NeoEMF: A Multi-database Model Persistence Framework for Very Large Models

Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gomez, Jordi Cabot

NeoEMF: a Multi-database Model Persistence Framework for Very Large Models

Gwendal Daniel[a], Gerson Sunyé[a], Amine Benelallam[a], Massimo Tisi[a], Yoann Vernageau[a], Abel Gómez[b], Jordi Cabot[c,b]

[a]*AtlanMod Team - Inria, IMT Atlantique, LS2N - Nantes, France*
[b]*Universitat Oberta de Catalunya - Barcelona, Spain*
[c]*ICREA - Barcelona, Spain*

Abstract

The growing role of Model Driven Engineering (MDE) techniques in industry has emphasized scalability of existing model persistence solutions as a major issue. Specifically, there is a need to store, query, and transform very large models in an efficient way. Several persistence solutions based on relational and NoSQL databases have been proposed to achieve scalability. However, they often rely on a single data store, which suits a specific modeling activity, but may not be optimized for other use cases. This paper presents NEOEMF, a tool that tackles this issue by providing a multi-database model persistence framework. Tool website: `http://www.neoemf.com`

Keywords: Model Persistence, Scalability, Large Models

1. Introduction

With the progressive adoption of MDE techniques in industry [5], existing model persistence solutions have to address scalability issues to store, query, and transform large and complex models. Indeed, existing modeling frameworks were first designed to handle simple modeling activities, and often relied on XMI-based serialization to store models. While this format is a good fit for small models, it has shown clear limitations when scaling to large ones [6].

To overcome these limitations, several persistence frameworks based on relational and NoSQL databases have been proposed [2, 6]. They rely on a *lazy-loading* mechanism, which reduces memory consumption by loading only accessed objects. These solutions have proven their efficiency but are generally tailored to a specific data-store implementation, and integrating them into existing applications often implies to update the code base.

Email addresses: `gwendal.daniel@inria.fr` (Gwendal Daniel), `gerson.sunye@inria.fr` (Gerson Sunyé), `amine.benelallam@inria.fr` (Amine Benelallam), `massimo.tisi@inria.fr` (Massimo Tisi), `yoann.vernageau@inria.fr` (Yoann Vernageau), `agomezlla@uoc.edu` (Abel Gómez), `jordi.cabot@icrea.cat` (Jordi Cabot)

In this article we present NeoEMF, a scalable model persistence framework based on a modular architecture enabling model storage into multiple data stores. NeoEMF provides three new model-to-database mappings that complement existing persistence solutions and enable to store models in graph, key-value, and column databases. The framework provides two APIs, one strictly compatible with the Eclipse Modeling Framework (EMF) API —which allows integrating NeoEMF into existing modeling tools with a few changes on the code base— and an *advanced* API —which provides specific features complementing the standard EMF API to further improve scalability of particular modeling scenarios—.

2. Problem and Background

Databases are a well-known solution to store and query large models. They are used in current modeling frameworks, such as CDO [2] or Morsa [6], and have proven their efficiency compared to state-of-the-art XMI serialization. However, existing tools generally rely on a client-server architecture that provides an additional API that has to be integrated in client code to access the model (e.g. to create the server, open a new connection, commit changes, etc).

In these approaches, the choice of the datastore is totally independent of the expected model usage (for example complex querying, interactive editing, or complex model-to-model transformation): the persistence layer offers generic scalability improvements, but it is not optimized for a specific scenario. For example, a graph-based representation of a model can improve scalability by exploiting databases' facilities to handle complex relationships between elements, but will have poor execution time performance in scenarios involving repeated atomic value accesses.

Our previous work on model persistence [3, 4] has shown that providing a well-suited data store for a specific modeling scenario can dramatically improve performance [7]. Based on this observation, we introduce a novel modeling framework based on a multi-database architecture, each one providing optimized performances for specific modeling scenarios.

Currently, NeoEMF provides three implementations—map, graph, and column—respectively optimized for fine-grained access, complex querying, and distributed model transformations. Note that the extensible architecture of NeoEMF allows easily integrating new backends. Furthermore, NeoEMF is, to our knowledge, the only model persistence framework that provides a complete mapping to store models in Neo4j, MapDB, and HBase, complementing other approaches based on relational [2] or document databases [6].

3. Software Framework

This section presents the details of NeoEMF. We first introduce an overview of the framework architecture and its integration in the modeling ecosystem, then we present the main functionalities of the tool and provide some pointers to advanced usages.

