Automated and Transparent Model Fragmentation for Persisting Large Models

Markus Scheidgen and Anatolij Zubow

Department of Computer Science, Humboldt Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
{scheidge,zubow}@informatik.hu-berlin.de

Abstract. Existing model persistence frameworks either store models as a whole or object by object. Since most modeling tasks work with larger aggregates of a model, existing persistence frameworks either load too many objects or access many objects individually. We propose to persist a model broken into larger fragments.
First, we assess the size of large models and describe typical usage patterns to show that most applications work with aggregates of model objects. Secondly, we provide an analytical framework to assess execution time gains for partially loading models fragmented with different granularity. Thirdly, we propose meta-model-based fragmentation that we implemented in an EMF based framework. Fourthly, we analyze our approach in comparison to other persistence frameworks (XMI, CDO, and Morsa) based on four common modeling tasks: create/modify, traverse, query, and partial loads.
We show that there is no generally optimal fragmentation, that fragmentation can be achieved automatically and transparently, and that fragmentation provides considerable performance gains compared to other persistence strategies.

1 Introduction

Modeling frameworks (e.g. the Eclipse Modeling Framework (EMF) [21] or Kermeta [11]) can only work with a model when it is fully loaded into a computer's main memory (RAM), even though not all model objects are used at the same time. This limits the possible size of a model. Modeling frameworks themselves provide only limited capabilities to deal with large models (i.e. resources and resource lazy loading in EMF [21]). Model persistence frameworks (e.g. Connected Data Objects (CDO) [1]), on the other hand, store models in databases and load and unload single model objects on demand. Only those objects that are used at the same time need to be maintained in main memory at the same time. This allows to work with models larger than the main memory would otherwise allow.

We claim that existing model persistence solutions may provide a main memory efficient solution to the model size issue, but not a time efficient one. In this paper, time efficiency always relates the time it takes to execute of one of four abstract modeling tasks. These tasks are (i) creating/modify models, (ii) traverse models (e.g. as necessary during model transformation), (iii) query models, and (iv) partially loading models (i.e. loading a diagram into an editor).

An obvious observation is that some of these modeling tasks (especially traversing models and loading parts of models) require to load large numbers of model objects eventually. Existing persistence frameworks, store and access model objects individually. If a tasks requires to load a larger part of the model, all its objects are still accessed individually from the underlying database. This is time consuming.

Our hypothesis is that modeling tasks can be executed faster, if models are mapped to larger aggregates within an underlying database. Storing models as aggregates of objects and not as single objects reduces the number of required database accesses, or as Martin Fowler puts it on his blog: *"Storing aggregates as fundamental units makes a lot of sense* [...]*, since you have a large clump of data that you expect to be accessed together"*, [7]. This hypothesis raises three major questions: Do models contain aggregates that are often *accessed together*? How can we determine aggregates automatically and transparently? What actual influence on the performance has the choice of concrete aggregates?

To answer these question, we will proceed as follows: First (section 2), we look at three typical modeling applications: which model sizes they work with and what concrete modeling tasks they perform predominantly. This will give us an idea of what aggregates could be and how often objects can be expected to be actually accessed as aggregates. Secondly (section 3), we will present our approach to finding aggregates within models. This approach is based on fragmenting models along their containment hierarchy. We will reason, that most modeling tasks need to access sub-trees of the containment hierarchy (fragments). In the related work section 4, we present existing model persistence frameworks and interpret their strategies with respect to the idea of fragmentation. Furthermore, we discuss key-value stores as a basis for persisting fragmented models. The following section provides a theoretical analysis and upper bound estimation for possible performance gains with optimal fragmentation. In section 6, we finally present a framework that implements our fragmentation concept. The next section is the evaluation section: we compare our framework to existing persistence frameworks with respect to time and memory efficient execution of the four mentioned abstract modeling tasks. Furthermore, we use our framework to measure the influence of fragmentation on performance to verify the analytic considerations from section 3. We close the paper with further work and conclusions.

2 Applications for Large Models

In this section, we look at examples for three modeling applications. We do this for two reasons. The first reason is to discuss the actual practical relevance of large models. The second reason is to identify model usage patterns: which of the four modeling tasks (create, traverse, query, partial load) are actually used, in what frequency, and with what parameters. At the end of this section, we provide a tabular summary of our assessment.

2.1 Software Models

Model Driven Software Development (MDSD) is the application that modelling frameworks like EMF were actually designed for. In MDSD all artifacts including traditional software models as well as software code are understood as models [22], i.e. directed labelled graphs of typed nodes with an inherent containment hierarchy.

Model size: Since models of software code (code models) provide the lowest level of abstraction, we assume that models of software code are the largest software models. In [**?**], we give an approximation for the size of code models based on counting abstract syntax tree nodes in the Linux kernel and analyzing the Linux kernels GIT repository. We also transferred all ratios learned from the Kernel to other OS software projects and publicly reported LOC counts. The results are presented in Fig. 1.

