Morsa: A Scalable Approach for Persisting and Accessing Large Models*

Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina

University of Murcia, Spain
{jespinazo,jesusc,jmolina}@um.es

Abstract. Applying Model-Driven Engineering (MDE) in industrial-scale systems requires managing complex models which may be very large. These models must be persisted in a scalable way that allows their manipulation by client applications without fully loading them.

In this paper we propose Morsa, an approach that provides scalable access to large models through load on demand; model persistence is supported by a NoSQL database. We discuss some load on demand algorithms and database design alternatives. A prototype that integrates transparently with EMF is presented and its evaluation demonstrates that it is capable of fully loading large models with a limited amount of memory. Moreover, a benchmark has been executed, exhibiting better performance than the EMF XMI file-based persistence and the most representative model repository, CDO.

Keywords: model persistence, scalability, large models.

1 Introduction

During the last decade, the growing maturity of Model-Driven Engineering (MDE) technologies is promoting their adoption by large companies [1][2], taking advantage of their benefits in terms of productivity, quality and reuse. However, applying MDE in this context requires industry-scale tools that operate with very large and complex models. One such relevant operation is model persistence and the corresponding access, which is typically supported by modeling frameworks. A well-known example of a modeling framework is EMF[3].

One critical concern for the industrial adoption of MDE is the *scalability* of tools when accessing large models. As noted by [4], *"scalability is what is holding back a number of potential adopters"*. Scalability may be tackled in different ways. One approach is the *modularization* of modeling languages [4] to keep models at a reasonable size. However, the complexity of large models makes it difficult to automatically divide them into parts that are easily accessible [5]. For example, code models extracted from a legacy system being modernized may not be properly modularizable because of the complexity

* This work is funded by the Spanish Ministry of Science (project TIN2009-11555) and Fundación Séneca (grant 14954/BPS/10).

J. Whittle, T. Clark, and T. Kühne (Eds.): MODELS 2011, LNCS 6981, pp. 77–92, 2011.

of their interconnections, hence having a scalable model persistence solution would be mandatory [6]. In EMF models are usually stored in XMI files, which have to be parsed in order to build models in memory. The usual EMF approach consists of a SAX parser that fully reads an XMI file and builds the entire model in memory at once. This solution does not scale since large models may not be fully kept in memory, causing the parser to overflow the client. Therefore handling large models requires some mechanism that allows the client to load only the objects that it will use [5]. Model repositories are emerging as persistence solutions for large models, providing remote model access with advanced features such as concurrent access, transaction support and versioning; model repositories are discussed in Section 7. Currently, CDO is the most mature repository for EMF; however, it does not scale properly as shown in Section 8.

Another concern that arises when client applications access persisted models is *tool integration*. The integration between a persistence solution and any client must be transparent, that is, it must conform to the standard model access interface defined by the modeling framework (e.g. the Resource interface of EMF). Moreover, it would be convenient for a persistence solution not to require any preprocessing on the (meta)models in order to load or store them, e.g. requiring source code generation for the persisted (meta)models [8][9] .

In this paper we present Morsa, a model persistence solution aimed at achieving scalability in large model access. While other approaches use object-relational mappings [8], Morsa relies on a document-based NoSQL database to achieve server scalability; moreover, document-based NoSQL provides a more natural model persistence backend than object-relational mappings since, for example, many-to-many relationships are represented just as any other kind of feature, while object-relational-mappings require intermediate tables. Morsa handles client scalability using a load on demand mechanism supported by an object cache which is configurable with different policies. We discuss how these policies fit for common model traversals such as depth-first order and breadth-first order. We contribute a prototype implementation for EMF [10] that integrates transparently with client tools such as model transformation languages. Its evaluation demonstrates that it is capable of fully loading large models with a limited amount of memory. Moreover, a benchmark has been executed, exhibiting better performance than the EMF XMI file-based persistence and CDO. In this paper we focus only on accessing models. Our implementation supports storing models into the repository, but the details are out of the scope of this paper.

The rest of the paper is structured as follows: Section 2 introduces the NoSQL movement and some terminology about models; Section 3 gives an overview of our approach; Sections 4, 5 and 6 discuss the database and loading algorithm design, and the integration and implementation of our approach, respectively; Sections 7 and 8 comment the related work and the evaluation of Morsa and finally Section 9 shows our conclusions and further work.

2 Background

As introduced in the previous section, this paper deals with the problem of persisting and accessing large models. In this section, the basic concepts regarding models and model persistence that will be used in the rest of the paper are explained. Moreover, the NoSQL paradigm is introduced as an alternative to relational databases and object-relational mappings for model persistence.

