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. Markus: Preliminary 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 either load to many objects or have to access many objects individually. We propose to persist a model broken into fragments. We discuss fragmentation strategies and fragmentation granularities. We further show that fragmentation can be achieved automatically and transparently.
First, we assess the size of large models and describe typical usage patterns to show that most application work with larger model aggregates. We provide an analytical framework to assess execution time gains for partially loading models and different fragmentation granularity. We show that model fragmentation can be achieved automatically and transparently. We compare model fragmentation to existing persistence frameworks (EMF XMI, CDO, and Morsa) based on for common modeling tasks: create/modify, traverse, query, and partial loads.

1 Introduction

Modeling frameworks (e.g. the Eclipse Modelling Framework (EMF) or kermeta) 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). Model persistence frameworks (e.g. Connected Data Objects (CDO)) on the other hand, store models in data bases 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 otherwise allows.

We claim that existing data base 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 for a single execution of one of four abstract modeling tasks. These tasks are (1) creating and extending models, (2) traversing models (e.g. as necessary during model transformation),

(3) querying models, and (4) loading parts of 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 data base. This is time consuming.

Our hypothesis is that modeling tasks can be executed faster, if models are mapped to larger aggregates within an underlying data base. Storing models as aggregates of objects and not as single objects reduces the number of required data base 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"*, [5]. 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 aggregates can be expected to be actually accessed together. Secondly (section 3), we will present our approach to finding aggregates within models. This approach is based on fragmenting models along their containment hierarchy (a fix inherent spanning tree that exist in each model). We will reason, that most modeling tasks need to access aggregates that are 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 for persisting fragmented models. The following section provides a theoretical analysis and upper bound estimation for possible performance gains of 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 abstract modeling tasks. Furthermore, we use our framework to measure the influence of fragmentation on performance to verify the analytical considerations from section 3. We close the paper with conclusions and further work.

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 [19], 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. Therefore, software projects with a large code base probably provide the largest examples for software models. We will first look at the Linux Kernel as an example for a large software project and then extend our estimates on other operating system projects.

Traditionally code size is measured in *lines of code* LOC (physical lines), SLOC (source LOC, like LOC but without empty lines, comments, duplicates), and LLOC (logical LOC, like SLOC but normalized to one statement per line). [20] These measures exist for many known software projects (and their history) and can be easily captured for open source projects.

To estimate the size of code models in number of objects, we additionally need to know how many model objects constitute an average LOC. We used the C Development Toolkit (CDT) to parse the current version of the Linux Kernel (3.2.1) and count the number of abstract syntax tree (AST) nodes required. We assume that model objects and AST nodes are equivalent for our estimation. We also counted the LOCs of the Kernel's code with *sloccount*. This provides an model objects per LOC ratio of 5.99 that we use for all further estimates.

From the GIT versioning system we learned the average numbers of added, removed, and manipulated LOCs per month based on last year's history (4,300 LOC added, 1,800 LOC removed, and 1,500 LOC modified per day). We extrapolate these numbers to estimate the older history of the Kernel. Based on the actual grow in LOC over the last years and our model objects per LOC ratio, we calculated the average number of modified objects in a modified LOC, and can finally estimates the number of all objects in the Kernel's code including its history. This is a model that only contains the changes.

We also transferred all ratios from the Kernel to other OS software projects and publicly reported LOC counts. The results represent rough estimates and are presented in Fig. 1. As you can see, these large software models can have a size upto a magnitude of 10^9 objects.

Furthermore, it is not clear whether further growth in software project size is exponential or linear. Some believe that software size follows Moore's Law. Others think that software is bound by increasing complexity and not by the limitations of underlying hardware.

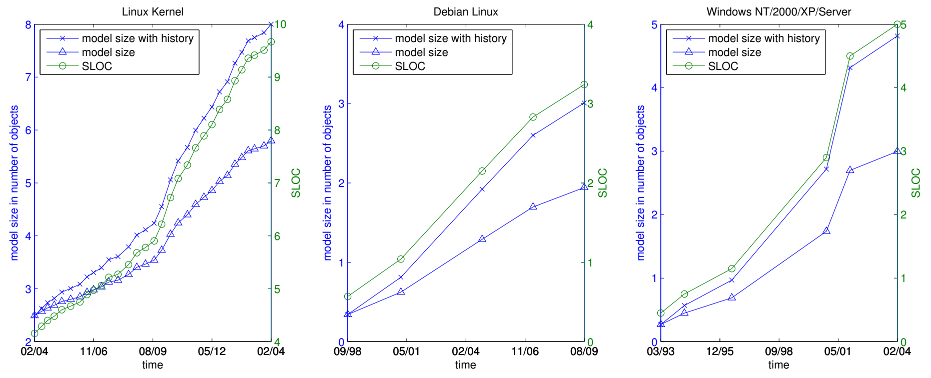

Fig. 1. Rough estimates for software code model sizes based on actual SLOC counts for existing software projects.

