Map-Based Transparent Persistence for Very Large Models

Abel Gómez, Massimo Tisi, Gerson Sunyé, Jordi Cabot

Map-based Transparent Persistence for Very Large Models

Abel Gómez, Massimo Tisi, Gerson Sunyé, and Jordi Cabot

AtlanMod team (Inria, Mines Nantes, LINA), France
{abel.gomez-llana|massimo.tisi|gerson.sunye|jordi.cabot}@inria.fr

Abstract. The progressive industrial adoption of Model-Driven Engineering (MDE) is fostering the development of large tool ecosystems like the Eclipse Modeling project. These tools are built on top of a set of base technologies that have been primarily designed for small-scale scenarios, where models are manually developed. In particular, efficient runtime manipulation for large-scale models is an under-studied problem and this is hampering the application of MDE to several industrial scenarios.
In this paper we introduce and evaluate a map-based persistence model for MDE tools. We use this model to build a transparent persistence layer for modeling tools, on top of a map-based database engine. The layer can be plugged into the Eclipse Modeling Framework, lowering execution times and memory consumption levels of other existing approaches. Empirical tests are performed based on a typical industrial scenario, model-driven reverse engineering, where very large software models originate from the analysis of massive code bases. The layer is freely distributed and can be immediately used for enhancing the scalability of any existing Eclipse Modeling tool.

Keywords: Model Driven Engineering, Model Persistence, Very Large Models, Key-Value Stores

1 Introduction

Part of the software industry is embracing the main concepts of Model-Driven Engineering, by putting models and code generation at the center of their software-engineering processes. Recent studies [22], as well as the proliferation of tools related to MDE, testify the increase in popularity of these concepts, which are applied to different contexts and scales. These scales vary from manual modeling activities with hundreds of model elements to very large models, VLMs, with millions of elements. Very large models are especially popular in specific domains such as the automotive industry [10], civil engineering [24], or software product lines [21], or are automatically generated during software modernization of large code bases.

Modeling tools are built around so-called modeling frameworks, that provide basic model-management functionalities and interoperability to the modeling ecosystem. Among the frameworks currently available, the Eclipse Modeling

Framework [7] (EMF) has become the *de facto* standard for building modeling tools. The *Eclipse marketplace* attests the popularity of EMF, counting more than two hundred EMF-based tools [6] coming from both industry and academia.

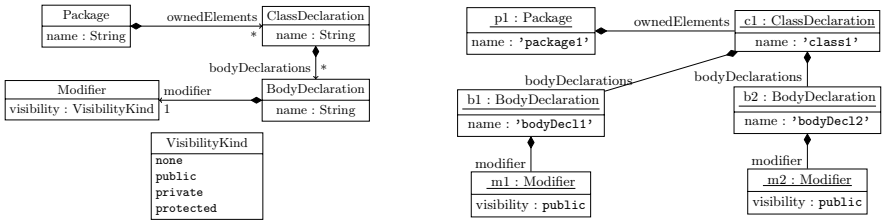
However, the technologies at the core of modeling frameworks were designed in the first place to support simple modeling activities and exhibit clear limits when applied to very large models. Problems in managing memory and persisting data while handling models of this size are under-studied and the current standard solution is to use a model/relational persistence layer (e.g., CDO for EMF [3]) that translates runtime model-handling operations into relational database queries. Existing solutions have shown clear performance limits in related work [19,20,23]. In this paper we propose a transparent persistence solution for very large models, that introduces the following innovations:

- The transparent persistence is designed to optimize runtime performance and memory occupancy of the atomic **low-level operations** on models. We argue that this strategy improves execution of model-driven tools on large models in real-world scenarios, without an *ad hoc* support from the tool (differently from similar proposals in related work [19, 20]);
- We propose a **map-based** persistence model for MDE tools, arguing that persisting model graphs directly as maps allows for faster runtime operation with respect to a more obvious graph-database persistence. In this sense this paper presents a novel and different approach completing our previous work [9], in which we built a persistence layer for EMF based on a graph database. We compare the different approaches and discuss the distinct application scenarios for each one.
- Persistence is built around a **database engine**, instead of interfacing with a full-fledged database, and directly manipulates low-level data structures. We argue that this approach (i) gives more flexibility when selecting the data structures that optimize model-specific operations, and (ii) reduces overhead by not requiring translation of model operations into a database query language.

Our persistence layer has been implemented as an open-source prototype[1]. The layer can be plugged into EMF-based tools to immediately provide enhanced support for VLMs. Experimental validation shows that (i) our layer allows handling models that cannot be handled by the standard file-based EMF backend—and even the CDO backend in configurations with little memory—and (ii) that queries perform (up to nine times) faster than the standard relational backend.

