Efficient Persistence and Query Techniques for Very Large Models

Gwendal Daniel

Efficient Persistence and Query Techniques for Very Large Models

Gwendal Daniel
AtlanMod Team / SOM Research Group
Inria, Mines Nantes, Lina & UOC
gwendal.daniel@inria.fr

Abstract

While Model Driven Engineering is gaining more industrial interest, scalability issues when managing large models have become a major problem in current modeling frameworks. In particular, 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 tackle these issues. However, existing solutions often rely on a single data store, which suits for a specific modeling activity, but may not be optimized for other scenarios. Furthermore, existing solutions often rely on low-level model handling API, limiting NoSQL query performance benefits. In this article, we first introduce NEOEMF, a multi-database model persistence framework able to store very large models in an efficient way according to specific modeling activities. Then, we present the MOGWAÏ query framework, able to compute complex OCL queries over very large models in an efficient way with a small memory footprint. All the presented work is fully open source and available online.

Keywords Model Persistence, Model Query, Scalability, NoSQL, OCL

1. Introduction

The growing use of Model Driven Engineering (MDE) techniques in industry (Hutchinson et al. 2011; Mohagheghi et al. 2009) has emphasized scalability of existing technical solutions to store, query, and transform large models as a major issue (Kolovos et al. 2013; Warmer and Kleppe 2006). Large models containing up to several millions of elements typically appear in various engineering fields, such as civil engineering (Azhar 2011), automotive industry (Bergmann et al. 2010), product lines (Pohjonen and Tolvanen 2002), and can be generated in model-driven reverse engineering processes (Bruneliere et al. 2014), such as software modernization.

Since the publication of the XMI standard (OMG 2016), XML-based serialization has been the preferred format for storing and sharing models and metamodels. The Eclipse Modeling Framework (EMF), the *de-facto* standard for building MDE tools, has even adopted it as their standard serialization mechanism. However, XMI-based serialization has two major drawbacks: (i) XMI files are verbose, favoring human-readability at the expanse of the compactness and (ii) XMI files have to be entirely parsed to obtain a navigable model of their contents. The first one decreases efficiency of I/O accesses, while the second greatly increases the memory needed to load and navigate a model. Moreover, XMI serializations typically lack support for advanced features such as transactions or collaborative edition, and large monolithic model files are challenging to integrate in existing versioning systems (Barmpis and Kolovos 2013).

To overcome these limitations, several research groups have proposed their own solutions (detailed in Section 5) based on relational/NoSQL databases (Eclipse Foundation 2016a; Pagán and Molina 2014; Scheidgen et al. 2012). They often rely on a *lazy-loading* mechanism that reduce memory consumption by bringing objects in memory from the datastore only when they are accessed.

While this evolution of model persistence backends has improved the support for managing large models, they are just a partial solution to the scalability problem in current modeling frameworks: they often provide a single generic way to represent models, regardless the way they will be used. In particular, most of them are focused on saving and loading models in an optimized time and memory consumption, without providing adequate solutions for specific modeling scenarios, such as interactive editing, query computation, or model transformations.

Furthermore, all persistence frameworks are based on the use of low-level model handling APIs (accessing individual model element, attribute, or reference) which are then used by most other MDE tools in the framework ecosystem. This approach is clearly inefficient when used on top of *lazy-loading* persistence frameworks because (i) the API granularity is too fine-grained to benefit from the advanced query capabilities of the backend and (ii) an important time and memory overhead is necessary to construct navigable intermediate objects that can be used to interact with the API.

To overcomes these limitations we introduce NEOEMF, a multi-database persistence framework able to store models in several NoSQL databases, depending on the expected usage of the model. NEOEMF is strictly compatible with the EMF API, and relies on a modular architecture which allows to change underlying backend transparently. We also present MOGWAÏ, a query framework that bypasses modeling API to compute OCL queries over large models in an efficient an scalable way.

The rest of the paper is structured as follows: Section 2 introduces NEOEMF and gives an overview of its feature and supported datastores, Section 3 presents the MOGWAÏ, our solution to compute model queries efficiently. Section 4 provides some insights on the implementation of the presented tools, and Section 5 reviews existing works in the fields of model persistence and model query.

Finally, Section 6 summarizes the key points of the paper, draws conclusions and presents our future work.