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 view individuals at the same time.

2.2 Heterogenous Sensor Data

Sensor data usually comprises of time series of measured physical values in the environment of a sensor; but sensor data can also contain patterns of values (e.g. video images). Sensor data is collected in sensor networks, that combine distributes sensors with an communication infrastructure. Sensor data can be heterogeneous: a sensor network can use different types of sensors that measure a multitude of parameters. [4,11].

Our research group build the *Humboldt Wireless Lab* [21], a 120 node wireless sensor network. Nodes are equipped with 3 axis accelerometers, but more essentially also produce data from monitoring all running software components (mostly networking protocols), and other system parameters (eg.g. CPU, memory, or radio statistics). We represent and analyse this data as EMF based models ([15]).

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 24h. During such an experiment, the network produces a model of $5 * 10^9$ objects.

In general, sensor networks are only bounded by the limitations of the technical infrastructure that supports them. Ambiguous computing and Smart Citys suggest future sensor data models of arbitrary sizes.

Usage Patterns There are two major use-cases: recording sensor data and analysing sensor data. Recording sensor data means to store it faster then 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 Geospatial Models

3D city models are a good example for structured geo-spatial information. The CityGML [6] standard, provides a set of xml-schemas (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) [6].

Model Size As for many cities, a CityGML model is currently established for berlin [17]. 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 roughly $128 * 10^6$ objects.

Estimated on existing CityGML models published for Berlin [17] a LOD 1 building consists of 12.5 objects, a LOD 2 building of 40 objects, and a LOD 3 building of 350 objects. A complete LOD 3-4 model of Berlin would therefore consist of $1 * 10^9$ objects. Berlin inhabits 3.5 million people. About 50% of the worlds $7 * 10^9$ people live in cities. This gives a whole LOD3-4 approximation of 10^{12} objects for a *world 3D city model*.

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 if 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 Fragmention in General

As briefly mentioned before, all models considered in this paper can be characterizes as directed labled graphs with a fix spanning-tree (*containment hierarchy*). In EMF based models, these trees consist of *containment references.* The meta-model determines, which references are containment references and which are simply *cross-references* through the use of *containment reference* or *cross-reference features.*

In the last section, we saw that typical model applications mostly use aggregates of model objects, more precisely sub-trees of the model's containment hierarchy. We propose model fragmentation. The model fragmentation approach, divides (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.* 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 and Fig. 3 present two examples.

3.2 Fragmentation Strategies

Originally a model has no fragmentation; also do we need to maintain a fragmentation when a model is manipulated. Further, we have to assume that fragmentation has an influence on performance. We denote algorithms necessary to create and maintain a fragmentation as *fragmentation strategies.*

There are two trivial strategies: *no fragmentation* and *total fragmentation.* No fragmentation means the whole model constitutes the one and only fragment, such as in regular 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.

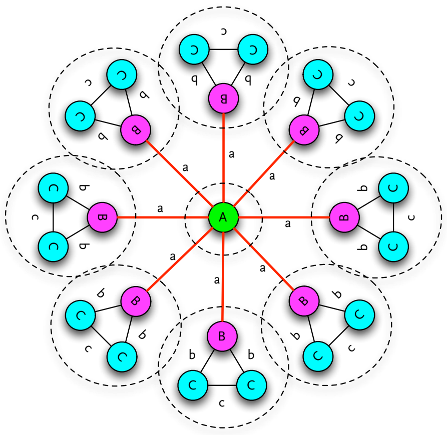

Fig. 2. A fragmented example model with classes A,B,C and reference features a,b,c. Fragments are denotes with circles, references with lines: black for intra- and red for inter-fragment references.

