* configuration through the Neo4EMF resource. The experiments also show that Neo4EMF is able to handle large queries and modifications in a limited amount of memory, compared to existing solutions.

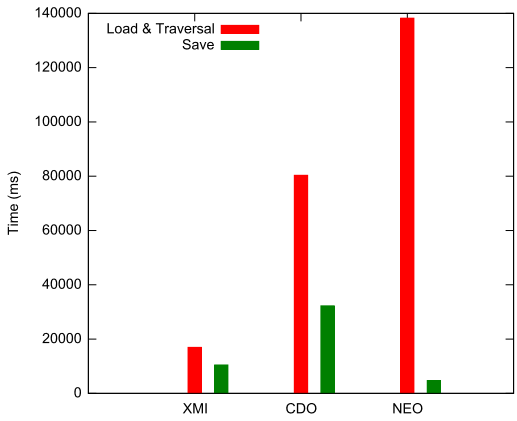


Figure 11: 420 MB **traversal and save performances**

We also run our benchmarks on different operating systems (Ubuntu 12.04 and 13.10) and we find that CDO and Neo4EMF time performances seem to be linked to the file partition format (especially in I/O accesses): Neo4j has better performances on these operating system (with a factor of 1.5) and CDO has slower times (with approximately the same factor). More investigation is needed to optimize our tool in different contexts.

Our experiments show that Neo4EMF is an interesting alternative to CDO to handle large models in memory constrained environment. On-demand loading and transparent unloading offer a small memory footprint (smaller than CDO in our experiments), but our solution does not provide advanced features like collaborative edition and versioning provided by CDO.

The unload strategy is transparent for the user, but may be intrusive in some cases, for instance if the hard-drive memory space is limited or the time performances are critical. This is why we introduce configuration for *dirty saving* and changelog size through the Neo4EMF resource.

5. RELATED WORK

Models obtained by reverse engineering with EMF-based tools such as MoDisco [17, 5, 11] can be composed of millions of elements. Existing solutions to handle this kind of models have shown clear limitations in terms of memory consumption and processing.

CDO is the *de facto* standard to handle large models using a server and a relational database. However, some experiments have shown that CDO does not scale well to very large models [2]. Pagán et al. [14, 15] propose to use NoSQL databases to store models, especially because those kind of databases should fit better to the interconnected nature of EMF models.

Mongo EMF [7] is a NoSQL approach that stores EMF models in MongoDB, a document-oriented database. However, Mongo EMF storage is different from the standard EMF persistence backend, and cannot be used *as is* to replace an other persistence solution in an existing system. Modifications on the client software are needed to integrate it.

Morsa [14] is an other persistence solution based on MongoDB database. Similarly to Neo4EMF, Morsa uses a standard EMF mechanism to ensure persistence, but it uses a client-server architecture, like CDO. Morsa has some similarities with Neo4EMF, notably in its *on-demand loading* mechanism, but does not use a graph database.

EMF Fragments [10] is another EMF persistence layer based on a NoSQL database. The EMF Fragments approach is different from other NoSQL persistence solutions: it relies on the proxy mechanism provided by EMF. Models are automatically partitioned and loading is performed by partition. Loading on demand is only performed for cross-partition references. Another difference with Neo4EMF is that EMF Fragments needs to annotate the metamodels to provide the partition set, whereas our approach does not require model adaptation or tool modification.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a strategy to optimize the memory footprint of Neo4EMF, a persistence layer designed to handle large models through *on-demand loading* and transparent unloading. Our experiments show that Neo4EMF is an interesting alternative to CDO for accessing and querying large models, especially in small available memory context, with a tolerable performance loss. Neo4EMF does not have collaborative model editing or model versioning features, which biases our results: providing those features may imply a more important memory consumption.

In future work, we plan to improve our layer by providing partial collection loading, allowing the loading of large collections subparts from the database. In our experiments, we detected some memory consumption overhead in this particular case: when an object contains a huge number of referenced objects (through the same reference) and they are all loaded at once.

We then plan to study the inclusion of attribute and reference meta-information directly in the database to avoid unnecessary object loading: some EMF mechanisms, like `isSet` may induce *load on demand* of the associated attribute, just in order to make a comparison. It could be interesting to provide this information from the database without a complete and costly object loading.

Finally, we want to introduce loading strategies such as prefetching or model partitioning (using optional metamodel annotations or a definition of the model usage) to allow users to customize the object life cycle.

7. REFERENCES

[1] AtlanMod. Neo4EMF, 2014. URL: `http://www.neo4emf.com/`.

[2] 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.

[3] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay. Neo4emf, a scalable persistence layer for emf models. July 2014.

[4] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. 2013.

[5] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot. Modisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012 – 1032, 2014.

[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. The CDO Model Repository (CDO), 2014. URL: `http://www.eclipse.org/cdo/`.

[9] INRIA and LINA. ATLAS transformation language, 2014.

[10] Markus Scheidgen. EMF fragments, 2014. URL: `https://github.com/markus1978/emf-fragments/wiki/`.

[11] Modeliosoft Solutions, 2014. URL: `http://www.modeliosoft.com/`.

[12] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire. Acceleo user guide, 2006.

[13] OMG. MOF 2.0 QVT final adopted specification (ptc/05-11-01), April 2008.

[14] 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.

[15] 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`.

[16] J. Partner, A. Vukotic, and N. Watt. *Neo4j in Action*. O'Reilly Media, 2013.

[17] The Eclipse Foundation. MoDisco Eclipse Project, 2014. URL: `http://www.eclipse.org/MoDisco/`.