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

Gwendal Daniel[1], Gerson Sunyé[1], Amine Benelallam[1], Massimo Tisi[1], Yoann Vernageau[1], Abel Gómez[2,4], and Jordi Cabot[3,4]

[1] AtlanMod Team
Inria, Mines Nantes & Lina
{first_name.last_name}@inria.fr
[2] Departamento de Informtica e Ingeniera de Sistemas
Universidad de Zaragoza
abel.gomez@unizar.es
[3] ICREA
jordi.cabot@icrea.cat
[4] Internet Interdisciplinary Institute
Universitat Oberta de Catalunya

Abstract. The growing use 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, existing solutions often rely on a single data store, which suits a specific modeling activity, but may not be optimized for other use cases. In this article we present NEOEMF, a multi-database model persistence framework able to store very large models in key-value stores, graph databases, and wide column databases. We introduce NEOEMF core features, and present the different data stores and their applications. NEOEMF is open source and available online.

Keywords: Model Persistence, Scalability, Large Models

1 Introduction

With the progressive adoption of MDE techniques in industry [10], existing model persistence solutions have to address scalability issues to store, query, and transform large and complex models [13]. Indeed, existing modeling frameworks were first designed to handle simple modeling activities, and often rely 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 [11].

To overcome these limitations, several persistence frameworks based on relational and NoSQL databases have been proposed [5,7,11]. They rely on a *lazy-loading* mechanism, which reduce memory consumption by loading only accessed objects. These solutions have proven their efficiency compared to state-of-the-art tools, but they are often tailored to a specific data-store implementation.

In these approaches, the choice of the datastore is totally decoupled from the expected model usage (for example complex querying, interactive editing, or complex model-to-model transformation): the persistence layer offers generic scalability improvements, but is not optimized for a specific scenario. For example, a graph-based representation of a model can improve scalability by offering a *lazy-loading* mechanism, but will have poor execution time performance in scenarios involving repeated atomic value accesses.

Our previous work on model persistence have shown that providing a well-suited data store for a specific modeling scenario can dramatically improve performance. Based on this observation, we present in this article NEOEMF, a scalable model persistence framework based on a modular architecture enabling model storage into multiple data stores. Currently, NEOEMF provides three implementations-map, graph, and column–each one optimized for a specific usage scenario. NEOEMF provides two APIs, one strictly compatible to the Eclipse Modeling Framework (EMF) API, easing its integration into existing modeling tools, and an *advanced* API that provides specific features that bypass the standard EMF API to further improve scalability of particular modeling scenarios.

The rest of the paper is organized as follows: Section 2 presents an overview of the NEOEMF architecture, Section 3 and 4 present the core features of the framework and the different datastores. Section 5 provides insights on the framework's implementation, and finally Section 6 summarizes the key points of the paper and presents our future work. Note that examples of NEOEMF usages are provided on NEOEMF's wiki[5] and in a demonstration video available online at `https://youtu.be/_OyWpcOMOfA`. It highlights NEOEMF's core features such as model import, API usage, and the lazy model editor which allows to navigate interactively large models with a low memory footprint. The demonstration also present a concrete use case by showing the different steps needed to integrate NEOEMF into an existing EMF-based application, and use it to store and query models containing several million of elements. Finally, the demonstration presents an overview of two tools developed on top of NEOEMF: the Mogwaï query framework and ATL-MR, a distributed version of the ATL transformation engine.

2 Architecture Overview

Figure 1 describes the integration of NEOEMF in the EMF ecosystem. Modelers typically access a 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. In its core, EMF delegates the operations to a persistence manager using its *Persistence API*, which is in charge of the serialization/deserialization of the model. The NEOEMF *core* component is defined at this level, and can be registered as a persistence manager for EMF, same as, for example, the default XMI persistence manager. This design makes NEOEMF both transparent to the client-application and EMF itself, that simply delegates calls without taking care of the actual storage.

[5] `https://github.com/atlanmod/NeoEMF/wiki`

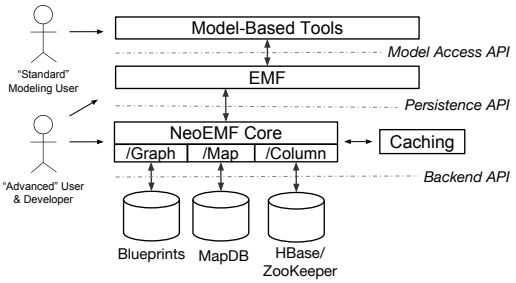

Fig. 1: NEOEMF Integration in EMF Ecosystem

Once the *core* component has received the modeling operation to perform, it forwards the operation to the appropriate database driver (*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.