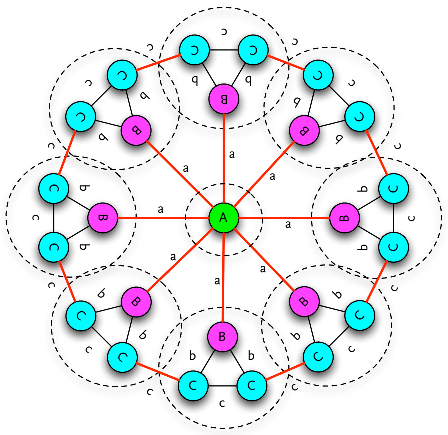

Fig. 3. This example contains inter-fragment cross-references (feature c), i.e. inter-fragment references *by accident*. Feature a produces inter-fragment containment references.

3.3 Meta-Model based Fragmentation

There are other fragmentation strategies that produce fragmentations between no and total fragmentation. In this paper, we propose and use *meta-model based fragmentation* as a new (but very simple) strategy.

A meta-model defines possible models by means of classes and associations (i.e. structural features in EMF). A good meta-model only allows models that are suitable for the envisioned application. Containment references (indirectly the containment hierarchy) are already used by the meta-modeller to aggregate[1] closely related objects. For the meta-model based fragmentation strategy, we ask the meta-modeller to mark some containment reference features as *inter-fragment reference features*. All instances of these features, will break the containment hierarchy into fragments (we call their instances *inter-fragment* references). This way, the meta-model determines where the spanning-tree is broken into fragments. Once inter-fragment reference features are designated within the meta-model, it is easy to create and maintain fragmentations automatically and transparently; ref. to section 6.

Only containment features can be designated as inter-fragment reference features and all instances of such a reference feature will be inter-fragment references (like a-references in Fig. 2). Other references (i.e. cross references) can become inter-fragment references *by accident* (like c-references in Fig. 3).

[1] In UML composition is a special case of aggregation

4 Related Work

4.1 Model Persistence

For EMF based models, there are at least four different approaches to large models: (1) regular EMF with XMI based persistence; (2) EMF resources, where a resource can be a file or an entry in a data base; (3) CDO [1] and other object relational mappings (ORM) for Ecore; (4) morsa [13] a EMF data-base mapping for non-relational data bases. [2]

First, regular EMF: Models are persisted as XMI documents and can only be used if loaded completely into a computer's RAM. EMF realises the *no fragmentation* strategy. The memory usage of EMF is linear to the model's size. It's time efficiency is good for small models, since it needs no further managing overhead, but is bad for large models due to pointless garbage collection and extensive allocation of memory.

Secondly, EMF resources [18]: EMF allows to fragment a model into different resources. Originally, each resource could only contain a separate containment hierarchy and only 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 the first object of a resource is accessed. Model objects have to be assigned to resources manually (*manual fragmentation*). To actually save memory the user has to unload resources manually too. Memory efficiency is good if resources are handled properly. Time efficiency is also good due to minimal management overhead. The framework MongoEMF [8] maps resources to entries in a MongoDB [14] data base.

Thirdly, CDO [1]: CDO is a ORM for EMF. [3] It supports several relational data bases. Classes and features are mapped to tables and their columns. Objects, references, and attributes are mapped to rows. CDO was designed for software modeling and versioning and provides transaction, views, and versions. These features produce some overhead. Relational data bases and SQL provide mechanisms to index and access objects and their features. This allows fast queries, if the user understands the underlying ORM. But, complex table structures and indexes cause very low entry (i.e. object) creation rates. Sorting entries into indexes and tables consumes time. CDO is RAM efficient. Traversing or loading parts of models is slow, because each object needs to be accessed individually.

Fourthly, morsa [13]: Different to CDO, morsa uses so mongoDB [14], a no-sql data base. In no-sql data bases (or other key-value stores) store arbitrary values in an key-value map. This simple data structure allows fast and easy distributable data bases. Morsa stores objects, their references and attributes

[2] Mentioned memory and time efficiency in this section are taken from the evaluation section 7.

[3] Lately, CDO also support non-relational data bases, such as MongoDB [14]. Such features were not evaluated in this paper; but one can assume characteristics similar to those of morsa.

as JSON documents. Morsa furthermore uses mongoDB's index feature to index specific characteristics (e.g. an objects meta-class reference). This produces similar characteristics than with CDO: fast queries, slow creation, traverse, and load.