The paper is structured as follows: Section 2 motivates the paper by describing a running scenario and the limits of current model-persistence solutions. Section 3 provides an overview of our approach and its main properties. Section 4 describes our publicly-available persistence layer. Section 5 illustrates the experimental evaluation, Section 6 compares our approach with related work and Section 7 concludes the paper.

[1] `http://www.emn.fr/z-info/atlanmod/index.php/NeoEMF/Map`



(a) (b)

Fig. 1: Excerpt of the *Java* metamodel (1a) and sample instance (1b)

2 Motivation

2.1 Running example

As a running example for this paper we will consider an industrial scenario that uses modeling tools for helping a Java code reverse-engineering and modernization process. The first step in this scenario is parsing the legacy Java codebase (that can be massive) to produce a very detailed Java model. Such model will be then processed by other modeling tools like analyzers or transformers for computer-aided program understanding.

Fig. 1a shows a small excerpt of a possible *Java* metamodel. 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*.

Fig. 1b shows a sample instance of the *Java* metamodel, i.e., a graph of objects conforming with the metamodel structure. The model contains a single *Package* (`package1`), containing only one *ClassDeclaration* (`class1`). The *Class* contains the `bodyDecl1` and `bodyDecl2` *BodyDeclaration*s. Both of them are `public`. Similar instances in large reverse-engineering projects can contain millions of model elements describing the full system code.

Within a modeling ecosystem, all tools that need to access or manipulate models have to pass through a single model management interface. This includes all reverse-engineering, code analysis, and code visualization tools in our running scenario. In some of these ecosystems—as it is the case of EMF—the model management interface is automatically generated from the metamodel of the modeling language. For example, from the metamodel in Fig. 1a EMF produces an API that allows, e.g., to construct the sample instance of Fig. 1b by the code in Listing 1.1.

Without any specific memory-management solution, the model would need to be fully contained in memory for any access or modification. While this approach would be suitable for small models like the one in Fig. 1b, models that exceed the main memory would cause a significant performance drop or the application crash. A possible solution would be a transparent persistence layer in

the modeling framework, able to automatically persist, load and unload model elements with no changes to the application code (e.g., to Listing 1.1). In this paper we want to design an efficient layer for this task.

2.2 Persisting very large models

Along the history of MDE, several modeling frameworks have emerged (e.g., Eclipse EMF [7], Microsoft DSL Tools [12], MetaEdit+ [15], GME [16]), each providing a uniform interface to its correspondent ecosystem of modeling tools. Modeling frameworks share a similar, object-oriented conceptual representation of models (e.g., based on OMG's MOF for EMF and GME, on Object-Property-Role-Relationship for MetaEdit+). They differ in the design of the model-management interface and persistence mechanism.

Since the publication of the XMI standard [18], file-based XML serialization has been the preferred format for storing and sharing models and metamodels. The choice was suited to the fact that modeling frameworks have been originally designed to handle human-produced models, whose size does not cause significant performance concerns. However, XML-based serialization results to be inefficient for large models: (i) XML files sacrifice compactness in favor of human-readability and (ii) XML files need to be completely parsed to obtain a navigational model of their contents. The first factor reduces efficiency in I/O accesses, while the second increases the memory required to load and query models and it is an obstacle to on-demand loading. Moreover, XML-based implementations of model persistence require *ad hoc* implementations of advanced features such as concurrent modifications, model versioning, or access control.

The design of additional relational back-ends for model persistence helped solve the problem for medium-size models. For instance, CDO [3] is the standard solution for persisting EMF models where scalability is an issue. It implements a

Listing 1.1: Creation of the sample instance using the generated API (Java-like pseudocode)

```
// Creation of objects
Package p1 := Factory.createPackage();
ClassDeclaration c1 := Factory.createClassDeclaration();
BodyDeclaration b1 := Factory.createBodyDeclaration();
BodyDeclaration b2 := Factory.createBodyDeclaration();
Modifier m1 := Factory.createModifier();
Modifier m2 := Factory.createModifier();
// Initialization of attributes
p1.setName("package1");
c1.setName("class1");
b1.setName("bodyDecl1");
b2.setName("bodyDecl2");
m1.setVisibility(VisibilityKind.PUBLIC);
m2.setVisibility(VisibilityKind.PUBLIC);
// Initialization of references
p1.getOwnedElements().add(c1);
c1.getBodyDeclarations().add(b1);
c1.getBodyDeclarations().add(b2);
b1.setModifier(m1);
b2.setModifier(m2)
```

client-server architecture with on-demand loading, and transactional, versioning and notification facilities. Although in theory CDO is a generic framework [1, 2, 4, 5], only relational databases are regularly used and maintained[2] and different experiences have shown that CDO does not scale well with VLMs in such a commmon setup [19, 20, 23].