Usage patterns: There are two major use cases in today's software development: editing and transforming or compiling. The first use case is either performed on diagrams (graphical editing) or on compilation units (e.g. Java-files, textual editing). Diagram contents roughly corresponds to package contents. Both packages and compilation units are sub-trees within the containment tree of a software model. Transformations or compilations are usually either done for the whole model or again on a per package or compilation unit basis. Within these aggregates, the (partial) model is traversed. A further use-case is analysis. Analysis is sometimes performed with single queries. But due to performance issues, model analysis is more often performed by traversing the model and by executing multiple queries with techniques similar to model transformations. Software models are only accessed by a few individuals at the same time.

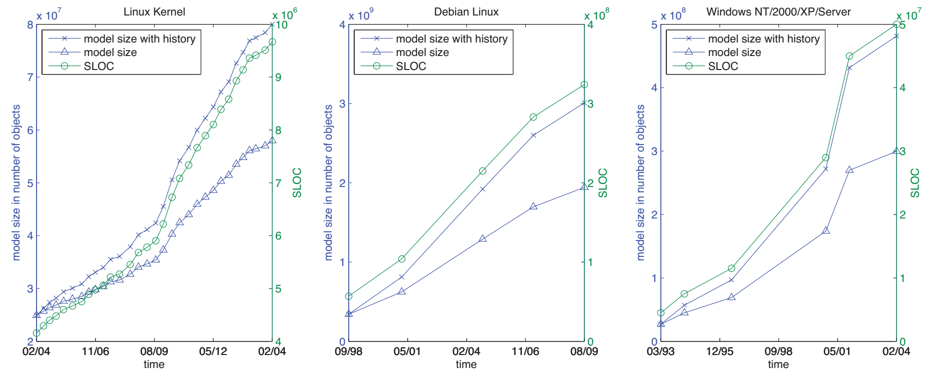

Fig. 1. Rough estimates for software code model sizes based on actual SLOC counts for existing software projects.

2.2 Heterogeneous Sensor Data

Sensor data usually comprises of time series of measured physical values in the environment of a sensor. Our research group build the *Humboldt Wireless Lab* [23], a 120 node wireless sensor network that produces heterogeneous sensor data: data from a 3 axis accelerometers, data from monitoring all running software components (mostly networking protocols), and other system parameters (e.g. CPU, memory, or radio statistics). We represent and analyze this data with EMF based models ([18]).

Model size: HWL's network protocols and system software components provide 372 different types of data sets. Each data set is represented as an XML document. Per second each node in the network produces XML entities that translate into an average of 1120 EMF objects. A common experiment with HWL involves 50 nodes and measures of a period of 24 h. During such an experiment, the network produces a model of 5×10^9 objects.

Usage patterns: There are two major use-cases: recording sensor data and analyzing sensor data. Recording sensor data means to store it faster than it is produced. If possible in a manner that supports later analysis. Sensor data is rarely manipulated. Analysis means to access and traverse individual data sets (mostly time series). Each data set or recorded set of data sets is a sub-tree in the sensor data model. Recording and analysis is usually performed by only a single (or a few) individuals at the same time.

2.3 Geo-spatial Models

3D city models are a good example for structured geo-spatial information. The CityGML [8] standard, provides a set of XML-schemata (building upon other standards, e.g. GML) that function as a meta-model. CityGML models represent the features of a city (boroughs, streets, buildings, floors, rooms, windows, etc.) as a containment hierarchy of objects. Geo spatial models usually come in different levels of details (LOD); CityGML distinguishes 5 LODs, 0-4) [8].

Model size: As for many cities, a CityGML model is currently established for Berlin [20]. The current model of Berlin covers all of Berlin, but mostly on a low-medium level of detail (LOD 1-2). To get an approximation of the model's size, we counted the XML entities. The current Berlin model, contains 128×10^6 objects. Based on numbers and average sizes per feature sizes in the Berlin model, a complete LOD 3-4 model of Berlin would consist of 10^9 objects. Extrapolating numbers to the world's population that lives in cities, a LOD3-4 *world 3D city model* would contain 10^{12}.

Usage patterns: Compared to model manipulation, model access is far more common and its efficient execution is paramount. If accessed, users usually load a containment hierarchies (sub-tree) corresponding to a given set of coordinates or address (geographic location): partial loads. Queries for distinct feature characteristics within a specific geographic location (i.e. with-in such a partial load) are also common. Geo-spatial models are accessed by many people at the same time.

Summary

The following table summarizes this section. Two + signs denote that execution times of the respective tasks are vital for the success of the application; a single + denotes that the task is executed often, but performance is not essential; a − denotes that the task is of minor importance.

application	**model size**	**create/mod.**	**traverse**	**query**	**partial load**
software models	$0 - 10^9$	+	++	+	+
sensor data	10^9	++	++	-	++
geo-spatial models	$10^9 - 10^{12}$	-	-	++	++

3 Model Fragmentation

3.1 Fragmentation in General

All models considered in this paper can be characterized as directed labeled graphs with a fix spanning-tree called *containment hierarchy*. In EMF based models, the containment hierarchy consists of *containment references*; other graph edges are *cross-references*.