2.1 Metamodeling

A model is an instance of a metamodel which defines the metaclasses and relationships that the model elements conform to. It can also be seen as a directed labeled graph, where each node represents an object (i.e., a model element) and each edge represents a relationship between objects, which may be containment or non-containment relationships. A containment relationship specifies a hierarchical transitive link between a parent object (source) and a child object (target), defining tree-like structures. Given this graph nature, the concepts of ascendant, descendant, sibling, breadth, depth, etc. common to this mathematical structure can also be used for models. An object that has no ascendants is called a *root object*.

Non-containment relationships define graph-like structures where objects may refer to non-directly related objects (i.e. non-sibling model elements sharing at least one ancestor). A special kind of non-containment relationship is *instanceOf*, which links a model element to the metamodel element that it conforms to. Since a metamodel is also an instance of a meta-metamodel (i.e. a metamodeling language such as Ecore), it may also be seen as a labeled directed graph containing objects that refer to each other, allowing for a homogeneous management of both models and metamodels.

2.2 Model Persistence

Models can be stored into persistence solutions for permanent storage using different approaches. These persistence solutions may be regular files (e.g. XMI), relational databases through object-relational mappings and, at a higher abstraction level, model repositories [8]. Modeling frameworks usually define persistence interfaces that allow client applications to access persisted models, e.g. the EMF Resource interface. These interfaces provide methods for fully loading, unloading and storing models and in some cases, loading single objects (e.g. EMF Resource's *getEObject* method). Storing a model consists in representing the object graph in the persistence solution and loading a model consists in rebuilding that graph at the client application. If the whole object graph is rebuilt, the model is fully loaded; otherwise, if only a subgraph (i.e. model partition) is loaded, the model is partially loaded. Client applications use these basic functions to access models and traverse them for different purposes. For example: a model-to-model transformation may search for a particular object that satisfies a given condition

and then traverse all its descedants; a model-to-code transformation may simply traverse a whole model, processing each object once or twice, etc.

A persistence solution provides *transparent integration* when client applications may access it using the persistence interface defined by the corresponding modeling framework without changing the models or metamodels, generating persistence-specific source code for metamodels, or any other form of specific pre or post-processing. For example, the XMI file-based persistence solution for EMF does not require generating metamodel-specific Java classes because it may use dynamic objects, which can be generically built at runtime.

2.3 The NoSQL Movement

The NoSQL [16] movement is composed of several specialized database paradigms that are used in very large web application scenarios such as Facebook, Google, Amazon, etc. In NoSQL, performance and scalability are more important than the ACID properties (Atomicity, Consistency, Isolation, Durability), proposing the BASE properties (Basically Available, Soft-state, Eventual consistency). Given the objectual structure of the data that are stored in some web applications, object-relational mappings have become an expensive solution that reduces their performance, while the different NoSQL databases are best suited for representing object models. There are also implementation differences between traditional relational databases and NoSQL databases, such as memory-based data storage instead of disk-based storage, logging and locking. [11].

The most used NoSQL database paradigms are key-value stores and document databases. *Key-value stores* have a simple data model in common: a map/dictionary allowing users to put and request values by key. They favor scalability over consistency and most of them omit rich querying and analytics features. A well-known key-value store is Amazon's Dynamo [12]. *Document databases* also use keys and values, but they are encapsulated into top-level structures called documents, which are schemaless. CouchDB [13] and MongoDB [14] are the major representatives of document databases. There is no standard query language in NoSQL; querying capabilities vary from one product to another. For example, CouchDB uses static view functions that implement the map/reduce data processing scheme [15], requiring a view function for each possible query; MongoDB uses a query-by-example approach through JSON documents and Dynamo queries consist simply in requesting values by their keys. The NoSQL movement has some features that are beneficial to our approach:

i. *Scalable*: as explained before, many MDE applications involve large models. Applications involving large amounts of data representing object models scale better in NoSQL than in relational databases [16].
ii. *Schemaless*: having no schemas means having no restrictions to co-evolve metamodels and models. Relational repositories usually create database schemas for each stored metamodel, difficulting their evolution and the conformance of existent models to the newer versions of their metamodels [8].

iii. *Accessible*: many NoSQL databases offer their data as JSON objects [17] through APIs that can be accessed via HTTP calls. This provides additional opportunities to access models from web browsers, web services, etc.

3 Overview

We propose Morsa, a persistence solution for managing large models. It relies on a document-based NoSQL database and integrates transparently with modeling frameworks. The architecture of our approach is shown in Figure 1. Morsa consists of a client and a NoSQL-based persistence backend.