3 Scalable model-persistence layer

In this paper we investigate the problem of persisting very large models and design a solution that improves the state of the art in terms of runtime execution time, memory occupancy, extensibility and compatibility. More precisely, we propose to satisfy a set of requirements, that we consider as necessary for an effective solution for scalable model persistence.

We identify three **interoperability requirements** to guarantee that the solution integrates well with the modeling ecosystem:

1. The persistence layer must be fully compliant with the modeling framework's API. For example, client code should be able to manage models persisted with an alternative persistence manager as if they were persisted using the standard serialization.
2. The underlying persistence backend engine should be easily replaceable to avoid vendor lock-ins.
3. The persistence layer must provide extension points for additional (e.g., domain-specific) caching mechanisms independent from the underlying engine.

Two **performance requirements** represent the improvement we want to achieve over the state of the art:

4. The persistence layer must be memory-friendly, by using on-demand element loading and by removing from memory unused objects.
5. The persistence layer must outperform the execution time of current persistence layers when executing queries on VLMs using the standard API.

In Figure 2, we show the high-level architecture of our proposal particularized for the EMF framework. Our solution consist in a transparent persistence manager behind the model-management interface, so that tools built over the modeling framework would be unaware of it. The persistence manager communicates with a map-database by a driver and supports a pluggable caching strategy. In particular we implement the NeoEMF/Map tool as a persistence manager for EMF on top of MapDB. NeoEMF also supports a graph backend [9].

The architecture answers the interoperability requirements. Requirement 1 is fulfilled by strictly conforming to the base modeling framework. Requirement 2 implies that (i) the APIs must be consistent between the model-management

[2] Indeed, only *DB Store* [1], which uses a proprietary Object/Relational mapper, supports all the CDO features and is released in the *Eclipse Simultaneous Release*.

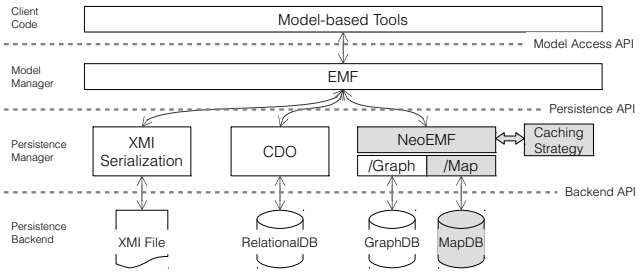


Fig. 2: Overview of the model-persistence framework

framework and the persistence driver (i.e., the module in charge of dealing with the underlying database engine); and (ii) low-level data structures and code accessing the database engine must be completely decoupled from the modeling framework high level code. Maintaining uniform APIs between the different levels allows including additional functionality on top of the persistence driver by using the decorator pattern, such as different cache levels, thus fulfilling Requirement 3.

For fulfilling the performance requirements, we have designed a map-based underlying data model. A map, also called *associative array* or *dictionary*, is an abstract data type composed of a collection of $\langle key, value \rangle$ pairs, such that each possible key appears at most once in the collection. *Hash tables* are typically used to implement maps. A *hash table* is a structure composed by an array of slots in which values are stored. A hash function computes the index of the slot with the correct value. The main advantage of hash tables is that they provide a constant cost on average for searches, insertions and deletions. Maps and hash-maps are one of the most commonly used structures to implement richer data models by their simplicity and performance, specially for data storage. Maps have been used to implement solutions ranging from raw database engines (such as *dbm*, *ndbm*, *MapDB*, etc.) to high level data storage systems (the so called *Key-Value Stores* such as *Oracle NoSQL* or *redis*). The advantage of using a map-based database engine resides in its simplicity, since they are typically provided in the form of software libraries that can be directly embedded in your own solution.

4 NeoEMF/Map

In this section, we describe NeoEMF/Map, our transparent persistence layer for EMF models. The solution is composed of three parts: (i) a memory management strategy, (ii) a map-based data model and (iii) an implementation of atomic model-management operation as low-cost map operations.

4.1 Memory management

Our memory management strategy combines two mechanisms for lightweight on-demand loading and efficient garbage collection. First we decouple dependencies among objects by assigning a **unique identifier** to all model objects. Then:

– To implement **lightweight on-demand loading**, for each live model object, we create a lightweight delegate object that is in charge of on-demand loading the element data and keeping track of the element's state. Delegates load data from the persistence backend by using the object's unique identifier.
– For **efficient garbage collection** in the *Java Runtime Environment*, we avoid to maintain hard Java references among model objects, so that the garbage collector will be allowed to deallocate any model object that is not directly referenced by the application. Thanks to unique element identification we can obtain this decoupling by replacing references among live objects with collections of identifiers corresponding to the referenced objects.

4.2 Map-based data model

We have designed the underlying data model of NeoEMF/Map to reduce the computational cost of each method of the EMF model access API. The design takes advantage of the unique identifier defined in the previous section to flatten the graph structure into a set of key-value mappings.