3.1. Software Architecture

Figure 1 describes the integration of NeoEMF in the Eclipse-based EMF ecosystem, the most popular modeling framework nowadays. Modelers typically access a

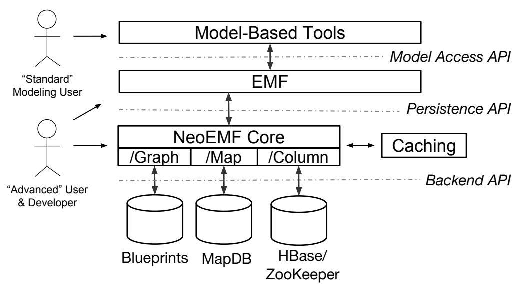

Figure 1: NEOEMF Integration in EMF Ecosystem

model using *Model-based Tools*, which provide high-level modeling features such as a graphical interface, interactive console, or query editor. These features internally rely on EMF's *Model Access API* to navigate models, perform CRUD operations, check constraints, etc. At its core, EMF delegates the operations to a persistence manager using its *Persistence API*, which is in charge of the (de)serialization of the model. The NEOEMF *core* component is defined at this level, and can be registered as a persistence manager for EMF, replacing, for instance, the default XMI persistence manager. This design makes NEOEMF both transparent to the client-application and EMF itself, that simply delegates the calls without taking care of the actual storage.

Once the NEOEMF *core* component has received the request of the modeling operation to perform, it forwards the operation to the appropriate database connector (*Map*, *Graph*, or *Column*), which is in charge of handling the low-level representation of the model. These connectors translate modeling operations into *Backend API* calls, store the results, and reify database records into EMF *EObjects* when needed. NEOEMF also embeds a set of default caching strategies that are used to improve performance of client applications, and can be configured transparently at the EMF API level.

The package diagram shown in Figure 2 details how the NEOEMF core component interacts with the NEOEMF/GRAPH database connector. Note that the same architecture is used for the NEOEMF/MAP and NEOEMF/COLUMN connectors. A *PersistentResource* is the NEOEMF implementation of the EMF *Resource* interface. It contains a set of *PersistentEObject* (the NEOEMF implementation of *EObject*) and references a *PersistentStore* that handles EMF API calls (*add*, *get*, *set* methods) and delegates them to a *PersistenceBackend* which manipulates the underlying database. When a new *PersistentResource* is created, it retrieves from a global *PersistenceBackendFactoryRegistry* the *PersistenceBackendFactory* associated to its *uri*, and uses it to create the *PersistentStore* and *PersistenceBackend* to use to store the model.

The NEOEMF/GRAPH component extends the core architecture at four levels: it defines (i) a *BlueprintsURI* class that is used to create graph-based *PersistentResources*, (ii) a *BlueprintsPersistenceBackend* that extends *PersistenceBackend* by providing methods to manipulate graph databases (*addVertex*, *addEdge*, etc), (iii) a *BlueprintsStore* that maps EMF API operations to the graph primitives provided by the *BlueprintsPersistenceBackend*, and (iv) a dedicated *BlueprintsPersistenceBackendFactory* that creates instances of *BlueprintsStore* and *BlueprintsPersistenceBackend* from *BlueprintsURIs*.

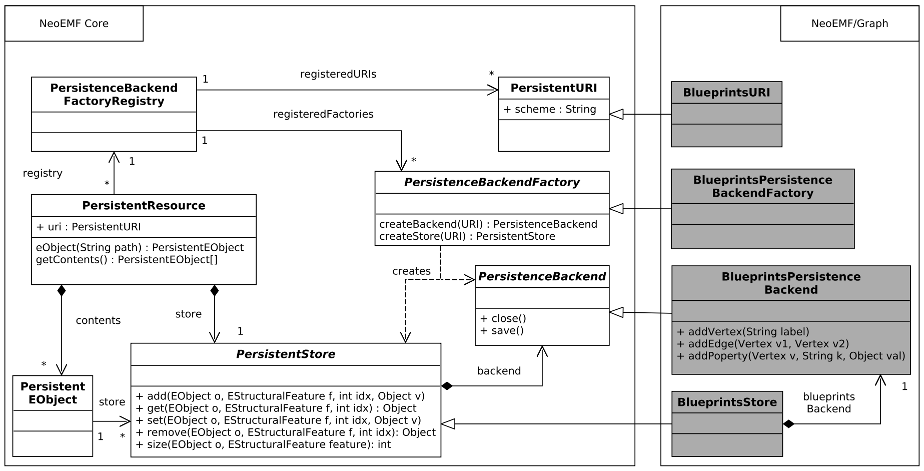
NeoEMF Core
NeoEMF/Graph
PersistenceBackend FactoryRegistry
registeredURIs
registeredFactories
PersistentURI
+ scheme : String
BlueprintsURI
registry
PersistenceBackendFactory
createBackend(URI) : PersistenceBackend
createStore(URI) : PersistentStore
BlueprintsPersistence BackendFactory
PersistentResource
+ uri : PersistentURI
eObject(String path) : PersistentEObject
getContents() : PersistentEObject[]
creates
PersistenceBackend
+ close()
+ save()
BlueprintsPersistence Backend
+ addVertex(String label)
+ addEdge(Vertex v1, Vertex v2)
+ addPoperty(Vertex v, String k, Object val)
contents
store
PersistentStore
+ add(EObject o, EStructuralFeature f, int idx, Object v)
+ get(EObject o, EStructuralFeature f, int idx) : Object
+ set(EObject o, EStructuralFeature f, int idx, Object v)
+ remove(EObject o, EStructuralFeature f, int idx): Object
+ size(EObject o, EStructuralFeature feature): int
backend
Persistent EObject
store
BlueprintsStore
blueprints Backend