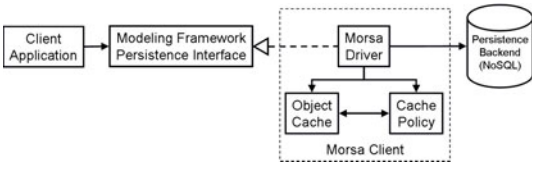

Fig. 1. Architecture of Morsa

The *client* side of Morsa supports tool integration through a driver that implements the modeling framework persistence interface, allowing client applications to access models in a standard way. Since Morsa is aimed at accessing large models, a *load on demand* mechanism has been designed to provide clients with efficient partial load of large models, achieving scalability [5]. This mechanism relies on an *object cache* that holds loaded model objects in order to reduce database queries and manage memory usage; it is managed by a configurable *cache replacement policy* that decides whether the cache is full or not and which objects must be unloaded from the client memory if needed. Section 5 discusses the model loading algorithm and the different cache replacement policies. On the *server* side, a NoSQL document database provides model persistence. We have chosen this kind of database because it provides a simple and natural way to map model elements (objects) to database elements (documents). Moreover, its schemaless architecture is beneficial for model persistence as stated above.

A running example is used to illustrate the design of our approach. It is based on the Grabats 2009 [18] reverse engineering case study, which is aimed at managing large models representing Java source code. A simplification of the JavaMetamodel metamodel provided by the contest is shown in Figure 2, representing Java projects, packages and types, and the source code declarations that are defined inside compilation units (i.e., .java files). The proposed test case was to retrieve every TypeDeclaration which contains a MethodDeclaration for a static and public method with the declared type as its returning type.

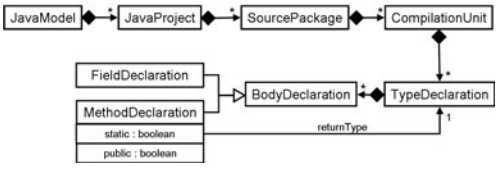

Fig. 2. Grabats 2009 contest JavaMetamodel metamodel simplification

4 Persistence Backend Design

Morsa relies on a document-based NoSQL database persistence backend. The main decision in its design was to choose the *granularity* of the documents, that is, how many documents are needed to represent a model. We have considered three alternatives of model granularity: one document per model, one document per object and one document per model partition.

i. A model can be represented as a *single document*. This is possible since the document-based NoSQL paradigm allows documents to store any number of objects, representing the structure of the model. However, this architecture may not scale for large models because it implies loading an entire model at once; it also has issues related to the maximum document size that some databases like MongoDB impose. Besides, querying single objects or partitions is cumbersome because nested objects are not globally visible.
ii. The opposite design, that is, *one document per object*, does not exploit the nesting capabilities of document-based NoSQL databases, but supports querying individual objects. However, object relationships have to be implemented using database references, that is, values that represent document identifiers, which are less efficient in time than nested objects. The resulting architecture would somehow resemble a relational schema, but it must be kept in mind that NoSQL is schemaless, so foreign keys between documents are far more flexible than the ones of the relational paradigm, since they may refer to any kind of model object.
iii. An intermediate solution would be to represent a model as a *set of documents* representing model partitions. Each model partition would be composed of objects that are always accessed together. Using partitions would speed up model loading because less database connections would be needed to load an entire model. Building these partitions requires access pattern analysis like the one explained in [7]; however, since the database partition is static, no optimal solution for every access pattern could be achieved.

Considering the previous discussion, we have designed Morsa using the second choice, that is, *a document per object*. A Morsa document is composed of a $<ID,$ *value, payload*$>$ tuple that where *ID* is the identifier of the object (object URI for EMF), *value* cointains the values of the object's features in a key-value format, where the *key* is the name of the feature and the *value* is the serialization of

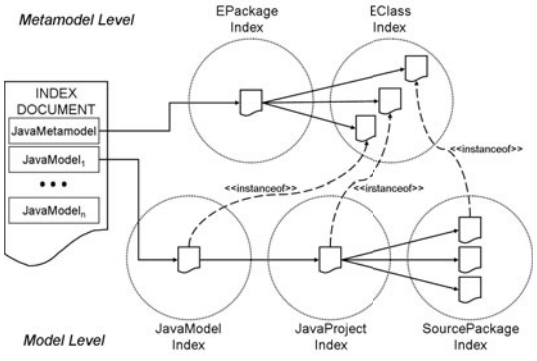

Fig. 3. Persistence backend structure excerpt for the running example

its value as a string; finally, *payload* specifies persistence-related metadata, such as references to the object's metaclass, the model's root object, etc. References to other objects are serialized as document references to their IDs. An index is created for every concrete metaclass, grouping their instances logically for faster queries, both for the meta-metamodel (e.g. an index for each metaclass of the Ecore meta-metamodel) and the metamodels (an index for each metaclass). Metamodels and models are represented homogeneously: documents representing model objects have references to the documents representing their corresponding metaclasses.