NeoEMF/Map uses three different maps to store models' information: (i) a *property map*, that keeps all objects' data in a centralized place; (ii) a *type map*, that tracks how objects interact with the meta-level (such as the *instance of* relationships); and (iii) a *containment map*, that defines the models' structure in terms of containment references. Tables 1, 2, and 3 show how the sample model in Fig. 1b is represented using a *key-vale* structure.

As Table 1 shows, keys in the *property map* are a pair, the object *unique identifier*, and the *property* name. The values depend on the *property type* and cardinality (i.e., upper bound). For example, values for single-valued attributes (like the *name* of a Java *Package*) are directly saved as a single literal value as the entry ⟨⟨`'p1'`, `'name'` ⟩, `'package'`⟩ shows; while values for many-valued attributes are saved as an array of single literal values (Fig. 1b does not contain an example of this). Values for single-valued references, such as the *modifier* containment reference from *BodyDeclaration* to *Modifier*, are stored as a single value (corresponding to the id of the referenced object). Examples of this are the entries ⟨⟨`'b1'`, `'modifier'` ⟩, `'m1'`⟩ and ⟨⟨`'b2'`, `'modifier'`⟩, `'m2'`⟩. Finally, multi-valued references are stored as an array containing the literal identifiers of the referenced objects. An example of this is the *bodyDeclarations* containment reference, from *ClassDeclaration* to *BodyDeclaration*, that for the case of the *c1* object is stored as ⟨⟨`'c1'`, `'bodyDeclarations'` ⟩, { `'b1'`, `'b2'` }⟩.

Table 2 shows the structure of the *type map*. The keys are, again, the identifiers of the persisted objects, while the values are named tuples containing the basic information used to identify the corresponding meta-element. For example, the second row of the table specifies the element *p1* is an instance of the *Package* class of the *Java* metamodel (that is identified by the `http://java` *nsUri*).

Structurally, EMF models are trees. That implies that every non-volatile *object* (except the root *object*) must be contained within another *object* (i.e., referenced from another *object* via a containment *reference*). The *containment* map is the data structure in charge of maintaining a record of which is the

Table 1: Property map

Key	Value
⟨'ROOT', 'eContents' ⟩	{ 'p1' }
⟨'p1', 'name' ⟩	'package1'
⟨'p1', 'ownedElement' ⟩	{ 'c1' }
⟨'c1', 'name' ⟩	'class1'
⟨'c1', 'bodyDeclarations' ⟩	{ 'b1', 'b2' }
⟨'b1', 'name' ⟩	'bodyDecl1'
⟨'b1', 'modifier' ⟩	'm1'
⟨'b2', 'name' ⟩	'bodyDecl2'
⟨'b2', 'modifier' ⟩	'm2'
⟨'m1', 'visibility' ⟩	'public'
⟨'m2', 'visibility' ⟩	'public'

Table 2: Type map

Key	Value
'ROOT'	⟨nsUri='http://java', class='RootEObject'⟩
'p1'	⟨nsUri='http://java', class='Package'⟩
'c1'	⟨nsUri='http://java', class='ClassDeclaration'⟩
'b1'	⟨nsUri='http://java', class='BodyDeclaration'⟩
'b2'	⟨nsUri='http://java', class='BodyDeclaration'⟩
'm1'	⟨nsUri='http://java', class='Modifier'⟩
'm2'	⟨nsUri='http://java', class='Modifier'⟩

Table 3: Containment map

Key	Value
'p1'	⟨container='ROOT', featureName='eContents'⟩
'c1'	⟨container='p1', featureName='ownedElements'⟩
'b1'	⟨container='c1', featureName='bodyDeclarations'⟩
'b2'	⟨container='c1', featureName='bodyDeclarations'⟩
'm1'	⟨container='b1', featureName='modifiers'⟩
'm2'	⟨container='b2', featureName='Mmodifiers'⟩

container for every persisted object. Keys in the structure map are the identifier of every persisted object, and the values are named tuples that record both the identifier of the container object and the name of the *property* that relates the container object with the child object (i.e., the object to which the entry corresponds). Table 3 shows in the first row that, for example, the container of the *Package p1* is `ROOT` through the *eContents* property (i.e., it is a root object and is not contained by any other object). In the second row we find the entry that describes that the *Class c1* is contained in the *Package p1* through the *ownedElements property*.

4.3 Model operations as map operations

As mentioned before, EMF generates a natural *Java* implementation of the concepts described in a metamodel. Operations on model elements are executed by calling the generated methods (mainly *getters* and *setters*). It is worth mentioning that multi-valued *properties* are represented using lists (see Listing 1.1, 16–18), thus, operations on such *properties* are executed using list operators.