Model fragmentation breaks (i.e. *fragments*) a model along its containment hierarchy. All *fragments* are disjoint; no object is part of two fragments. Fragmentation is also always complete, i.e. each object is part of one fragment. The set of fragments of a model is called *fragmentation*. References between fragments are called inter-fragment and references within a fragment are called intra-fragment references. [1]

3.2 Fragmentation Strategies

Originally a model is not fragmented; once it was fragmented, the fragmentation needs to be maintained when the model is modified. Further, we have to assume that fragmentation has an influence on performance (refer to sections 5 and 7). We denote a set of algorithms that allows to create and maintain a fragmentation as *fragmentation strategy*.

There are two trivial strategies: *no fragmentation* and *total fragmentation*. No fragmentation means the whole model constitutes of one fragment, such as in EMF (without resources). Total fragmentation means each object constitutes its own fragment. There are as many fragments as objects in the model. This strategy is implemented by existing persistence frameworks like CDO.

[1] Based on these characteristics, fragments can be compared to EMF's resources (especially with containment proxies); refer to section 6, where we use resources to realize fragmentation.

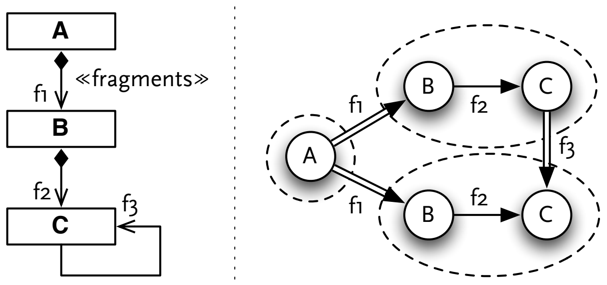

Fig. 2. Example meta-model (left) and model (right). In the model: dashed ellipses denote fragments, double lines inter- and normal lines intra-fragment references. The references of feature *f1* determine the fragments, the reference of *f3* is a inter-fragment cross-reference by accident.

3.3 Meta-Model based Fragmentation

In this paper, we propose and use *meta-model based fragmentation* as fragmentation strategy. A meta-model defines possible models by means of classes and their attribute as well as reference features. Whereby, the meta-model determines which reference features produce containment and which produce cross-references. The meta-modeler already uses containment reference features to aggregate related objects.

In meta-model based fragmentation, we ask the meta-modeler to additionally mark those containment reference features that should produce inter-fragment containment references. This way, the meta-model determines where the containment hierarchy is broken into fragments, and it becomes easy to create and maintain fragmentations automatically and transparently (ref. to section 6). Only containment reference features determine fragmentation, cross-references can become inter-fragment references by accident. See Fig. 2 for an example.

4 Related Work

4.1 Model Persistence

EMF: Models are persisted as XMI documents and can only be used if loaded completely into a computer's main memory. EMF realizes the *no fragmentation* strategy. The memory usage of EMF is linear to the model's size.

There are at least three different approaches to deal with large EMF models: (1) EMF resources, where a resource can be a file or an entry in a database; (2) CDO [1] and other object relational mappings (ORM) for Ecore; (3) morsa [15] a EMF data-base mapping for non-relational databases.

First, EMF resources [21]: EMF allows to fragment a model into different resources. Originally, each resource could only contain a separate containment hierarchy and only inter-resource cross-references were allowed. But since EMF version 2.2 containment proxies are supported. EMF support lazy loading: resources do not have to be loaded manually, EMF loads them transparently once objects of a resource are navigated to. Model objects have to be assigned to resources manually (*manual fragmentation*). To actually save memory the user has to unload resources manually too. The framework MongoEMF [10] maps resources to entries in a MongoDB [16] database.

Secondly, CDO [1]: CDO is a ORM for EMF. [2] It supports several relational databases. Classes and features are mapped to tables and columns. CDO was designed for software modeling and provides transaction, views, and versions. Relational databases provide mechanisms to index and access objects with SQL queries. This allows fast queries, if the user understands the underlying ORM.

Thirdly, morsa [15]: Different to CDO, Morsa uses mongoDB [16], a *NoSQL* database that realizes a key-value store (see below). Morsa stores objects, their references and attributes as JSON documents. Morsa furthermore uses mongoDB's index feature to create and maintain indices for specific characteristics (e.g. an objects meta-class reference).

4.2 Key-Value Stores

Web and cloud computing require scaleability (replication and sharding[3] in a peer-to-peer network) from a database, and traditional ACID [9] properties can be sacrificed if the data store is easily distributeable. This explains the popularity of *key-value stores*. Such stores provide only a simple map data structure: there are only keys and values. For more information and an comparison of existing key-value stores refer to [14].

Model fragmentation dos not need any complex database structure, since a fragment's content can be serialized (e.g. with XMI) and fragments can be identified by keys (e.g. URIs). Key-value stores on the other hand provide good scaleability for large models (sharding) or for parallel access (replication).

There are three different applications that inspired three groups of key-value stores. First, there are web applications and the popular MongoDB [16] and CouchDB [3] databases. These use JSON documents as values and provide additional indexing of JSON attributes.