In addition to this transparent integration into existing EMF applications, NEOEMF provides a specific API, which targets advanced users / high-performance applications. This API provides utility methods which overcome EMF limitations, allow fine-grained tuning of the databases, and access to internal caches. By using this API, NEOEMF can be tuned to improve execution time and/or scalability of a specific modeling scenario.

To provide this smooth integration into the EMF infrastructure, the NEOEMF *core* component redefines the behavior of several EMF classes. For instance, each NEOEMF driver defines a specific implementation of `PersistenceBackendFactory` that is responsible of the concrete data store creation. This factory creates an instance of the data store that corresponds to the *Resource* options. Once the data store has been created, the driver instantiates a specific implementation of the *EStore* interface–also depending on the *Resource* options–that translates the delegated method calls into datastore specific API calls. This architecture allows to change the underlying data store by simply updating the *Resource* options. The *EStore* also returns `PersistentEObject` from the database when needed, using a specific reification mechanism.

3 Software Features

As introduced in the previous Section, NEOEMF provides two API levels: one for a *standard* use of existing EMF applications / APIs, and one *advanced* that allows to bypass EMF's limitations, tune internal data stores, and configure caches. In this Section we present first the *standard* features, available simply by plugging NEOEMF into an existing application, then we introduce its *advanced* features.

3.1 Standard Features

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 integrate NEOEMF with a very small amount of efforts and benefit immediately from NEOEMF scalability improvements. Existing code manipulating regular EMF *EObjects* does not have to be modified, and will behave as expected.

Specifically, NEOEMF supports the following EMF features:

- **Code generation:** NEOEMF provides a dedicated code generator that transparently extends the EMF one, and allows client applications to manipulate models using generated java classes.
- **Reflexive/Dynamic API:** reflexive and dynamic EMF methods (`eSet, eGet, eUnset, eDynamicGet, eDynamicSet ...`) can be used on NEOEMF objects, and behave as their standard implementations.
- **Resource API:** NEOEMF also implements the resource specific API, such as `getContents`, `getAllContents`, `save`, and `load`. In addition, NEOEMF takes advantage of the flexible save and load options to enable backend-specific customizations.

As other model persistence solutions [5, 11], NEOEMF achieves scalability using a **lazy-loading** mechanism, which loads into memory objects only when they are accessed, overcoming XMI's limitations. **Lazy-loading** is defined at the *core* component: NEOEMF implementation of *EObject* consists of a simple 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), 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.

3.2 Advanced Features

In addition to its compliance with the EMF API, NEOEMF provides specific utility features to tackle EMF's limitations, such as the List<EObject> allInstances(EClass eClass) method, which is accessible through the `PersistentResource` interface. This feature tackles the problem of *allInstances* computation in EMF [14] by delegating it to the data store, allowing to retrieve requested element fastly, using data store indexes, or specific data representation.

NEOEMF also includes an *io* module, providing a scalable **Model Importer**, that consists of an event-based XMI parser that bypasses the EMF API to efficiently store the model in a dedicated database with a low memory footprint. The importer is designed to be generic and can be implemented in each backend component. We also plan to add an efficient **Model Exporter** module that would allow to produce optimized model serializations from their database representation.

Finally, NeoEMF contains a set of caching strategies that can be plugged on top of the data store according to specific needs. Note that these caches are available for all connectors, unless otherwise stated.

- **EStructuralFeaturesCaching:** a LRU cache storing loaded objects by their accessed feature.

- **IsSetCaching:** a cache keeping the result of `isSet` calls to avoid multiple accesses to the database.
- **SizeCaching:** a cache keeping the result of `size` calls on multi-valued features to avoid multiple accesses to the database.
- **RecordCaches:** a set of database-specific caches maintaining a list of records to improve execution time.

These caches can be configured using the *save* and *load Resource* methods, which allows to add specific options which are then forwarded to the appropriate `PersistenceBackendFactory`.

4 Datastores

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

4.1 NEOEMF/MAP

NEOEMF/MAP [7] 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 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 [7] show 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/MAP data model is composed of three different maps that store model information: (i) a *property map*, which keeps all objects data in a centralized place; (ii) a *type map*, which tracks how objects relate to the meta-level (such as the *instance of* relationships); and (iii) a *containment map*, which defines the model structure in terms of containment references.

4.2 NEOEMF/GRAPH

NEOEMF/GRAPH [2] 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 complex navigation paths among several vertices/edges. These advanced query capabilities have been used to develop the Mogwaï [4] tool, that maps OCL expressions

to graph navigation traversals. On the other hand, graph databases are not well-suited to compute atomic accesses of single elements or attributes, which are typical queries computed in interactive model edition.