Table 4 shows a summary of the minimum and maximum number of operations that are performed on the underlying map-based data structures for each

Table 4: Summary of accesses to the underlying map-based storage system

Method	Lookups		Inserts	
	Min.	Max.	Min.	Max.
operations on objects				
getType	1	1	0	0
getContainer	1	1	0	0
getContainingFeature	1	1	0	0
operations on properties				
*get**	1	1	0	0
*set**	0	3	1	3
*isSet**	1	1	0	0
*unset**	1	1	0	1
operations on multi-valued properties				
add	1	3	1	3
remove	1	2	1	2
clear	0	0	1	1
size	1	1	0	0

model operation. It is noteworthy that all the operations (lookups and inserts) in the underlying maps have always a constant cost. For example getting a *property* always implies a single lookup. Setting a *property* may imply from a single insert to 3 lookups and 3 inserts.

5 Experimental evaluation

We evaluate the performance of our proposal by comparing different solutions in the running scenario. Based on our joint experience with industrial partners, we have reverse-engineered three models from open-source Java projects whose sizes resemble those one can find in real world scenarios (see Table 5). Moreover, we have defined a set of queries that are executed on those models. The first of these queries is a well-known scenario in academic literature [14]. The others have been selected to mimic typical model access patterns in reverse engineering.

5.1 Selected backends and execution environment

We have selected NeoEMF/Map, NeoEMF/Graph and CDO for a thorough comparison (see Section 6 for a description of NeoEMF/Graph and other backends). Only the standard EMF interface methods are used in the experi-

Table 5: Summary of the experimental models

#	Model	Size in XMI	Elements
1	org.eclipse.gmt.modisco.java	19.3MB	80 665
2	org.eclipse.jdt.core	420.6MB	1 557 007
3	org.eclipse.jdt.*	984.7MB	3 609 454

ments[3] that are hence agnostic of which backend they are running on. Other backends have been discarded because they do not strictly comply with the standard EMF behavior (e.g. *MongoEMF*), they require manual modifications in the source models or metamodels (e.g. *EMF-fragments*), or because we were only able to run a small subset of the experiments on them (e.g. *Morsa*).

All backends use their respective native EMF generated code for the Java MoDisco metamodel and have been tested in their default configurations with the following exceptions: (i) the timeout values for CDO have been increased since the default ones are too low; (ii) for the sake of a fairer comparison, some extended features have been disabled in CDO (e.g. audit views and branches); and (iii) the Neo4j memory buffers have been tweaked in NeoEMF/Graph to reduce the memory consumption of the embedded Neo4j engine. CDO maintains its caching and prefetching facilities with their default values. In the case of NeoEMF/Map and NeoEMF/Graph no high-level caching is performed.

5.2 Experiments

Experiment I: Import model from XMI — In this experiment (Table 6) we measure the time taken to load a source experimental model—that has been previously derived from the Java code and saved in XMI—and to save it in the selected persistence backend. The saved models are the ones used in next experiments. This experiment measures the time taken to create models that grow monotonically. Only a single configuration (setting the heap size to 8 GB) is used because the XMI model should be completely loaded into memory before saving it in the backend under study.

Experiment II: Simple model traversal — In this experiment we measure the total time taken to load a model, execute a visitor and unload a model. The visitor traverses the full containment tree starting from the root of the model. At each step of the traversal the visitor loads the element content from the backend. We show the results for the three scalable backends plus the standard XMI-based one. Three different maximum heap sizes have been used in this and the following benchmarks to demonstrate how the different backends perform in different configurations: 8GB for demonstrating the performance in an ideal scenario, and 512MB and 256MB to demonstrate how the loading/unloading mechanisms behave in setups with extremely limited memory. Table 7 shows the results of this experimentation over the test models. NeoEMF/Map and NeoEMF/Graph are abbreviated as N/M and N/G respectively.

Experiment III: Query without full traversal — Results of an example query of this type are shown in Table 8. The query returns all the orphan and non-primitive types by navigating and filtering the *orphanTypes* association.

[3] Configuration details: Intel Core i7 3740QM (2.70GHz), 16 GB of DDR3 SDRAM (800MHz), Samsung SM841 SATA3 SSD Hard Disk (6GB/s), Windows 7 Enterprise 64, JRE 1.7.0_40-b43, Eclipse 4.4.0, EMF 2.10.1, NeoEMF/Map uses MapDB 0.9.10, NeoEMF/Graph uses Neo4j 1.9.2, CDO 4.3.1 runs on top of H2 1.3.168

Table 6: Import model from XMI

Model	NeoEMF/Map	NeoEMF/Graph	CDO
1	9 s	41 s	12 s
2	161 s	1 161 s	120 s
3	412 s	3 767 s	301 s

Table 7: Model traversal (includes loading and unloading time)