Secondly, there is cloud computing and commercial Google Big-Table [4] and Amazon's Dynamo [6] inspired data stores. HBase [12] and Cassandra [13] are respective open source implementations. Those databases strive for massive distribution, they provide no support for indexing inner value attributes, but integrate well into map-reduce [5] execution frameworks, such as Hadoop (HBase is Hadoop's native data store).

A third application is high performance computing. Scalaris [19] is a key-value store optimized for massive parallel, cluster, and grid computing. Scalaris

[2] Lately, CDO also supports non-relational databases, such as MongoDB [16]. Such features were not evaluated in this paper; but one can assume characteristics similar to those of Morsa.

[3] *Sharding* denotes horizontal partitioning of a database, i.e. to put different parts of the data onto different nodes in the network

provides mechanisms for consistency and transactions and brings some ACID to key-value stores.

5 Possible Performance Gains from Model Fragmentation

In this section, we analyze the theoretically possible execution times of partially loading models with fragmentations of different granularity. This includes an assessments for performance gains from optimal fragmentation strategies compared to no or total fragmentation.

To keep this analysis simple, we have to make two assumptions that will probably seldom hold in reality, but still lead to analysis results that provide reasonable upper bounds for possible gains. The first assumption: we are only consider fragmentations where all fragments have the same size f. This means a fragmentation for a model of size m consist of $\lceil m/f \rceil$ fragments[4]. The second assumption: all fragmentations are optimal regarding partial loads. This means to load a model part of size l, we only need to load $\lceil l/f \rceil$ fragments at most.

To determine the execution time for partial loading depending on the parameters model size m, fragment size f and size of the model part l, we need two functions that determine the time it takes to read and parse a model and to access a value in a key value store. The read and parse function is linear depending on parsed model size s: $parse(s) = \mathcal{O}(s)$, the access function is logarithmic depending on the number of keys k: $access(k) = \mathcal{O}(log(k))$. Most key-value stores, including HBase (that we use for our implementations) provide $\mathcal{O}(log)$ accesses complexity (ref. also to Fig. 4).

With the given assumptions, parameters, and functions the time to execute a partial load is:

$$t_{m,f}(l) = \overbrace{\left\lceil \frac{l}{f} \right\rceil}^{\text{number of fragments to load}} \underbrace{\left(access(\left\lceil \frac{m}{f} \right\rceil) + parse(f) \right)}_{\text{time to load one fragment}}$$

To actually use this cost function, we need concrete values for *parse* and *access*. We measured the execution times for *parse* with EMF's XMI parser for models of various sizes and fit a linear functions to the measured values (Fig. 3). For *access* we measured the execution time for accessing keys in HBase for database tables with various numbers of keys k. For $k < 10^6$ we use a linear function and for $k \geq 10^6$ a logarithmic function as a fit (Fig. 4).

Now, we can discuss the influence of fragment size f on $t_{m,f}(l)$. First, we use a model of size $m = 10^6$ and vary $f \in \{10^0, \ldots, 10^6\}$. Fig. 5 shows the computed times t over loaded model objects l for the different fragment sizes f. We can observe four things. First, there is no optimal fragment size. Depending on the number of loaded objects, different fragment sizes are optimal. But intermediate fragment sizes provide good performance. With fragment size $f = 10^2$ for

[4] $\lceil x \rceil$ denotes the ceiling of x

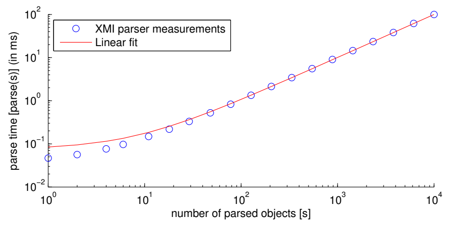

Fig. 3. EMF's XMI parser performance.

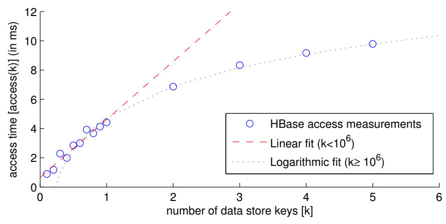

Fig. 4. HBase accesses performance.

example, all partial loads take three times the optimal time at most. Second, total fragmentation ($f = 1$) requires roughly 100 times more time than optimal fragmentation, when larger numbers of objects $\geq 10^2$ are loaded. Thirdly, no fragmentation ($f = m$) is only a time efficient option, if we need to load almost all of the model. But in those cases no fragmentation is usually not practical for memory issues. Fourthly, for small partial models total fragmentation is far better than no fragmentation, for large partial models no fragmentation provides better performance.