2. NeoEMF: a Multi-Datastore Persistence Framework for EMF

Our previous works and experiments on model persistence (Gómez et al. 2015; Benelallam et al. 2014; Gómez et al. 2015) have shown that providing a well-suited data store for a specific modeling scenario can dramatically improve performance of client applications. For example, a graph database can be the optimal solution to compute complex model queries, while it would be quite inefficient for repeated atomic accesses. Based on this observation, we developed NEOEMF (Daniel et al. 2016a), a scalable model persistence framework based on a modular architecture enabling model storage into multiple data stores. It is composed of a transparent persistence layer integrated into EMF, and a set of database connectors which are in charge of the serialization of the model into specific databases. Currently, NeoEMF provides three implementations-map, graph, and column–each one optimized for a specific usage scenario.

In what follows we first introduce the NEOEMF framework and its integration into the EMF ecosystem, then we present the key features of the software, and we briefly introduce the available backends and the typical modeling scenario they address.

2.1 Framework Overview

Figure 1 presents an overview of the NEOEMF framework and its integration within the EMF environment. 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. *Model-based Tools* internally rely on EMF's *Model Access API* to navigate models, create and delete elements, verify 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.

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

In addition to this transparent integration into existing EMF applications, NEOEMF provides its own 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.

2.2 Software 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 ensure that a call to a NEOEMF method produces the same behavior (including possible side effects) as standard EMF API calls. As a result, existing applications can move from EMF to NEOEMF with a very small amount of efforts and benefit immediately from its scalability improvements.

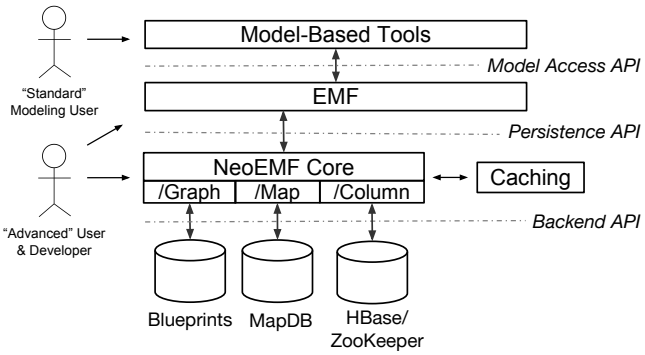


Figure 1. NeoEMF Integration in EMF Ecosystem

In particular, NEOEMF supports the following EMF features:

- **Code generation:** NEOEMF embeds a dedicated code generator that transparently extends the EMF one, and allows client applications to manipulate models using generated java classes.
- **Reflexive/Dynamic API:** in addition to generated code, reflexive and dynamic EMF methods 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`.

As other model persistence solutions (Eclipse Foundation 2016a; Pagán and Molina 2014), 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 the corresponding database driver. Using this technique, NEOEMF benefits from data store 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.

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

- **EStructuralFeaturesCaching:** a 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 storing the size of 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.

Finally, in our last work (Daniel et al. 2016c) we have extended the cache support in NEOEMF with an integrated prefetching/-caching framework that allows to customize data access in order to speed-up query computation. The PrefetchML framework is composed of a DSL that allows designers to specify prefetching and caching rules with a high-level of abstraction, and an execution engine that is in charge of triggering the rules and fetching the elements from the database.

2.3 Data Stores

For now, NEOEMF provide three connectors that are able to represent model into specific data stores. In this section we present these connectors and the modeling scenario they are optimized for.

2.3.1 NEOEMF/MAP

NEOEMF/MAP (Gómez et al. 2015) 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 (Gómez et al. 2015) 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.

2.3.2 NEOEMF/GRAPH

NEOEMF/GRAPH (Benelallam et al. 2014) relies on the rich traversal features that graph databases usually provide to compute efficiently complex queries over models. This specific modeling scenario is further explained in the next Section, where we present a framework able to compute OCL queries efficiently by translating them into graph traversals. NEOEMF/GRAPH maps models to property graphs, where model elements are translated into *vertices*, attributes into *vertex properties*, and references as *edges*. Note that to enable complex query computation, metamodel elements are also persisted as *vertices*, and are linked to their instances through a dedicated `INSTANCE_OF` relationship.

2.3.3 NEOEMF/COLUMN

NEOEMF/COLUMN (Gómez et al. 2015) relies on a distributed column-based data store to enable the development of distributed MDE-based applications. 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 (Benelallam et al. 2015) tool, a distributed engine for model transformations in the ATL language on top of MapReduce.