	-Xmx8g				-Xmx512m				-Xmx256m			
Model	XMI	N/M	N/G	CDO	XMI	N/M	N/G	CDO	XMI	N/M	N/G	CDO
1	4 s	3 s	16 s	14 s	4 s	3 s	15 s	13 s	4 s	3 s	15 s	13 s
2	35 s	25 s	201 s	133 s	Error[a]	42 s	235 s	550 s	Error[b]	121 s	239 s	650 s
3	79 s	62 s	708 s	309 s	Error[b]	366 s	763 s	348 s	Error[b]	443 s	783 s	403 s

[a]java.lang.OutOfMemoryError: Java heap space

[b]java.lang.OutOfMemoryError: GC overhead limit exceeded

Table 8: Model queries that do not traverse the model

	Orphan Non-Primitive Types								
	-Xmx8g			-Xmx512m			-Xmx256m		
Model	N/M	N/G	CDO	N/M	N/G	CDO	N/M	N/G	CDO
1	<1 s[c]	<1 s[c]	<1 s[c]	<1 s[c]	<1 s[c]	<1 s[c]	<1 s[c]	<1 s[c]	<1 s[c]
2	<1 s[c]	2 s	<1 s[c]	<1 s[c]	4 s	<1 s[c]	<1 s[c]	5 s	<1 s[c]
3	<1 s[c]	19 s	2 s	<1 s[c]	19 s	2 s	<1 s[c]	20 s	2 s

[c]Execution time is less than the precission used

Table 9: Model queries that traverse the model

	-Xmx8g			-Xmx512m			-Xmx256m		
Model	N/M	N/G	CDO	N/M	N/G	CDO	N/M	N/G	CDO
	Grabats								
1	1 s	11 s	9 s	1 s	10 s	9 s	1 s	11 s	9 s
2	24 s	188 s	121 s	48 s	217 s	558 s	127 s	228 s	665 s
3	61 s	717 s	299 s	367 s	736 s	370 s	480 s	774 s	479 s
	Unused Methods								
1	2 s	17 s	9 s	1 s	15 s	8 s	1 s	16 s	9 s
2	36 s	359 s	131 s	212 s	427 s	1 235 s	336 s	467 s	1 034 s
3	101 s	1 328 s	294 s	884 s	1 469 s	2 915 s	1290 s	1 818 s	Error[d]
	Thrown Exceptions per Package								
1	1 s	10 s	9 s	1 s	10 s	8 s	1 s	10 s	8 s
2	24 s	184 s	120 s	40 s	214 s	544 s	119 s	224 s	666 s
3	62 s	678 s	296 s	360 s	719 s	353 s	450 s	758 s	427 s
	Invisible Methods								
1	1 s	11 s	9 s	1 s	10 s	9 s	1 s	11 s	9 s
2	26 s	263 s	119 s	55 s	399 s	545 s	158 s	733 s	190 s
3	119 s	3 247 s	320 s	412 s	n/a[e]	404 s	496 s	n/a[e]	1 404 s
	Class Declaration Attributes								
1	1 s	10 s	9 s	1 s	10 s	9 s	1 s	10 s	8 s
2	24 s	183 s	120 s	37 s	216 s	540 s	156 s	226 s	670 s
3	61 s	694 s	294 s	261 s	749 s	348 s	457 s	756 s	460 s

[d]java.lang.OutOfMemoryError: GC overhead limit exceeded

[e]Process killed after 2 hours of computation

Table 10: Model modification and saving

	Orphan Non-Primitive Types								
	-Xmx8g			-Xmx512m			-Xmx256m		
Model	N/M	N/G	CDO	N/M	N/G	CDO	N/M	N/G	CDO
1	1 s	11 s	9 s	1 s	11 s	8 s	1 s	11 s	9 s
2	24 s	191 s	118 s	41 s	213 s	536 s	160 s	224 s	723 s
3	62 s	677 s	296 s	356 s	718 s	334 s	472 s	Error[f]	Error[f]

[f]java.lang.OutOfMemoryError: Java heap space

Experiment IV: Queries with full traversal — These queries start their computation by accessing the list of all the instances of a particular element type, and then apply a filtering to this list to select the starting points for navigating the model. In the experience of our industrial partners, this pattern covers the majority of computational-demanding queries in real world scenarios of the reverse-engineering domain. While the first of these queries is well-known in academic literature, the others have been selected to mimic typical model access patterns: (i) *Grabats* [14] returns the set of classes that hold static method declarations having as return type the holding class (i.e., Singleton); (ii) *Unused Methods* returns the set of method declarations that are private and not internally called; (iii) *Thrown Exceptions per package* collects and returns a map of *Packages* with the *Exceptions* that may be thrown by any of the methods declared by its contained classes; (iv) *Invisible Methods* returns the set of method declarations that are private or protected; and (v) *Class Declaration Attributes* returns a map associating each *Class* declaration to the set of its attribute declarations.