Figure 2: NEOEMF Backend Interface

Using this architecture, creating a graph-based *PersistentResource* only requires to associate *BlueprintsURI* and *BlueprintsPersistenceBackendFactory* in the *Persistence-BackendFactoryRegistry*. The created graph-specific store and backend are transparently associated to the *PersistentResource*.

3.2. Software Functionalities

An important characteristic of NEOEMF is its compliance with the EMF API. All classes/interfaces extending existing EMF ones strictly define all their methods, and we put a special attention to ensure that calling a NEOEMF method produces the same behavior (including possible side effects) as standard EMF API calls. As a result, existing applications can easily integrate NEOEMF and benefit immediately from its scalability improvements. Current code manipulating regular EMF *EObjects* does not have to be modified, and will behave as expected.

Specifically, NEOEMF supports all typical EMF features including: (i) a dedicated *code generator* that allows client applications to manipulate models using generated java classes, (ii) support of *Reflective/Dynamic EMF API*, and (iii) a *Resource API* implementation.

As other model solutions, NEOEMF achieves scalability using a *lazy-loading* mechanism, which loads into memory objects only when they are accessed. *Lazy-loading* is defined at the *core* component: NEOEMF implementation of *EObject* consists of a lightweight wrapper delegating all its method calls to an *EStore*, that directly manipulates elements at the database level. Using this technique, NEOEMF benefits from datastore optimizations (such as caches or indices), and only maintains a small amount of elements in memory (the ones that have not been saved), reducing drastically the memory consumption of modeling applications.

In addition to its compliance with the EMF API, NEOEMF provides specific utility features to bypass EMF's limitations, tune internal data stores, and configure caches.

4. Datastores

The previous features are available for a variety of databases supported by NEOEMF. In this section we introduce the different available data stores, describing briefly the model representation in these stores, their differences and the specific modeling scenario they better address. Both standard and advanced features presented in the previous section are implemented in all of them.

NEOEMF/MAP has been designed to provide fast access to atomic operations, such as accessing a single element/attribute and navigating a single reference. This implementation is optimized for EMF API-based accesses, which typically generate this kind of atomic and fragmented calls on the model. NEOEMF/MAP embeds a key-value store, which maintains a set of in-memory/on disk maps to speed up model element accesses. The benchmarks performed in previous work [4] shows that NEOEMF/MAP is the most suitable solution to improve performance and scalability of EMF API-based tools that need to access very large models on a single machine.

NEOEMF/GRAPH persists models in an embedded graph database that represents model elements as *vertices*, attributes as *vertex properties*, and references as *edges*. Metamodel elements are also persisted as *vertices* in the graph, and are linked to their instances through the *instance_of* relationship. Using graphs to store models allows NEOEMF to benefit from the rich traversal features that graph databases usually provide, such as fast shortest-path computation, or efficient processing of complex navigation paths. For instance, these advanced query capabilities have been used to develop the Mogwaï tool [1], that maps OCL expressions to graph navigation traversals.

NEOEMF/COLUMN has been designed to enable the development of distributed MDE-based applications by relying on a distributed column-based datastore. In contrast with Map and Graph implementations, NEOEMF/COLUMN offers concurrent read/write capabilities and guarantees ACID properties at model element level. It exploits the wide availability of distributed clusters in order to distribute intensive read-/write workloads across datanodes.

5. Implementation and Empirical Results

NEOEMF has been implemented as a set of open source Eclipse plugins distributed under the EPL license. The NEOEMF website presents an overview of the key features and current ongoing work. The source code repository and wiki are available on GitHub (`http://www.github.com/atlanmod/NeoEMF`). NEOEMF has been used as the persistence solution of the MONDO European project[8] and is used to store large models automatically extracted from reverse engineering processes. Details on dependencies and library versions are provided in Table 3.