3. Mogwaï: a Framework to Perform OCL Queries on Large Models

In the previous Section we introduced the NEOEMF framework, that provides a transparent way to store models into NoSQL databases. While this architecture allows to store very large models in a scalable way, the presented solution is tailored to the low-level modeling API, which generates fragmented queries on the data store, reducing the benefits of advanced database query capabilities. Furthermore, the EMF API imposes to reify each traversed element into a navigable EMF object, even if it is not part of the final result of the query, increasing the memory needed to compute a query.

To address these issues we propose the MOGWAÏ (Daniel et al. 2016b) query framework that is able to handle complex queries on large models. The MOGWAÏ framework takes benefits of the advanced query language available on NEOEMF/GRAPH's internal data store. The MOGWAÏ framework translates queries expressed in OCL (Object Constraint Language) into Gremlin (Tinkerpop 2016), a graph traversal query language. Generated queries are then sent to the database that is in charge of their computation, bypassing EMF API limitations.

In this Section we first show an overview of the Gremlin language, then we present our transformation approach, and we introduce some experimental results.

3.1 The Gremlin Language

Gremlin is a Groovy based query language which is part of the Tinkerpop initiative, a set of tools that aims to uniform graph database under a common API.It is built on top of *Pipes*, a data-flow framework based on process graphs. A process graph is composed of vertices representing computational units and communication edges which can be combined to create a complex processing. In the Gremlin terminology, these complex processing are called *traversals*, and are composed of a chain of simple computational units named *steps*.

Existing work have shown that Gremlin is an interesting alternative to Cypher, the pattern matching language used to query Neo4j graph database (Holzschuher and Peinl 2013) that can even outperform the native query language for specific query scenarios. Gremlin defines four types of steps:

- **Transform steps:** functions mapping inputs of a given type to outputs of another type. They constitute the core of Gremlin: they provide access to adjacent vertices, incoming and outgoing edges, and properties. In addition to built-in navigation steps, Gremlin defines a generic *transformation* step that applies a function to its input and returns the computed results.
- **Filter steps:** functions to select or reject input elements w.r.t. a given condition. They are used to check property existence, compare values, remove duplicated results, or retain particular objects in a traversal.
- **Branch steps:** functions to split the computation into several parallelized sub-traversals and merge their results.
- **Side-effect steps:** functions returning their input values and applying side-effect operations (edge or vertex creation, property update, variable definition or assignation).

In addition, the *step* interface provides a set of built-in methods to access meta information: number of objects in a step, output existence, or first element in a step. These methods can be called inside a traversal to control its execution or check conditions on particular elements in a step.

We chose Gremlin as our target language because its expressivity allows to map the entire OCL, and because it is to our knowledge the only one that is supported by several NoSQL databases.

3.2 Framework Overview

The MOGWAÏ framework is composed of two components: (i) the `OCL2Gremlin` model-to-model transformation, which maps OCL expressions on to Gremlin traversals, and (ii) the `NeoEMF/Mogwaï` persistence layer, an extension of NEOEMF/GRAPH that provides an advanced query API for graph databases. We choose OCL as our input language because it is a well-known OMG standard used to complement graphical (meta) modeling languages with textual descriptions of invariants, operation contracts, derivation rules, and query expressions. Gremlin is a NoSQL query language designed to query databases implementing the Blueprints API, an abstraction layer on top of graph stores which has been implemented by several databases. Therefore, we choose Gremlin as our target language, because it is the most mature and generic solution to query a wider variety of NoSQL databases.

Figure 2 shows the overall query process of (a) the MOGWAÏ query framework and compares it with (b) standard EMF API based approaches. An initial textual OCL expression is parsed and transformed into an OCL query model. This model constitutes the input of the `OCL2Gremlin` MOGWAÏ component, which consists

of a model-to-model transformation generating the corresponding Gremlin traversal model.

This transformation is composed of a mapping from OCL on to Gremlin and a translation algorithm that implements this mapping and merge the created *steps* into a single query. The Gremlin model is then converted to a textual expression and sent to the `NeoEMF/Mogwaï` component, that computes it on the database side. Query results are then reified as standard EMF objects by `NeoEMF/Mogwaï`, making them usable in any EMF-based scenario.