Experiment V: Model modification and saving — The last experiment aims to measure the overhead introduced by the transactional support provided by the different back-ends. Table 10 shows the execution times for renaming all method declarations and saving the modified model.

5.3 Discussion

From the analysis of the results, we can observe that NeoEMF/Map performs, in general, better than any other solution when using the standard API. Only in scenarios with constrained memory the execution times tend equalize due to excessive garbage collection. Nevertheless, other persistence backends tend to be more erratic in those scenarios, running out of memory or presenting big differences in computation times between experiments.

In the XMI import experiment (Table 6) we can observe that NeoEMF/Map presents import times in the the same order of magnitude than CDO, but it is about a 33% slower for the largest model. The simple data model with low-cost operations implemented by NeoEMF/Map contrasts with the more complex data model—and operations—implemented by NeoEMF/Graph which is consistently slower by a factor between 7 and 9. It can be observed that NeoEMF/Map is affected by the overhead produced by modifications on big lists that grow monotonically since it does not implement any caching yet.

Table 7 shows that a traversal of a very large model is much faster (up to 9 times) by using the NeoEMF/Map persistence layer with respect to both a CDO and NeoEMF/Graph. However, in scenarios with very constrained memory, some garbage collection overhead can be noticed. Additionally, if load and unload times are considered (which are negligible for NeoEMF/Map, NeoEMF/Graph and CDO), NeoEMF/Map also outperforms XMI. This is because before executing the traversal, the XMI-based backend needs to parse the whole input model, which is a costly operation. It can also be observed that XMI is unable to load the model into memory for small heap sizes.

Queries that do not require traversing a large part of the model are computed in a negligible time both in NeoEMF/Map and CDO. NeoEMF/Graph shows higher execution times, specially on bigger models (Table 8). In this case, it can be observed that using the rich graph-based data model cannot be exploited when using the standard methods for model traversal.

The fast model-traversal ability of NeoEMF/Map is exploited by the pattern followed by most of the queries in the modernization domain (Table 9). As a result, NeoEMF/Map is consistently faster than other backends for all queries, model sizes and heap sizes. Only in few cases NeoEMF/Map has similar performance to CDO, while in other scenarios NeoEMF/Map is up to 9 times faster. The low memory consumption of NeoEMF/Map also is revealed, since there appear cases in which CDO behaves more erratically, running out of memory or experiencing slowness issues caused by the garbage collector.

Typical queries that traverse the model to apply and persist changes perform, in general, significantly better on NeoEMF/Map (Table 10): 5 times faster on average (on big models) and even up to 9 faster (on small models). In cases with limited memory, however, CDO may present better results than NeoEMF/Map due to garbage collection overhead. Nevertheless, this is not always the case, and CDO also reveals its tendency to run out of memory in such scenarios.

6 Related work

As EMF models are designed following an object-oriented paradigm, our model-persistence backend is inspired by object persistence systems for software tools that have been extensively studied in the last three decades [13, 25]. In recent works, different authors have provided some evidence that the use of schema-free databases may improve performance in persisting VLMs. However, most of them have put focus on graph-oriented [8,9] or document-oriented databases [11,19,20]. Although document-oriented databases can be considered a form of *Key-Value Stores*, NeoEMF/Map is, as far as we know, the only proposal that focus on the optimization of atomic operations by using maps with simple keys and values.

One of the proposals that uses a document-oriented database as its persistence backend is Morsa [19]. It provides on-demand loading capabilities together with incremental updates, and can be used seamlessly to persist models using the standard EMF mechanisms. Performance of the storage backend and their own query language has been reported [19, 20]. Our persistence layer resembles Morsa in several aspects (notably in on-demand loading) but we aim at designing an efficient data representation for models, to optimize runtime performance.

Mongo EMF [11] 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 [17] is another NoSQL-based persistence layer for EMF aimed at achieving fast navigation and fast storage of new data. EMF fragments principles are simpler than in other similar approaches. Those principles are based on the EMF proxy mechanism. In EMF fragments, models are automatically partitioned in several chunks (fragments). Unlike our approach, CDO, and Morsa, all data from a single fragment is loaded at a time. Only links to another fragments are loaded on demand. Another characteristic of this approach 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.

NeoEMF/Graph—previously known as Neo4EMF [9]—is our graph-based proposal for storing VLMs. In NeoEMF/Graph we consider that, since models are a set of highly interconnected elements, graphs are the most natural way to represent them. As we have experienced, however, a significant gain in performance is only obtained when using native queries on the underlying persistence back-end. Although this can be acceptable in some scenarios (as shown in [8,19]), the use of native queries or custom languages implies changes in the client code.