In the following we present a result extracted from the NEOEMF benchmarks available on the project repository (see the wiki for more details and complete results). Note that additional evaluations are also provided in our previous work [4, 3]. We consider four persistence solutions in our evaluation: NEOEMF/GRAPH, NEOEMF/MAP, CDO, and the default XMI serialization mechanism of EMF. The executed query accesses the model using the standard EMF API, making them agnostic of which backend they are running on.

The executed query is extracted from a software modernization use case, and finds in a model representing a Java program all the **unused methods**, that corresponds to private methods that are not internally called. The query is executed over three models of increasing sizes, containing respectively 6756, 80 665, and 1 557 007 elements.

5.1. Results

Table 1 presents the results of executing the presented query over the benchmarked persistence frameworks. Note that execution time is measured in milliseconds, and each table cell contains both the execution time in a large (8 GB) and a small (512 MB) JVM configuration, in order to evaluate how the persistence frameworks handle highly-constrained memory environments.

Model	**XMI**		**CDO**		**NeoEMF/Graph**		**NeoEMF/Map**	
set1	7	7	3212	2924	1942	2346	1425	1437
set2	46	42	12 255	12 169	10 274	11 652	7283	7177
set3	654	*OOM*	171 558	1 160 980	97 782	1 368 399	114 539	118 498

Table 1: UnusedMethods Results in milliseconds (Large VM / Small VM)

5.2. Discussion

The analysis of the results show that both NeoEMF/Graph and NeoEMF/Map are interesting candidates to store and access large models in constrained memory environments. Both NeoEMF implementations perform better than CDO in a large JVM context, and are able to handle *set3* in a constrained memory environment while XMI-based implementation crashes with an *OutOfMemory* error. However when the model to query fits in memory, the XMI serialization outperforms all the existing solutions in terms of execution time. This result is expected because XMI initially loads the full model, allowing to compute the entire query in memory while *lazy-loading* approaches bring into memory elements when they are needed, and have to perform more input/output operations to enable element unloading and improve memory consumption.

In the presented results NeoEMF/Map outperforms other scalable persistence frameworks in terms of exectution time. In addition, the constrained memory environment does not have a significant impact on the connector's performance, enabling very large model querying. This can be explained by the model to data-store mapping used in NeoEMF/Map that is optimized to access a single feature from a modeling element. Technically, the framework does not require any complex in-memory structure to represent the model, and only keeps in memory one key-value pair representing the element currently processed. This architecture allows removing elements from memory as soon as they have been processed, thus reducing the memory consumption.

NeoEMF/Graph also outperforms CDO when a large virtual machine is allocated to the computation, but is less interesting in constrained memory environment. This can be explained by the underlying model to graph mapping, which allows efficient model navigations, while CDO's relational schema requires multiple table join operations to compute a complex navigation. However, the nature of the EMF API that performs low-level and fragmented queries implies a lot of database lookups to find a node corresponding to a given element, which is typically costly in terms of memory in graph databases, limiting NeoEMF/Graph benefits in highly constrained memory environment.

6. Example

NeoEMF wiki provides a set of examples and resources for beginners and advanced users: a tutorial showing how to install and get started with NeoEMF, a ready to use demonstration, code examples, database configuration snippets, and specific backend configurations. An additional demonstration video is available online[1].

As an example, Listing 1 shows how to create and manipulate a NeoEMF/Graph *Resource*. First, we register the *BlueprintsPersistenceBackendFactory* that will be used to create the database connection and persist the model (lines 1 and 2). This initial step is specific to NeoEMF, but it is transparently done when running the application in an Eclipse-based environment, thanks to the *extension points* mechanism. Then, we create and initialize a *ResourceSet* using standard EMF methods (lines 4-6) and associate the *PersistentResourceFactory* to the NeoEMF/Graph protocol. The *ResourceSet* is then used to create a *Resource* using the *BlueprintsURI* helper to create a NeoEMF/Graph compatible *URI* (lines 8-9). NeoEMF provides option builders to ease the definition of backend-specific settings through the standard EMF option *Map*. In our example we use the builder *BlueprintsNeo4jOptionsBuilder* to set the *autocommit* behavior to our *Resource* (lines 10-11). Finally, we *save* the *Resource* to create the underlying database with the provided options, and we manipulate it using standard EMF API calls (lines 12-14).