A (meta)model is represented as an entry in an *index document* that maps each (meta)model URI to an array of references to the documents that represent its root objects. This design is particulary useful for metamodeling languages like Ecore, where every object except the root ones must be contained by other objects, thus saving space in the index document.

Figure 3 shows an excerpt of the persistence backend structure for the running example. At the metamodel-level, the index for EPackage holds the document that corresponds to the root package of the JavaMetamodel (shown in Figure 2); this document references the documents that correspond to each metaclass (JavaModel, JavaProject, SourcePackage, etc.), which are held by the EClass index. At the model level, there is an index for each metaclass. The index document references a document representing a JavaModel, which is held by the JavaModel index; this document references a document that represents a JavaProject in its corresponding index and so on.

5 Model Loading

Our approach is intented to manipulate large models. In this paper we focus on the task of model loading, which involves three scenarios that require different approaches and algorithms: *full load*, *single load on demand* and *partial load on demand*. The load on demand scenarios have been tackled using an *object cache*

managed by a *cache replacement policy*. Metamodels are always fully loaded and kept in memory for efficiency reasons: they are relatively small compared to models and it is worth loading them once instead of accessing the database every time a metaclass is needed. Each object is identified in the database by a global ID attribute (object URI in EMF). A mapping between loaded objects and their IDs is held by the object cache in order to know which objects have been loaded, preventing the driver to load them again.

Consider a model that is small or medium-sized, hence it can be kept in memory by a client application. If the whole model is going to be traversed, it would be a good idea to load the model once, saving communication time with the persistence backend. We call this scenario *full load* and this is the way EMF works when loading XMI files. We aim at supporting full load with the least memory and time overhead possible. The Morsa full load algorithm works as the one for load on demand, which will be explained below, but considering an unlimited object cache, breadth and depth.

5.1 Load on Demand

Consider a model that is too large to be kept in memory by a client application; consider also a model that can be kept in memory but only a part of it is going to be traversed. A solution for both cases would be to load only the necessary objects as they are needed and then unload them to save client memory. We call this scenario *load on demand*. We define two kinds of load on demand: single load on demand and partial load on demand.

A *single load on demand* algorithm fetches objects from the database one by one. This behavior is preferred when the objects that need to be accessed are not closely related (i.e, they are not directly referenced by relationships) and memory efficiency is more important than network performance, that is, when the round-trip time of fetching objects from the database is not relevant. The resultant cache will be populated only with the traversed objects.

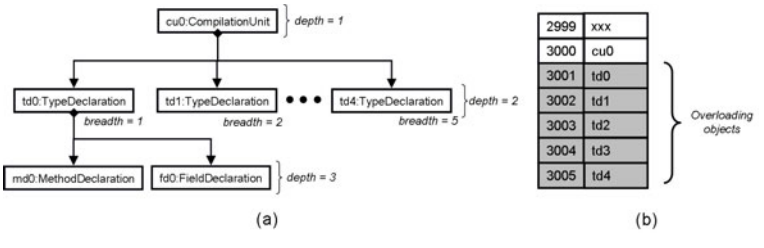

Fig. 4. Object loading in the running example: a) object model b) object cache

On the other hand, a *partial load on demand* algorithm fetches object clusters from the database. The structure of a cluster is customizable: given a requested object, its cluster may contain all its referenced objects, both directly and indirectly within a certain depth and breadth values. For example, when loading

the model shown in Figure 4(a), a partial load on demand algorithm configured with a maximum depth and breadth of 2 would load *cu0* (depth 1) and its two first contained TypeDeclaration objects, *td0* and *td1* (depth 2, breadth 2), but not *md0* nor *fd0* (depth 3). This behavior is preferred when all the objects that are related to an object will be traversed soon and memory efficiency is less important than network performance, that is, when the round-trip time of fetching objects from the database is critical. The resultant cache will be populated with the objects that have been traversed and those expected to be traversed in the near future. This is a simple form of prefetching that tries to take advantage of spatial locality. Our load on demand algorithm works as follows:

1. The client application requests an object by its ID
2. The Morsa driver fetches the document identified by that ID
3. A new object is created, filling its attributes with the values stored in the document and its references with proxies whose URIs refer to the referenced documents. A *proxy* is a special object that does not hold any feature value but an URI (containing the object ID and some persistence metadata such as database URL) that allows it to be resolved, i.e., filled with its actual values. In EMF, the idea of proxy is used to represent cross-resource references
4. The new object and its proxies are stored in the object cache, mapping them to their IDs
   (a) If single load on demand is being used, go to step 5
   (b) If partial load on demand is being used, the documents that correspond to the proxies are fetched all at once, saving networking time. The Morsa driver resolves these documents recursively following the two previous steps. This process stops if the cache becomes full or if the maximum depth and breadth is reached
5. If the cache becomes overloaded, some objects of the cache are unloaded
6. The new object is returned to the client application, which can use it as a regular object. When a reference is navigated and its value is a proxy, the resolution of that proxy is automatically requested, executing this algorithm

The size limit of the cache is configurable in terms of object counting, but this limit is soft because some modeling frameworks such as EMF require objects to have their references filled, that is, their values must be fetched in the form of proxies or actual objects. For example, consider Figure 4: an object cache containing 2999 elements is shown (b); its size limit is 3000 objects. Because the modeling framework requires an object to be fully filled, when *cu0* is loaded its 5 contained TypeDeclaration objects (*td0..td4*) must also be fetched as proxies, causing the cache to be overloaded with 3005 objects.

Whenever the cache becomes overloaded, the exceeding objects must be *unloaded*. A *cache replacement policy* algorithm selects the objects to be unloaded. Unloading an object also implies downgrading it to a proxy, i.e. unsetting all its features. A proxy requires less memory than a resolved object and it may be freed by the underlying language if it is not referenced by any other object.

5.2 Cache Replacement Policies

As introduced in the previous section, when the object cache becomes overloaded, a *cache replacement policy* algorithm selects which objects will be unloaded to free the client memory. We have considered four cache replacement policies:

i. A FIFO (First In-First Out) policy would unload the oldest objects in the cache. This policy is useful when a model is traversed in depth-first order, but only if the cache can hold the average depth of the model. On the contrary, it would cause objects to be unloaded after being traversed and then loaded again when requested for traversal.
ii. A LIFO (Last In-First Out) policy would unload the newest objects in the cache. This policy is useful when a model is traversed in breadth-first order, but only if the cache can hold the average breadth of the model. Both this and the FIFO policies calculate the size of the partition directly contained by the object that caused the cache overload and unloads that many objects. In the example of Figure 4, a LIFO policy would unload the objects in positions 3001 to 3005, while a FIFO policy would unload first 5 objects in positions 1 to 5.
iii. A LRU (Less Recently Used) policy would unload the least used objects in the cache. This policy is well known in the area of operating systems. It would be equivalent to a FIFO policy for depth-first and breadh-first orders.
iv. A LPF (Largest Partition First) policy would unload all the objects that conform the largest model partition contained in the cache. This is a conservative solution that is useful when a model is traversed in no specific order. It does not consider if the selected elements are going to be traversed so it may lead to multiple loads of the same objects. This policy unloads at least an amount of objects proportional to the maximum size of the cache.

The choice of which cache replacement policy is used is currently made by the end-user. However, this choice could be automatically made by the Morsa driver by analysis of (meta)models and access patterns (i.e. prefetching).

6 Integration and Implementation

Morsa is intented to be integrated with modeling frameworks and their applications. Our current prototype is integrated with EMF [3]. A transparent way of achieving this integration is to design the Morsa driver as an implementation of the persistence interface of the modeling framework (EMF Resource for EMF). Persisting a model in Morsa is done without any preprocessing, since there is no need of generating model-specific classes, modifiying metamodels or registering them into the persistence solution, as opposed to other approaches [8][9][19]. Metamodels are seamlessly persisted if they are not already in the database. Additional information for persistence configuration can optionally be passed to the driver; Morsa uses the standard parameters of the EMF load and save methods to pass this configuration information.

Morsa supports both *dynamic* and *generated* EMF. A dynamic model object is generated at runtime using EMF dynamic objects (instances of DynamicEObjectImpl) which use reflection to generically instantiate metaclasses. On the other hand, a generated model object is an instance of a metamodel-specific class that has been explicitly generated through an EMF generator model. Dynamic objects are preferred for tool integration since they do not require code generation. Other approaches [8] support only generated model objects reimplementing part of the EMF framework to handle persistency.

We have developed a prototype that exhibits some of the features described previously: EMF integration, single and partial load on demand, FIFO, LIFO and LPF cache replacement policies and full store. Its integration in EMF includes all the methods defined in the Resource interface and also methods for parent resolution (i.e., obtaining the container object of a given object) and special partial loading methods such as loading every instance of a metaclass.