7 Conclusion and Future Work

In this paper we proposed a map-based database persistence layer to handle large models and compared it against previous alternatives based on relational databases and graph databases. Using EMF as the implementation technology, we used queries from some of our industrial partners in the model-driven modernization domain as experiments. The main lesson is that—in terms of performance—typical model-access APIs, with fine-grained methods that only allow for one-step-navigation queries, do not benefit from complex relational or graph-based data structures. Much better results are potentially obtained by optimized low-level data structures, like hash-tables, that guarantee low and constant access times. Additional features that may be of interest in scenarios where performance is not an issue (such as versioning and transactional support provided by CDO) have not been considered.

As further work we want, first, to extend our study on optimized persistence layers for MDE by analyzing caching strategies, especially with reference to element unloading and element prefetching. Caching, and smart prefetching and unloading, aim to (i) alleviate the impact of garbage collection and (ii) reduce the overhead in modifications on lists that grow monotonically. In parallel, we will continue analyzing the benefits of other backends depending on the specific application scenario, such as the graph-based persistence solutions for high-level queries on tools that can drop some of our requirements. In theses cases, for example, bypassing the model access API by translating the queries to high-performance native graph-database queries may provide great benefits.

References

1. CDO DB Store (2014), `http://wiki.eclipse.org/CDO/DB_Store`

2. CDO Hibernate Store (2014), http://wiki.eclipse.org/CDO/Hibernate_Store
3. CDO Model Repository (2014), http://www.eclipse.org/cdo/
4. CDO MongoDB Store (2014), http://wiki.eclipse.org/CDO/MongoDB_Store
5. CDO Objectivity Store (2014), http://wiki.eclipse.org/CDO/Objectivity_Store
6. Eclipse Marketplace - Modeling Tools (2014), http://marketplace.eclipse.org/category/categories/modeling-tools
7. Eclipse Modeling Framework (2014), http://www.eclipse.org/modeling/emf/
8. Barmpis, K., Kolovos, D.S.: Comparative analysis of data persistence technologies for large-scale models. In: Proceedings of the 2012 Extreme Modeling Workshop. pp. 33–38. XM '12, ACM, New York, NY, USA (2012)
9. Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., Launay, D.: Neo4EMF, A Scalable Persistence Layer for EMF Models. In: Modelling Foundations and Applications, Lecture Notes in Computer Science, vol. 8569, pp. 230–241. Springer (2014)
10. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over emf models. In: Model Driven Engineering Languages and Systems, pp. 76–90. Springer (2010)
11. Bryan Hunt: MongoEMF (2014), https://github.com/BryanHunt/mongo-emf/
12. Cook, S., Jones, G., Kent, S., Wills, A.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional, 1st edition edn. (2007)
13. Gallo, F., Minot, R., Thomas, I.: The Object Management System of PCTE as a Software Engineering Database Management System. SIGPLAN Not. 22(1), 12–15 (1987)
14. Jouault, F., et al.: An AmmA/ATL Solution for the GraBaTs 2009 Reverse Engineering Case Study. In: 5th Int. Workshop on Graph-Based Tools (2009)
15. Kelly, S., Lyytinen, K., Rossi, M.: Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In: Advanced Information Systems Engineering, Lecture Notes in Computer Science, vol. 1080, pp. 1–21. Springer (1996)
16. Ledeczi, A., Maroti, M., et al.: The generic modeling environment. In: Workshop on Intelligent Signal Processing. vol. 17 (2001)
17. Markus Scheidgen: EMF fragments (2014), https://github.com/markus1978/emf-fragments/wiki
18. OMG: OMG MOF 2 XMI Mapping Specification version 2.4.1 (August 2011)
19. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: Morsa: A Scalable Approach for Persisting and Accessing Large Models. In: 14th Int. Conf. on Model Driven Engineering Languages and Systems. pp. 77–92. Springer-Verlag (2011)
20. Pagán, J.E., Molina, J.G.: Querying large models efficiently. Information and Software Technology 56(6), 586 – 622 (2014)
21. Pohjonen, R., Tolvanen, J.P.: Automated production of family members: Lessons learned. In: 2nd International Workshop on Product Line Engineering-The Early Steps: Planning, Modeling, and Managing. pp. 49–57 (2002)
22. Ruscio, D.D., Paige, R.F., Pierantonio, A.: Guest editorial to the special issue on success stories in model driven engineering. Sci. Comput. Program. 89, 69–70 (2014)
23. Scheidgen, M., Zubow, A., Fischer, J., Kolbe, T.H.: Automated and Transparent Model Fragmentation for Persisting Large Models. In: 15th Int. Conf. on Model Driven Engineering Languages and Systems, pp. 102–118. Springer-Verlag (2012)
24. Steel, J., Drogemuller, R., Toth, B.: Model interoperability in building information modelling. Software & Systems Modeling 11(1), 99–109 (2012)
25. Wakeman, L., Jowett, J.: PCTE: The Standard for Open Repositories. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1993)