Compared to existing query frameworks, MOGWAÏ does not rely on the EMF API to perform a query. In general, API based query frameworks translate OCL queries into a sequence of low-level API calls, which are then performed one after another on the persistence layer (in this example `NeoEMF/Graph`). While this approach has the benefit to be compatible with every EMF-based application, it does not take full advantage of the database structure and query optimizations. Furthermore, each object fetched from the database has to be reified to be navigable, even if it is not going to be part of the end result. Therefore, execution time of the EMF-based solutions strongly depends on the number of intermediate objects fetched from the database while for the MOGWAÏ framework, execution time does not depend on the number of intermediate objects, making it more scalable over large models.

3.3 Experimental Results

Experimental results presented in (Daniel et al. 2016b) show that using the MOGWAÏ framework to perform complex queries over large models can dramatically improve performances both in terms of memory consumption and execution time. In particular, `allInstances` based queries computed with the MOGWAÏ are up to 20 times faster and up to 75 times better in terms of memory consumption than the Eclipse OCL interpreter and the EMF-Query framework, two state of the art tools in EMF-based model queries.

Instead, if the query traverses a small part of the model, or if an important part of the intermediate results are needed anyway the benefits of using the MOGWAÏ framework are reduced. In particular, the overhead implied by the transformation engine may not be worthwhile when dealing with relatively small models or simple queries.

The main disadvantage of the MOGWAÏ framework concerns its integration to an EMF environment. To benefit from the MOGWAÏ, other Eclipse plug-ins need to be explicitly instructed to use it. Integration with the MOGWAÏ framework is straighforward but must be explicitly done. Instead, other solutions based on the standard EMF API provide benefits in a transparent manner to all tools using that API.

4. Tool Support

NEOEMF is composed of a set of open source Eclipse plugins distributed under the EPL license. Available components are actively developed and maintained, and the source code repository is fully available on GitHub (`https://github.com/atlanmod/NeoEMF`). The NEOEMF website[1] presents an overview of the supported datastores, the key features, and current ongoing work. NEOEMF has been released as part of the MONDO platform (Kolovos et al. 2015).

- NEOEMF/GRAPH relies on Blueprints, a high-level interface designed to unify graph databases under a common API. Blueprints has been implemented by several datastores such as Neo4j, OrientDB, and Titan. Using this abstraction layer, client applications can choose the graph store of their choice to persist models through NEOEMF/GRAPH. For now, NEOEMF embeds Blueprints 2.5.0 and provides a convenience wrapper for Neo4j 1.9.6.
- NEOEMF/MAP is built on top of MapDB 1.0.9, a key-value store providing Maps, Sets, Lists, and other collections backed by off-heap or on-disk storage. MapDB provides advanced features such as database snapshots, ACID transactions support, and incremental backups.
- NEOEMF/COLUMN persists models in Apache HBase 0.98.12-hadoop2, a wide column database providing distributed data storage on top of HDFS. HBase is designed to handle very large tables atop clusters of commodity hardware. The distribution of the model on the cluster is hidden from client applications, which accesses them transparently through the EMF API.

[1] `www.neoemf.com`

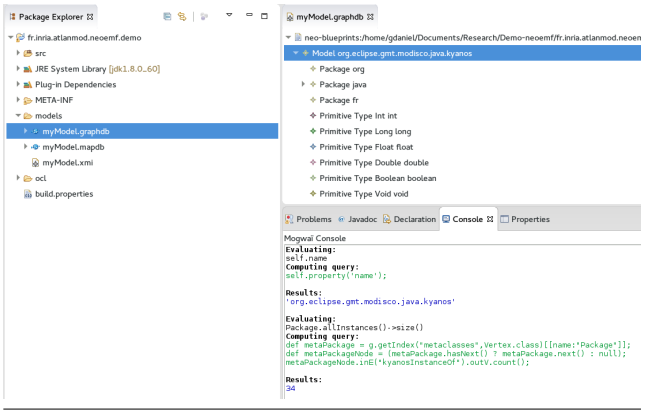


Figure 3. Mogwaï Interactive Console

A prototype of the MOGWAÏ framework has been developed as part of NEOEMF (`https://github.com/atlanmod/Mogwai`). It extends the standard EMF API provided by NEOEMF by defining additional `query` methods at the `Resource` level. The query API accepts a textual OCL expression or an URI to an OCL file containing the expressions to compute. In addition, it is possible to provide input values that represents *self* and parameter variables. The framework also provides an OCL console (see Figure 3) integrated into Eclipse that allows to query NEOEMF/GRAPH models interactively.