We have chosen MongoDB [14] as the NoSQL database engine for our prototype; however, its architecture could be easily implemented in other engines. MongoDB has JSON access, dynamic queries (as opposed to the static views of CouchDB), server-side Javascript programming and uses BSON [17] objects for communication which provide fast and bandwith-efficient object transfer between the client and the database. MongoDB uses collections to logically organize documents, like the indexes introduced in Section 4. A collection is a set of documents which can be indexed by one or several attributes, allowing faster document access.

7 Related Work

Model persistence is not a novel research field. As the interest in MDE has grown many approaches have been proposed to solve this problem. The standard EMF solution is to persist models in XMI resources, but there are other alternatives. One approach is using binary indexed files [21]. Another approach is to use model repositories. A *repository* is a persistence solution remotely accesible by users and tools. Repositories usually rely on databases and provide additional features such as transactions and versioning. There are many EMF model repositories available today, being the most mature ones CDO [8], ModelBus [19] and EMFStore [9].

The *ModelBus* repository is a web service application that manages an embedded Subversion engine which implements the actual repository; however, Subversion is not designed to be integrated in client applications that access to parts of persisted elements, i.e., it does not support partial access to models. There have been attempts to make model access scalable in ModelBus [22]; however, the official release does not implement them. *EMFStore* implements a different architecture but shares the same philosophy as Subversion: models are fully loaded and stored by human clients using a GUI. This solution does not scale and it is best suited for design environments.

Currently *Connected Data Objects* (CDO) is the only model repository that is capable of managing large models using load on demand. CDO also provides

a rough version control system and EMF integration through its EMF Resource implementation, CDOResource. However, CDO is not application-transparent. First of all, we haven't been able to make it work with dynamic model objects, which is a severe drawback for its integration with EMF. Moreover, CDO requires metamodels to be pre-processed in order to persist their instances. One kind of pre-processing is to generate the Java model classes of a metamodel. This allows CDO to work with *legacy objects*. The other kind is to generate CDO-aware model classes from a generator model. This allows CDO to work with *native objects*. The main difference between legacy and native objects is that legacy objects cannot be demand-loaded or unloaded, having a huge impact on performance as will be shown in the next section. Native objects are unloaded from a CDO client when its memory becomes full using a *soft reference* approach, i.e. an object is removed by garbage collector when no other object refers to it with a reference that is not soft.

There are other domains where large and complex data needs to be accessed; for example, ontologies may be very large and complex and many solutions have been proposed, such as creating higher-level descriptions [23], which may be seen as a form of building views. Client scalability has been also tackled in the field of object-relational mappings, proposing prefetching mechanisms that load clusters of objects that will be used by the client application [24][25]. Object caching has also been a subject of study in the field of object databases, with mathematical approaches to optimizing cache coherence, replacement and invalidation [27][28]. Our approach could benefit from this reasearch to improve caching and prefetching with adaptive mechanisms. Finally, as far as we know, little or no research has been published on applying NoSQL to model persistence.

8 Evaluation

As stated in the previous section, CDO is the main alternative to our approach, so the evaluation consisted in executing a set of test cases with Morsa, CDO and the standard EMF XMI parser, comparing their performance results. We have considered the models proposed in the Grabats 2009 contest [18]. They conform to the JavaMetamodel metamodel that is shown simplified in Figure 2. There are five models, from Set0 to Set4, each one larger than its predecessor (from a 8.8MB XMI file with 70447 model elements representing 14 Java classes to a 646MB file with 4961779 model elements representing 5984 Java classes).

Two benchmarks have been executed: model access and model query. The model access benchmark consists in traversing models in depth-first order and breadth-first order. The model query benchmark executes the query proposed in the Grabats contest, which searches for every class that declares a public static method whose returning type is that same class. Each benchmark has been executed using the EMF XMI loading facility, a CDO repository configured for best speed and least memory footprint in legacy mode and native mode and Morsa for least memory footprint and best speed using single and partial load on demand. All tests have been executed under a Intel CoreI5 760 PC at 2.80GHz

with 8GB of physical RAM running 64-bit Linux 2.6.35 and JVM 1.6.0. CDO 4.0 is configured using DBStore over a dedicated MySQL database and is used in read-only mode in order to avoid versioning overhead.