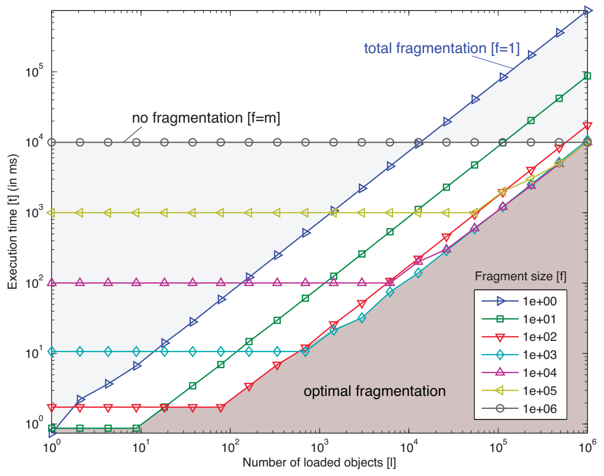

Fig. 5. Computed execution times for partial loads from a model with 10^6 objects and fragmentations of different granularity.

6 Implementation of Model Fragmentation

In this section, we present the EMF based persistence framework EMFFrag which implements the presented meta-model based fragmentation strategy (refer to section 3).

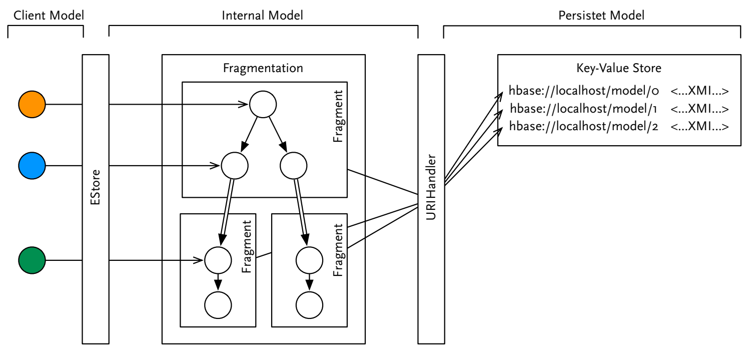
Client Model
Internal Model
Persistet Model
EStore
Fragmentation
Fragment
Fragment
Fragment
URIHandler
Key-Value Store
hbase://localhost/model/0 <...XMI...>
hbase://localhost/model/1 <...XMI...>
hbase://localhost/model/2 <...XMI...>

Fig. 6. EMFFrag partially loads a persisted model as internal model of dynamic EMF objects and exposes the model as client model via EMF generated model code with feature delegation.

Design goal: The main goal in our implementation is to (re)use EMF resource as much as possible. EMF resources already provide many required functionalities: they realize partial model persistence, resources manage inter-resource references through proxies, resources lazy-load, they can be added and deleted, and objects can be moved between resources. EMFFrag extends the existing implementations of EMF resources. EMFFrag could be realized with a very small code base of less than 900 lines of code.

Underlying key-value store: EMFFrag uses a simple interface that abstracts from concrete key-value stores. We provide an implementation for HBase (this was used for all measurements in this paper). EMFFrag implements EMF's `URIHandler` interface to realize key-value store values as resources. Each fragmented model is stored in its own table.

Fragments and fragmentation: EMF `XMIResource`s are used as fragments and `ResourceSet`s act as fragmentations. The model is internally realized as a purely dynamic (no generated sources) EMF model.

Transparent load and unload of fragments: Fragments, Fragmentations, and internal model are hidden from clients (ref. to Fig. 6). Clients use the model through the usual EMF generated interfaces and classes. Those are configured with reflective feature delegation to an `EStore` ([21] explains the concept). EMFFrag's `EStore` implementation simply delegates all calls to internal objects. If necessary, it creates an internal object for each client created object, and a client object for each internal object. Client objects hold references to their internal counter parts. Fragments manage client objects that correspond to the internal objects they contain via Java's *weak references*. When clients loose all strong references to a fragment's contents, the JVM collects the client objects as garbage (despite existing weak references) and notifies the owning fragment. Thus, fragments know if clients hold references to their objects, and they can safely unload once no more client reference to their contents exist no more.

Inter-fragment containment references: Client model classes have to be generated with enabled containment proxies (see [21]) to allow containment references between resources (i.e. fragments). Users can use EMF Ecore annotation to mark containment reference features as inter-fragment features. When EMFFrag's `EStore` implementation delegates a call that manipulates an inter-fragment containment feature, it creates or deletes fragments accordingly and puts objects into their respective fragments.

Inter-fragment cross references: EMF persists references between XMI resources with URIs. The first part of an URI identifies the resource (i.e. the fragment within a key-value store). The second URI part (URI fragment part) identifies the referenced object within the containing resource. For all inter-fragment containment references and for cross references within a fragment EMF's default *intrinsic ID's* [21] are used.

Intrinsic IDs are similar to XPath expressions and identify an object via its position in the containment hierarchy. Intrinsic IDs cannot be used for inter-fragment cross references: when an object is moved, its intrinsic ID (URI fragment) changes and all persisted referencing object use invalid URIs. For this reason EMFFrag uses model-wide unique *extrinsic IDs* (an existing EMF functionality). EMFFrag maintains a secondary index (i.e. another table in the key-value store) that maps extrinsic IDs to respective intrinsic IDs. When an object moves this entry is updated and all cross-references are updated automatically. Extrinsic IDs and secondary index are only maintained for objects that are actually cross referenced from another fragment to keep the index small.