Listing 1: NeoEMF Resource Creation and Manipulation

```
 1  PersistenceBackendFactoryRegistry.register(BlueprintsURI.SCHEME,
 2    BlueprintsPersistenceBackendFactory.getInstance());
 3
 4  ResourceSet rSet = new ResourceSetImpl();
 5  rSet.getResourceFactoryRegistry().getProtocolToFactoryMap()
 6    .put(BlueprintsURI.SCHEME, PersistentResourceFactory.getInstance());
 7
 8  Resource resource = rSet.createResource(
 9    BlueprintsURI.createFileURI(new File("models/sample.graphdb")))) {
10  Map<String,Object> options = BlueprintsNeo4jOptionsBuilder.newBuilder()
11    .autocommit().asMap();
12  resource.save(options);
13
14  resource.getContents().add(...); // Standard EMF calls
15
16  resource.save(options);
```

7. Conclusion

We have presented NeoEMF, a multi-datastore model persistence framework. It relies on a *lazy-loading* capability that can be configured to load model elements individually or larger collections, allowing very large model navigation in a reduced amount

[1]`http://hdl.handle.net/20.500.12004/1/U/293557`

of memory, by loading elements when they are accessed. NeoEMF provides three implementations that can be plugged transparently to provide an optimized solution to different modeling use cases: atomic accesses through interactive editing, complex query computation, and cloud-based model transformation.

References

[1] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. Mogwaï: a Framework to Handle Complex Queries on Large Models. In *Proc of the 10th RCIS Conference*, pages 225–237. IEEE, 2016.

[2] Eclipse Foundation. The CDO Model Repository (CDO), 2016.

[3] Abel Gómez, Amine Benelallam, and Massimo Tisi. Decentralized Model Persistence for Distributed Computing. In *Proc. of the 3rd BigMDE Workshop*, pages 42–51. CEUR-WS.org, 2015.

[4] Abel Gómez, Gerson Sunyé, Massimo Tisi, and Jordi Cabot. Map-based Transparent Persistence for Very Large Models. In *Proc. of the 18th FASE Conference*, pages 19–34. Springer, 2015.

[5] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *SCP*, 89:144–161, 2014.

[6] Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. A repository for scalable model management. *Software & Systems Modeling*, 14(1):219–239, 2015.

[7] Seyyed M Shah, Ran Wei, Dimitrios S Kolovos, Louis M Rose, Richard F Paige, and Konstantinos Barmpis. A framework to benchmark NoSQL data stores for large-scale model persistence. In *Proc. of the 17th MoDELS Conference*, pages 586–601. Springer, 2014.

[8] The MONDO Project. Scalable Modelling and Model Management on the Cloud. URL: `http://www.mondo-project.org/`, accessed Feb. 2017.

Required Metadata

Current executable software version

Note that NeoEMF has been developped as a set of Eclipse plug-ins and is provided in a packaged update site available online `https://atlanmod.github.io/NeoEMF/releases/1.0.2/plugin/`. A jar version of the tool is also available on Maven central for non-Eclipse platforms `https://mvnrepository.com/search?q=neoemf`.

Nr.	**(executable) Software metadata description**	**Please fill in this column**
S1	Current Software Version	1.0.2
S2	Permanent link to executables of this version	Eclipse Update Site: `https://atlanmod.github.io/NeoEMF/releases/1.0.2/plugin/` Maven Repository `https://mvnrepository.com/search?q=neoemf`
S3	Legal Software License	EPL (NeoEMF) GPL (Neo4j convenience bundle)
S4	Computing Platform / Operating System	Java 8-compatible platform Eclipse users: Eclipse Luna or later
S5	Installation requirements & dependencies	Java 8
S6	If available, link to user manual - if formally published include a reference to the publication in the reference list	Website: `www.neoemf.com` Tutorial: `https://github.com/atlanmod/NeoEMF/wiki/Get-Started`
S7	Support email for questions	`neoemf@googlegroups.com`

Table 2: Software metadata (optional)

Current code version

Nr.	Code metadata description	Please fill in this column
C1	Current Code Version	1.0.2
C2	Permanent link to code / repository used of this code version	`https://github.com/atlanmod/NeoEMF`
C3	Legal Code License	EPL (NeoEMF) GPL (Neo4j convenience bundle)
C4	Code versioning system used	GIT
C5	Software code languages, tools, and services used	Java, Eclipse, Neo4j 1.9.6, MapDB 3.0.2, HBase 1.2.4
C6	Compilation requirements, operating environments	Java, Maven
C7	If available Link to developer documentation / manual	`https://atlanmod.github.io/NeoEMF/releases/1.0.2/doc/`
C8	Support email for questions	`neoemf@googlegroups.com`

Table 3: Software metadata (optional)