8.1 Results

Table 1 shows the results for the model access test cases. Memory footprint is shown in Megabytes and time is shown in seconds. The *Opt* column specifies whether the configuration optimizes speed or memory. As expected, CDO Native mode is more efficient than CDO Legacy mode, but still Morsa is faster and uses less memory for all the models. Note that the minimum memory used by CDO for the Set1 breadth-first order doubles the memory needed by XMI, while Morsa uses 20% less memory than XMI. We haven't been able to load the Set2 model (271MB XMI file, containing 2082481 model elements representing 1605 Java classes) with CDO within a reasonable time (less than 45 minutes). The cache replacement policies used for least memory footprint in Morsa were a LIFO policy for breadth-first order and a FIFO policy for depth-first order. Cache size was 900 objects for load on demand and unlimited for full load (best speed). CDO was configured with a maximum available memory of 70MB for the Set0 model and 30MB and 100MB for the Set1 in depth-first order and breadth-first order, respectively, for the least memory footprint and unlimited memory for the best speed. These configurations have been obtained empirically.

For all models (including Set3 and Set4, which are not shown), Morsa is much slower than XMI, but still can load and traverse them entirely. In the best case for the Set2 model, Morsa uses 17 times less memory than XMI spending 20 times more time. Note that with an unlimited cache, Morsa spends a similar time than the best speed case with a small one (1.5% time difference). This is due to the fact that with an unlimited cache, our prototype holds references to every model object, difficulting garbage collection. On the other hand, a cache with limited size unloads objects more often, facilitating the garbage collection. Since we haven't been able to store the Set3 and Set4 models in CDO, despite assigning it the maximum available memory (Set2 could be stored, but causing an exception on commit), the model access test cases for these models are not shown. For these models, XMI is faster than Morsa but needs much more memory.

The potential of Morsa shows up not only with a limited amount of memory, but also when models do not have to be completely traversed. For example, the Grabats 2009 contest query, whose execution results are shown in Table 2, shows that Morsa is more efficient in memory and time than CDO and XMI for this particular task. These results illustrate that our approach can be very beneficial for applications that do not need to process an entire model, such as certain model transformations. An application can query Morsa for specific objects, consuming less time and memory and achieving scalability. The query has been implemented using dynamic EMF for Morsa and generated model classes for CDO and XMI. CDO and Morsa allow querying the database for all instances of a given metaclass and then traversing the results to check the query condition, while XMI requires loading the entire model prior to its traversal.

Table 1. Performance results for the model access test cases

Order	Opt	Solution	Mode	Set0		Set1		Set2	
				Mem	Time	Mem	Time	Mem	Time
-	-	XMI	-	63	1.313	113	2.265	1257	15.632
Depth	-	CDO	Legacy	162	32.156	516	91.136	-	-
Breadth	-	CDO	Legacy	172	31.609	444	92.160	-	-
Depth	Speed	CDO	Native	289	21.783	435	59.188	-	-
Breadth	Speed	CDO	Native	308	21.046	467	56.017	-	-
-	Speed	Morsa	Full	113	8.762	363	26.671	1300	317.331
Depth	Mem	CDO	Native	59	31.218	87	80.594	-	-
Depth	Mem	Morsa	Single	25	12.130	32	32.348	92	313.027
Depth	Mem	Morsa	Partial	30	14.163	29	39.197	98	410.829
Breadth	Mem	CDO	Native	59	30.010	250	78.204	-	-
Breadth	Mem	Morsa	Single	32	18.889	90	31.530	400	322.045
Breadth	Mem	Morsa	Partial	40	29.239	96	85.197	460	761.692

Table 2. Performance results for the query test case

Opt	Solution	Mode	Set0		Set1		Set2		Set3		Set4	
			Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time
-	XMI	-	70	1.513	121	2.465	1265	16.023	2940	81.340	3512	141.752
Speed	CDO	Native	7	0.445	23	0.968	129	18.149	-	-	-	-
Speed	Morsa	Single	5	0.706	8	0.985	168	9.724	205	26.760	254	29.339
Mem	CDO	Native	4	0.545	6	1.731	61	25.798	-	-	-	-
Mem	Morsa	Single	5	0.706	5	1.518	36	14.822	96	36.944	59	40.129

9 Conclusions and Further Work

We have presented Morsa, a persistence solution aimed at achieving scalability for client applications that access large models. Morsa uses load on demand mechanisms to allow large models to be accessed without overflowing the client application memory. We have developed several cache replacement policies that cover different model access patterns. Server scalability is achieved using a document-based NoSQL database, which is a novel feature since model repositories usually work with object-relational mappings. As far as we know, applying document-based NoSQL databases to MDE has not been proposed before, and is a promising approach to build industrial-scale model persistence solutions.

We have implemented a prototype for EMF that in its early development stage shows promising performance results. An evaluation of our prototype is shown, executing two benchmarks against large models and comparing its results with the ones of XMI and the well-stablished CDO repository. This comparison shows that Morsa suits better for partial model access and model querying than XMI and CDO, and that it handles larger models than CDO does.