7 Evaluation

This section has two goals. First, we want to compare our fragmentation approach to other model persistence frameworks. Secondly, we want to verify our findings from section 5. All measurements were performed on a Notebook computer with Intel Core i5 2.4GHz CPU, 8 GB 1067 MHz DDR3 RAM, running Mac OS 10.7.3. All experiments were repeated at least 20 times, and all present results are respective averages. Code executing all measurements and all measured data can be downloaded as part of EMFFrag [17].

7.1 Fragmentation Compared to other Persistence Frameworks

To compare fragmentation to EMF's XMI implementation, CDO, and Morsa, we measured execution time for the three tasks (i) create/modify, (ii) traverse, and (ii) query. To analyze traverse and query, we used example models from the Grabats 2009 contest [2] as benchmarks. Those were already used to compare Morsa with XMI and CDO here [15]. There are five example models labeled *set0* to *set4* and they all model Java software based on the same meta-model. Please note: even though the models increase in size, their growth is not linear and the internal model structure is different. To measure create/modify performance, we used a simple test model. We don't provide any comparative measures for partial loads. Partial loads are extensively measured for EMFFrag in the next section.

Fig. 12 shows the number of fragments that each framework produces for each model. Morsa and CDO implement total fragmentation and the number of fragments is also the number of objects in the model. For XMI there is always only one fragment, because it implements *no fragmentation*. For EMFFrag, we provided two different meta-model based fragmentations. The first one puts each Java compilation unit and class file into a different fragment (labeled *EMFFrag coarse*). The second one additionally puts the ASTs for each method block into a different fragment (*EMFFrag fine*). The number of fragments differs significantly for *set2* and *set3* which have to contain a lot of method definitions. We could not measure CDO's performance for *set3* and *set4*: the models are to large to be imported with a single CDO transaction, and circular cross-references do not allow to import the model with multiple transactions.

Create/modify: We measured the performance of instantiating and persisting objects. A meta-model with only one class was used. We created test models with 10^5 objects, a binary containment hierarchy, and two different densities of cross references: one cross reference per object and no cross references. We used a transaction size of 10^3 objects for CDO and a fragment size of 10^3 for EMFFrag.

Fig. 7.1 shows the average number of objects that could be persisted within one second. The number of cross-references has only a minor influence on the performance of CDO, Morsa, and EMFFrag. EMFFrag is a little slower than XMI depending on the fragment size (Fig. 8). CDO and Morsa (both based on complex indices that have to be maintained) can only create less than one tenth of the objects per seconds that could be created with XMI and EMFFrag. Fig. 8 shows EMFFrag's create performance for different fragment sizes.

Traverse: Fig. 9 shows the measurement results in traversed objects per second. XMI performs well for small models, but numbers deteriorate for large models. Interestingly, Morsa and CDO both use *total fragmentation* and achieve both a comparable low 4,500 objects per second. EMFFrag performs depending on the number of fragments: the less fragments the better. With the Grabats models, fragmentation allows to traverse about 10-18 times the number of objects per second than with CDO or Morsa do.

Query: The Grabats contest also provides an example query: find all Java type declarations that contain a static method which has its containing type as return type. Depending on the persistence framework, queries can be implemented in different ways. With XMI and EMFFrag there are no indices that would help to implement the query and we have to traverse the model until we found all type declarations. CDO allows to use SQL to query and Morsa provides a meta-model class to objects index. We measured both: executing the queries with these specific query mechanisms and with the previously mentioned traverse based implementation.

The results are shown in Fig. 10. XMI performs badly for large models. CDO and Morsa with SQL and meta-class index perform best. But even though

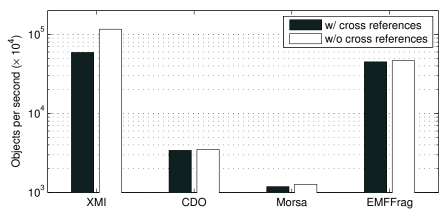

Fig. 7. Number of objects per second that can be created with the different persistence frameworks.

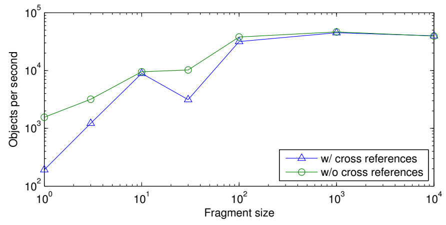

Fig. 8. Number of objects per second that can be created with the EMFFrag and different fragmentation granularity.

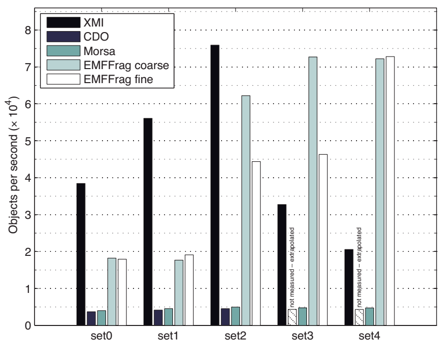

Fig. 9. Number of objects traversed per second during traversing the different Grabats models.