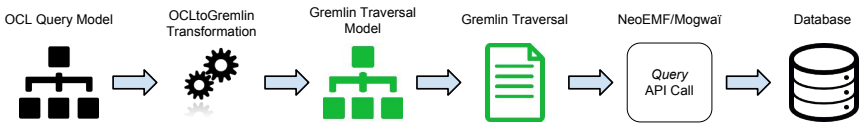
OCL queries are parsed using Eclipse MDT OCL, and the core transformation creating the Gremlin model from the OCL one is composed of a set of 70 ATL (Jouault et al. 2008) rules and helpers. The created Gremlin model is then expressed using its textual syntax and sent to an embedded Gremlin engin, which executes the query and returns the results. The reification of these results into model elements is delegated to NEOEMF/GRAPH, that is in charge of the mapping between graph and model elements.

5. Related Work

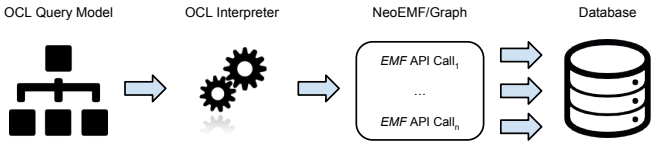
In this Section we present existing solutions that aims to tackle scalability issues to store and query large models and we compare them with NEOEMF on the persistence side, and the MOGWAÏ framework on the query one.

5.1 Scalable Model Persistence

The CDO model repository (Eclipse Foundation 2016a) is a scalable model persistence framework based on a client-server architecture to handle large model in a collaborative environment. It provides some advanced features such as transaction support or basic prefetching, and provides a lazy-loading mechanism to reduce memory consumption. CDO can be plugged with several database



(a) The Mogwaï Query Framework



(b) EMF-based Query Frameworks

Figure 2. Comparison of OCL execution

connectors to store a model, but in practice only relational ones are used. In addition, different experiences have shown that CDO faces scalability issues when dealing with very large models (Pagán and Molina 2014; Scheidgen et al. 2012).

Morsa (Pagán et al. 2011; Pagán and Molina 2014) is one of the first approaches that use NoSQL databases to handle very large EMF models. It relies on a client-server architecture based on MongoDB[2] and aims to manage scalability issues using document-oriented database facilities and a *lazy-loading* mechanism. Morsa model persistence is available through the standard EMF mechanisms, making its integration transparent in existing EMF based applications. NeoEMF is similar to Morsa in several aspects, but aims to provide multiple backends that can be chosen according to a specific modeling scenario.

EMF fragments (Scheidgen et al. 2012) is another NoSQL-based persistence layer for EMF aimed at achieving fast storage of new data and fast navigation of persisted models. Supported backends are MongoDB, Apache Hbase and regular files on the file system. EMF fragments is based on the proxy mechanism used by EMF for inter-document relationships: models are automatically partitioned in several chunks (fragments) using metamodel annotations, and linked together using the standard EMF proxy mechanism. 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 metamodels have to be modified to indicate where the partitions should be made to get the partitioning capabilities, whereas NeoEMF can be plugged directly into existing EMF-based applications.

5.2 Model Query

There are several frameworks to query models, specially targeting the EMF framework (including one or more of the EMF backends mentioned before). The main ones are Eclipse MDT OCL (Eclipse Foundation 2016b), EMF-Query (The Eclipse Foundation 2016) and IncQuery (Bergmann et al. 2009).

Eclipse MDT OCL provides an execution environment to evaluate OCL invariants and queries over models. It relies on the EMF API to navigate the model, and stores `allInstances` results in a cache to speed up their computation.

EMF-Query is a framework that provides an abstraction layer on top of the EMF API to query a model. It includes a set of tools to ease the definition of queries and manipulate results. Compared to the Mogwaï framework, these two solutions are strongly dependent on the EMF API, providing on the one hand an easy integration in existing EMF applications, but on the other hand they are unable to benefit from all performance advantages of NoSQL databases due to this API dependency.

EMF-IncQuery (Bergmann et al. 2009) is an incremental pattern matcher framework to query EMF models. It bypasses API limitations using a persistence-independent index mechanism to improve model access performance. It is based on an adaptation of a RETE algorithm, and query results are cached and incrementally updated using the EMF notification mechanism to improve performance. While EMF-IncQuery shows great execution time performances (Bergmann et al. 2011) when repeating a query multiple times on a model, the results presented in this article show mitigated performances for single evaluation of queries. This is not the case for our framework. Caches and indexes must be built for each query, implying a non-negligible memory overhead compared to the Mogwaï framework. In addition, the initialization of the index needs a complete resource traversal, based on EMF API, which can be costly for *lazy-loading* persistence frameworks.