4.3 NEOEMF/COLUMN

NEOEMF/COLUMN [6] has been designed to enable the development of distributed MDE-based applications by relying on a distributed column-based datastore. NEOEMF/COLUMN uses a single table with three column families to store model information: (i) a *property column family* that keeps all objects data stored together; (ii) a *type column family* that tracks how objects relate to the meta-level (such as the *instance of* relationships); and (iii) a *containment column family* that defines the model structure in terms of containment hierarchy.

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. The distributed nature of this persistence solution is used in the ATL-MR [3] tool, a distributed engine for model transformations in the ATL language on top of MapReduce. NEOEMF/COLUMN, enables the cluster's nodes to share read/write rights over the same set of input/output models.

5 Implementation

NEOEMF has been implemented as a set of open source Eclipse plugins distributed under the EPL license. The NEOEMF website[6] presents an overview of the key features and current ongoing work, and the source code repository is fully available on GitHub (`https://github.com/atlanmod/NeoEMF`). NEOEMF has been released as part of the MONDO platform [8].

The NEOEMF/GRAPH implementation relies on Blueprints [12], an interface designed to unify graph databases under a common API. Blueprints has been implemented by a large number of databases, such as Neo4j, OrientDB, and Titan. The use of this abstraction layer on top of graph databases enable client applications to use the graph implementation of their choice, as long as it implements the Blueprints API. For now, NEOEMF/GRAPH embeds Blueprints 2.5.0 and provides a convenience wrapper for Neo4j 1.9.6. An implementation relying on the new Blueprints API (called Tinkerpop3) is under study for now, as well as the creation of additional database wrappers.

NEOEMF/MAP embeds the key-value store MapDB 1.0.9. MapDB provides Maps, Sets, Lists, Queues and other collections backed by off-heap or on-disk storage, and describes itself as a hybrid between *Java Collections* and an embedded database engine [9]. It provides advanced features such as ACID transactions, snapshots, and incremental backups. NEOEMF/MAP relies on the set of `Maps` provided by MapDB and uses them as a key-value store.

NEOEMF/COLUMN is built on top of Apache HBase [1] 0.98.13-hadoop2, a non-relational wide column database providing distributed data storage on top of HDFS. It

[6] `www.neoemf.com`

is able to host very large tables–billions of rows containing millions of columns–atop clusters of commodity hardware. Model distribution is hidden from client applications, which accesses the elements transparently using the standard EMF API.

6 Conclusion

In this article we have presented NeoEMF, a multi-datastore model persistence framework. It relies on a *lazy-loading* capability allowing very large model navigation in a reduced amount 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. Apache. Apache HBase, 2016. URL: `https://hbase.apache.org/`.
2. Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4EMF, a Scalable Persistence Layer for EMF Models. In *Proc. of the 10th ECMFA*, pages 230–241, York, United Kingdom, 2014.
3. Amine Benelallam, Abel Gómez, Massimo Tisi, and Jordi Cabot. Distributed Model-to-Model Transformation with ATL on MapReduce. In *Proc. of the 8th SLE Conference*, pages 37–48. ACM, 2015.
4. Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. Mogwaï: a Framework to Handle Complex Queries on Large Models. In *Proc of the 10th RCIS Conference (to appear)*. IEEE, 2016. Available Online at `http://tinyurl.com/jgopmvk`.
5. Eclipse Foundation. The CDO Model Repository (CDO), 2016. URL: `http://www.eclipse.org/cdo/`.
6. 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.
7. 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.
8. Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan de Lara, István Ráth, Dániel Varró, Gerson Sunyé, and Massimo Tisi. MONDO: Scalable Modelling and Model Management on the Cloud. In *Proc. of the Projects Showcase, (STAF 2015)*, pages 44–53, 2015.
9. MapDB. MapDB, 2016. URL: `www.mapdb.org`.
10. Parastoo Mohagheghi, Miguel A Fernandez, Juan A Martell, Mathias Fritzsche, and Wasif Gilani. MDE Adoption in Industry: Challenges and Success Criteria. In *Models in Software Engineering*, pages 54–59. Springer, 2009.
11. Javier Espinazo Pagán and Jesús García Molina. Querying Large Models Efficiently. *IST*, 2014.
12. Tinkerpop. Blueprints API, 2016. URL: `blueprints.tinkerpop.com`.
13. JB Warmer and AG Kleppe. Building a Flexible Software Factory using Partial Domain Specific Models. In *Proc. of the 6th OOPSLA DSM Workshop*. University of Jyvaskyla, 2006.
14. Ran Wei and Dimitrios S Kolovos. An Efficient Computation Strategy for allInstances(). In *Proc. of the 3rd BigMDE Workshop*, pages 32–42. CEUR-WS.org, 2015.