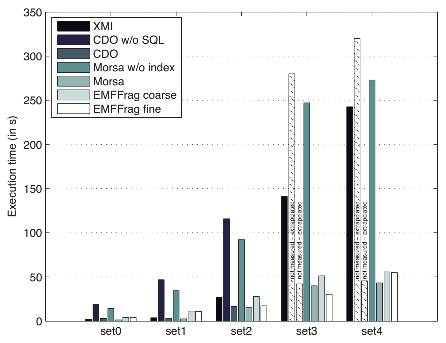

Fig. 10. Execution time for querying the different Grabats models with the example query.

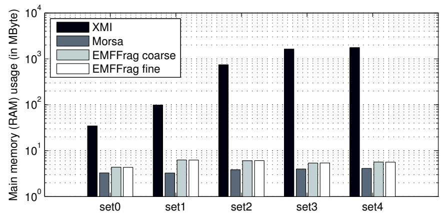

Fig. 11. Memory usage during traversal of the different Grabats models.

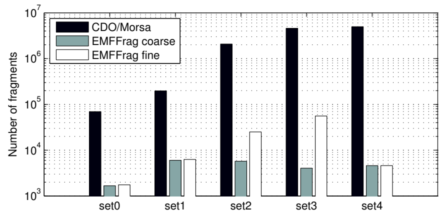

Fig. 12. Number of fragments used by the different persistence frameworks.

EMFFrag needs to traverse the model its performance is similar to CDO and Morsa. For *set3* and fine fragmentation, EMFFrag even outperforms Morsa's index. Remember, with the fine fragmentation, EMFFrag does not need to load any method bodies to execute the query (partial load). Using the traverse implementation, CDO's and Morsa's performance difference to EMFFrag is similar to the measures for model traverse (here we basically perform a partial traverse).

Memory usage: During model traverse, we also measured the memory usage (Fig. 11). XMI's memory usage is proportional to model size, because it needs to load the full models into memory. All other approaches need a comparable constant quantity of memory independent of model size.

7.2 The Influence of Fragmentation on Partial Load Performance

In section 5, we looked at fragmentation analytically and provided a plot (Fig. 5) that describes the expected influence of fragmentation granularity on partial load execution times. Here, we create the same plot, but based on data measured with EMFFrag. For this purpose, we used the same simple meta-model as before (to measure create/modify) and generated models of size 10^6 with different fragment sizes f. We measured the execution times for loading parts of different sizes l. The results are presented in Fig. 13.

The plots show a similar picture with comparable values. Although, the measured times are generally larger due to additional EMFFrag implementation overhead that was not considered in our theoretical examination.

8 Future Work

Sorted and distributed key-value stores: Our fragmentation strategy is based on unsorted key-value store accesses with $\mathcal{O}(log)$ complexity. Neither our analysis, nor out implementation EMFFrag, or our evaluation consider sorted key-value stores that allow to access sequential keys with constant time (scans). Neither did we consider distributed key-value stores which would allow parallel access. Key-value stores are easily distributed in peer-to-peer networks. This is done for two scaleability reasons: replication (allows more users to access the same data in less time) and sharding (distributes data to allow faster and larger storage). Fragmentation can have an influence on both.

Transactions: If multiple user access/modify a model transactions become a necessity. Transaction can either be provided by the underlying data store (e.g.

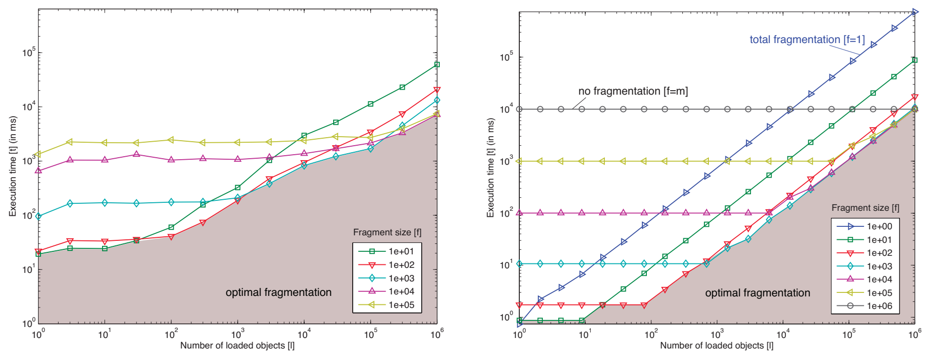

Fig. 13. Execution times for loading model parts with different fragmentation granularity.

with Scalaris [19]) or can be implemented into EMFFrag. On non-distributed data stores, the usual transaction mechanisms can be implemented. More interesting is to explore the influence of fragmentation on transactions (and versioning), because fragmentation granularity also determines the maximum transaction granularity.

Large value sets: In large models, single objects can become very large themselves if they hold large sets of attribute values and references. CDO maps an object's feature values to individual entries in a database table and can manage such objects, but does this slowly. EMFFrag (and Morsa), on the other hand, consider objects as atomic entities and large object can become a performance burden. We need to extend the fragmentation idea to large value sets. Similar to all consideration in this paper, strategies for large value sets have to be optimized and evaluated for the abstract tasks manipulation, iteration (traverse), indexed access (query), and range queries (partial load).