6. Conclusion

In this article we have presented NeoEMF, our solution to store and access very large models using a multi-datastore model persistence framework. NeoEMF relies on a *lazy-loading* capability allowing very large model navigation in a reduced amount of memory, by loading elements from the datastore only when they are accessed. NeoEMF has been designed to be fully compatible with existing EMF-based applications by providing a complete implementation of the EMF API. Datastores' behavior and internal caches can be tuned by providing options to the standard `save` and `load` EMF methods. Currently, NeoEMF provides three implementations (graph, map, and column) that can be plugged transparently to provide an optimized solution to different modeling use cases: frequent and repeated atomic accesses, complex query computation, and cloud-based model transformation.

In addition to the persistence layer itself, we have introduce the Mogwaï framework that generates Gremlin traversals from OCL queries in order to maximize the benefits of using a graph backend to store large models. Mogwaï is integrated in the NeoEMF infrastructure, extending NeoEMF/Graph with custom query capabilities. OCL queries are translated using model-to-model transformation into Gremlin traversals that are then computed on the database side, reducing the overhead implied by modeling API and the reification of intermediate. Experiments detailed in previous work (Daniel et al. 2016b) have shown that using this approach brings a significant improvement both in terms of execution time and memory consumption. NeoEMF and Mogwaï are developed as open-source Eclipse plugins and available online.

[2] `http://www.mongodb.org`

Model transformations intensively use model queries to navigate the model to transform, match source elements, or set target values. Integrating the MOGWAÏ framework in model transformation engines (such as ATL (Jouault et al. 2008)) to compute these queries could drastically reduce the execution time and memory consumption implied by the transformation of large models. Another possible approach would be to extend the MOGWAÏ to translate the transformation itself into database queries and compute it entirely on the database side.

As future work we plan to study the interest of other datastores that could be beneficial for specific use cases. For example, we want to study if a document-based representation could provide some performance gains. We also want to study how datastores can be combined to optimize a set of modeling activities (for example a map/graph backend that would speed-up both query computation and atomic accesses). Finally, we plan to integrate into NEOEMF advanced features which are typically needed by modeling processes, such as model versioning, or collaborative edition. The later could for example benefit of the distributed architecture provided by NEOEMF/COLUMN.

Another ongoing work is to study the integration of the MOGWAÏ framework into model persistence solutions that do not rely on a Gremlin compatible database. For example, we plan to adapt existing work on EOL to SQL translation (Carlos et al. 2014) to test our model-to-model transformation-based approach over SQL databases. Generating SQL queries would also enable to use the Spark-SQL connector for HBase in order to improve query execution time and memory consumption over NEOEMF/COLUMN.

References

S. Azhar. Building information modeling (BIM): Trends, benefits, risks, and challenges for the AEC industry. *Leadership and Management in Engineering*, pages 241–252, 2011.

K. Barmpis and D. Kolovos. Hawk: Towards a scalable model indexing architecture. In *Proc. of BigMDE'13*, pages 6–9. ACM, 2013.

A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay. Neo4EMF, a Scalable Persistence Layer for EMF Models. In *Proc. of the 10th ECMFA*, pages 230–241. Springer, 2014.

A. Benelallam, A. Gómez, M. Tisi, and J. Cabot. Distributed Model-to-Model Transformation with ATL on MapReduce. In *Proc. of the 8th SLE Conference*, pages 37–48. ACM, 2015.

G. Bergmann, Á. Horváth, I. Ráth, and D. Varró. Efficient model transformations by combining pattern matching strategies. In *Proc. of the 2nd ICMT*, pages 20–34, Zurich, Switzerland, 2009. URL `http://dx.doi.org/10.1007/978-3-642-02408-5_3`.

G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös. Incremental evaluation of model queries over EMF models. In *Proc. of the 13th MoDELS Conference*, pages 76–90. Springer, 2010.

G. Bergmann, A. Horváth, I. Ráth, and D. Varró. Incremental evaluation of model queries over EMF models: A tutorial on EMF-IncQuery. In *Proc. of the 7th ECMFA*, pages 389–390, Berlin, Heidelberg, 2011. ISBN 978-3-642-21469-1. URL `http://dl.acm.org/citation.cfm?id=2023522.2023565`.