Our future work is to continue optimizing Morsa while implementing new features. Among others, these features include: *incremental store*, that will allow the client to store changes done to objects that are going to be unloaded, an advanced *query API*, support for *query languages* such as OCL and making our load on demand algorithms and cache replacement policies more adaptative by *collecting metadata information* about the structure of the persisted models.

References

1. Mohagheghi, P., Fernandez, M.A., Martell, J.A., Fritzsche, M., Gilani, W.: MDE Adoption in Industry: Challenges and Success Criteria. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 54–59. Springer, Heidelberg (2009)
2. Baker, P., Loh, S.C., Weil, F.: Model-Driven Engineering in a Large Industrial Context — Motorola Case Study. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 476–491. Springer, Heidelberg (2005)
3. The Eclipse Modeling Framework, `http://www.eclipse.org/emf`
4. Kolovos, D., Paige, R., Polack, F.: Scalability: The Holy Grail of Model-Driven Engineering. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 35–47. Springer, Heidelberg (2009)
5. Selic, B.: Personal Reflections on Automation, Programming, Culture and Model-based Software Engineering. Automated Software Engineering 15(3-4), 379–391 (2008)
6. Canovas, J., Garca, J.: An architecture-driven modernization tool for calculating metrics. IEEE Software 27(4), 37–43 (2010)
7. Varro, G., Friedl, K., Varro, D.: Adaptive Graph Pattern Matching for MOdel Transformations using Mode-sensitive Search Plans. ENTCS 152, 191–205 (2006)
8. The CDO Model Repository, `http://www.eclipse.org/cdo`
9. EMFStore: A model repository for EMF models. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Cape Town (South Africa) vol. 2, pp. 307–308 (2010), `http://www.emfstore.org`
10. Morsa prototype, `http://www.modelum.es/morsa`
11. Stonebraker, M.: SQL Databases vs NoSQL Databases. Communications of the ACM 53(4), 10–11 (2010)
12. DeCandia, G., Hastorun, D., et al.: Dynamo: Amazon's Higly-Available Key-value Store. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, pp. 205–220. ACM, New York (2007)
13. CouchDB: couchdb.apache.org
14. MongoDB, `http://www.mongodb.org`
15. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters (2004), `http://labs.google.com/papers/mapreduce-osdi04`
16. Strauch, C.: NoSQL Databases. Stuttgart Media University (2011), `http://www.christof-strauch.de/nosqldbs.pdf`
17. JavaScript Object Notation, `http://www.json.org`
18. Grabats 2009 5th International Workshop on Graph-Based Tools: a reverse engineering case study, Zurich (Switzerland) (July 2009), `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/`
19. Blanc, X., Gervais, M.-P., Sriplakich, P.: Model Bus: Towards the Interoperability of Modelling Tools. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 17–32. Springer, Heidelberg (2005), `http://www.modelbus.org`
20. Binary JSON, `http://www.bsonspec.org`
21. Jouault, F., Sottet, J.: An AmmA/ATL Solution for the Grabats 2009 Reverse Engineering Case Study. In: Grabats 2009 5th International Workshop on Graph-Based Tools, Zurich, Switzerland (July 2009)
22. Sriplakich, P., Blanc, X., Gervais, M.: Collaborative Software Engineering on Large-scale models: Requirements and Experience in ModelBus. In: Proceedings on the 2008 ACM Symposium on Applied Computing, pp. 674–681. ACM, New York (2008)

23. Bhm, C., Lorey, J., Fenz, D., Kny, E., Pohl, M., Naumann, F.: Creating voiD Descriptions for Web-scale Data. Winner of the 2010 Billion Triple Track Semantic Web Challenge (2010)
24. Ibrahim, A., Cook, W.: Automatic by Traversal Profiling in Object Persistence Architectures. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 50–73. Springer, Heidelberg (2006)
25. Han, W., Whang, K., Moon, Y.: A Formal Framework for Prefetching Based on the Type-Level Access Pattern in Object-Relational DBMSs. IEEE Transactions on Knowledge and Data Engineering 17, 1436–1448 (2005)
26. Chang, F., Dean, J., et al.: Bigtable: A Distributed Storage System for Structured Data (2006)
27. Leong, H., Si, A.: On Adaptive Caching in Mobile Databases. In: Proceedings of the 1997 ACM Symposium on Applied Computing, pp. 302–309. ACM, New York (1997)
28. Rathore, R., Prinja, R.: An Overview of Mobile Database Caching (2008), http://www-users.cs.umn.edu/ rohinip/Rohini_Prinja/Research_files/8701Project.pdf