9 Conclusions

Large software models consist of up to 10^9 objects. Models from other application can have a size of up to 10^{12} objects. Traversing models and loading larger aggregates of objects are common tasks (section 2). Depending on fragment size, partially loading models can be done faster than loading whole models or loading models object by object. There is no optimal fragment size, but intermediate fragment sizes provide a good approximation (sections 5 and 7). We provide a persistence framework that allows automatic and transparent fragmentation, if appropriate containment features are marked as fragmentation points in the meta-model (sections 3 and 6). We compared our framework to existing frameworks (EMF's XMI implementation, CDO and Morsa) and our framework performs significantly better for the tasks create/manipulate, traverse, and partial loads. Execution times are 5 to 10 times smaller. Model queries (that favor object-by-object based model persistence with indexes, such as in CDO and Morsa) can be executed with comparable execution times (section 7). All together, fragmentation combines the advantages of both worlds, low memory usage and fast queries like with CDO or Morsa, and traverse and partial load execution times similar to those of XMI.

Model fragmentation also determines the granularity of transactions, which can be a disadvantage. Further problems are single objects with features that can hold large value sets; the fragmentation approach has to be extended for fragmentation of such value sets (section 8). Our framework stores fragments in key-value stores. Those scale easily (both replication and sharding is supported) and integrate well with peer-to-peer computation schemes (e.g. map-reduce). Fragmentation is therefore a good preparation for modeling in the cloud applications (section 4).

References

1. Connected Data Objects (CDO). http://www.eclipse.org/cdo/
2. Grabats 2009, 5th International Workshop on Graph-based Tools: A Reverse Engineering Case Study. http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/ (Jul 2009)
3. Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide Time to Relax. O'Reilly Media, Inc., 1st edn. (2010)
4. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation. OSDI '06, vol. 7, pp. 15–15. USENIX Association, Berkeley, CA, USA (2006)
5. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM 51, 107–113 (Jan 2008)
6. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-Value Store. In: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. pp. 205–220. SOSP '07, ACM, New York, NY, USA (2007)
7. Fowler, M.: Aggregate Oriented Databases. http://martinfowler.com/bliki/ (Jan 2012)
8. Gröger, G., Kolbe, T.H., Czerwinski, A., Nagel, C.: OpenGIS City Geography Markup Language (CityGML) Encoding Standard, Version 1.0.0. Tech. Rep. Doc. No. 08-007r1, OGC, Wayland (MA), USA (2008)
9. Haerder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery. ACM Comput. Surv. 15, 287–317 (December 1983)
10. Hunt, B.: Mongoemf. http://github.com/BryanHunt/mongo-emf/wiki
11. Jézéquel, J.M., Barais, O., Fleurey, F.: Model Driven Language Engineering with Kermeta. In: Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III. pp. 201–221. GTTSE'09, Springer-Verlag, Berlin, Heidelberg (2011)
12. Khetrapal, A., Ganesh, V.: HBase and Hypertable for Large Scale Distributed Storage Systems A Performance evaluation for Open Source BigTable Implementations. Tech. rep., Purdue University (2008)
13. Lakshman, A., Malik, P.: Cassandra: Structured Storage System on a P2P Network. In: Proceedings of the 28th ACM symposium on Principles of distributed computing. pp. 5–5. PODC '09, ACM, New York, NY, USA (2009)
14. Orend, K.: Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace an Object-relational Persistence Layer. Master's thesis, Technische Universität München (2010)
15. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: Morsa: a scalable approach for persisting and accessing large models. In: Proceedings of the 14th international conference on Model driven engineering languages and systems. pp. 77–92. Springer-Verlag (2011)
16. Plugge, E., Hawkins, T., Membrey, P.: The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing. Apress, Berkely, CA, USA, 1st edn. (2010)
17. Scheidgen, M.: EMFFrag – Meta-Model-based Model Fragmentation and Persistence Framework. http://metrikforge.informatik.hu-berlin.de/projects/emffrag (2012)

18. Scheidgen, M., Zubow, A., Sombrutzki, R.: ClickWatch – An Experimentation Framework for Communication Network Test-beds. In: IEEE Wireless Communications and Networking Conference. France (2012)
19. Schütt, T., Schintke, F., Reinefeld, A.: Scalaris: Reliable Transactional P2P Key/Value Store. In: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG. pp. 41–48. ERLANG '08, ACM, New York, NY, USA (2008)
20. Stadler, A.: Making interoperability persistent: A 3D geo database based on CityGML. In: Lee, J., Zlatanova, S. (eds.) Proceedings of the 3rd International Workshop on 3D Geo-Information, pp. 175–192. Springer Verlag, Seoul, Korea (2008)
21. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional, 2nd edn. (2009)
22. Thomas, D.: Programming With Models – Modeling with Code. Journal of Object Technology 5(8) (2006)
23. Zubow, A., Sombrutzki, R.: A Low-cost MIMO Mesh Testbed based on 802.11n. In: IEEE Wireless Communications and Networking Conference. France (2012)