H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot. MoDisco: A model driven reverse engineering framework. *IST*, pages 1012 – 1032, 2014.

X. D. Carlos, G. Sagardui, and S. Trujillo. Mqt, an approach for run-time query translation: From EOL to SQL. In *Proc. of OCL 2014 co-located with MoDELS 2014*, pages 13–22, Valencia, Spain, 2014.

G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, and J. Cabot. Neoemf: a multi-database model persistence framework for very large models. In *Proc. of the MoDELS 2016 Tool Demonstration Session [To appear]*. CEUR-WS, 2016a. Available Online at `http://tinyurl.com/jhkqoyx`.

G. Daniel, G. Sunyé, and J. Cabot. Mogwaï: a framework to handle complex queries on large models. In *Proc. of the 10th RCIS Conference [To appear]*. IEEE, 2016b. Available Online at `http://tinyurl.com/zx6cfam`.

G. Daniel, G. Sunyé, and J. Cabot. Prefetchml: a framework for prefetching and caching models. In *Proc. of the 19th MoDELS Conference [To appear]*. ACM/IEEE, 2016c. Available Online at `http://tinyurl.com/huc55hl`.

Eclipse Foundation. The CDO Model Repository (CDO), 2016a. URL `http://www.eclipse.org/cdo/`. URL: `http://www.eclipse.org/cdo/`.

Eclipse Foundation. MDT OCL, 2016b. URL `www.eclipse.org/modeling/mdt/?project=ocl`. URL: `www.eclipse.org/modeling/mdt/?project=ocl`.

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

A. Gómez, G. Sunyé, M. Tisi, and J. Cabot. Map-based transparent persistence for very large models. In *Proc. of the 18th FASE Conference*. Springer, 2015.

F. Holzschuher and R. Peinl. Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j. In *Proc. of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204, New York, NY, USA, 2013. ISBN 978-1-4503-1599-9. doi: 10.1145/2457317.2457351. URL `http://doi.acm.org/10.1145/2457317.2457351`.

J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 633–642. IEEE, 2011.

F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *SCP*, pages 31 – 39, 2008.

D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, et al. A research roadmap towards achieving scalability in model driven engineering. In *Proc. of BigMDE'13*, pages 1–10. ACM, 2013.

D. S. Kolovos, L. M. Rose, R. F. Paige, E. Guerra, J. S. Cuadrado, J. de Lara, I. Ráth, D. Varró, G. Sunyé, and M. Tisi. MONDO: Scalable Modelling and Model Management on the Cloud. In *Proc. of the Projects Showcase, (STAF 2015)*, pages 44–53, 2015.

P. Mohagheghi, M. A. Fernandez, J. A. Martell, M. Fritzsche, and W. Gilani. MDE adoption in industry: challenges and success criteria. In *Proc. of Workshops at MoDELS 2008*, pages 54–59. Springer, 2009.

OMG. OMG MOF 2 XMI Mapping Specification version 2.5.1, 2016. URL `http://www.omg.org/spec/XMI/2.5.1/`.

J. E. Pagán and J. G. Molina. Querying large models efficiently. *IST*, 2014. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2014.01.005. URL `http://dx.doi.org/10.1016/j.infsof.2014.01.005`.

J. E. Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: A scalable approach for persisting and accessing large models. In *Proc. of the 14th MoDELS Conference*, pages 77–92. Springer, 2011.

R. Pohjonen and J.-P. Tolvanen. Automated production of family members: Lessons learned. In *Proc. of PLEES'02*, pages 49–57. IESE, 2002.

M. Scheidgen, A. Zubow, J. Fischer, and T. Kolbe. Automated and Transparent Model Fragmentation for Persisting Large Models. In *Proc. of the 15th MoDELS Conference*, pages 102–118. Springer, 2012.

The Eclipse Foundation. EMF Query, 2016. URL `https://projects.eclipse.org/projects/modeling.emf.query`.

Tinkerpop. The Gremlin Language, 2016. URL `www.gremlin.tinkerpop.com`. URL: `gremlin.tinkerpop.com`.

J. Warmer and A. Kleppe. Building a flexible software factory using partial domain specific models. In *Proc. of the 6th DSM Workshop*, pages 15–22. University of Jyvaskyla, 2006.