4.2 Key-Value Stores

The general term *data base* has long been used as a synonym for relational data bases. But in the last decade a new kind of data base has become more and more popular. Web and cloud computing require scalability above all, and traditional ACID [7] properties can be sacrificed if the data store is easily distributable. This explains the popularity of so called *No-SQL* data bases or *key-value store*. Such key-value stores only provide a simple map data structure: there are only keys and values. For more information and an comparison of existing key-value stores refer to [12].

Key-value stores are ideal data bases for persisting fragmented models. Fragments can be identified by URIs, which are easily mapped to key-value store keys, and XMI (as native model persistence format) can be used for values. Key-value stores provide a scalable persistent solution with good performance characteristics. This allows to react to increasing requirements for model sizes and number of parallel access (e.g. for the geo-spatial models modelling application).

There are three different applications that inspired three groups of key-value stores. First, there are web applications and the popular MongoDB [14] and CouchDB [2] data bases. These use JSON documents as values and provide additional indexing for JSON attributes.

Secondly, there is cloud computing and commercial Google Big-Table [**?**] and Amazon's Dynamo [3] inspired data stores. HBase [9] and Cassandra [10] are respective open source implementations. Those data bases strive for massive distribution, they provide no support for additional value structuring, but integrate well into map-reduce [**?**] execution frameworks, such as Hadoop (HBase is Hadoop's native data store).

A third application is high performance computing. Scalaris [16] is a key-value store opt for massive parallel, cluster, and grid computing. Scalaris provides mechanisms for consistency and transactions and brings some ACID to key-value stores.

For the implementations in this paper, we use HBase, which provides everything we need and nothing more.

5 Possible Performance Gains from Model Fragmentation

In this section, we analyse the theoretical 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. 5).

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 realizations for the functions $parse$ 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. 4. For $access$ we measured the execution time for accessing keys in HBase for data base 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 fit (Fig. 5). For details on the measuring environment refer to section 7.

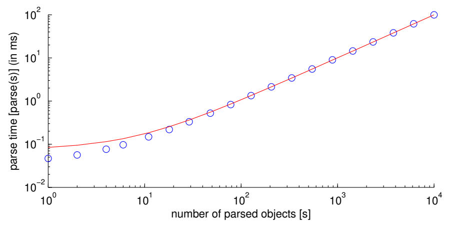

Fig. 4. Measurements for EMF's XMI parser (dots) and linear fit (line).

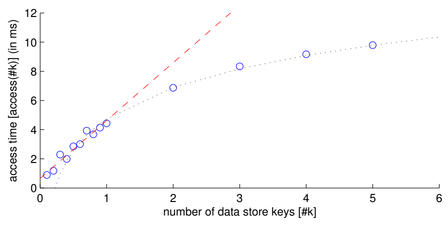

Fig. 5. HBase accesses measures (dots) and fits (linear dashed, log. dotted).

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

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 $f \in \{10^i | 0 \leq i \leq 6\}$. 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 example, all partial loads take at most three times the optimal time. 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.

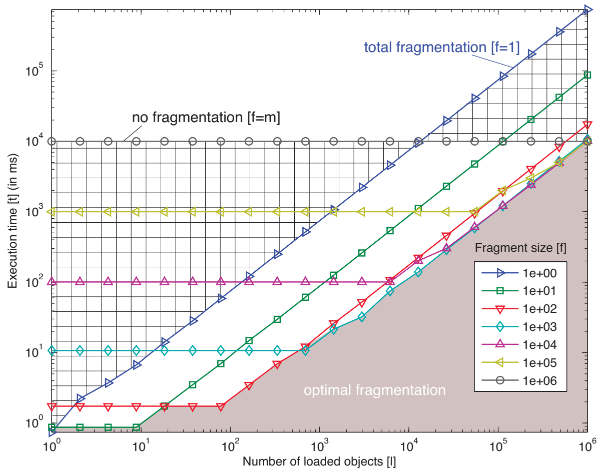

Fig. 6. Computed execution times for partial loads from a model with 10^6 objects and fragmentations of different granularity.

Fig. 7 shows the same data as contour plot. The largest partial models can be loading if the loaded parts have the same size as the fragments $l = f$. Fig. 8 shows times for a larger model with $m = 10^9$. Intermediate fragment sizes are still good fragment sizes. Such large models produce large number of data base keys for total fragmentation, but the parsing overhead $parse(0) > 0$ has obviously an larger influence than growing access times. Thus the plot for total fragmentation is still a line, and for large number of loaded objects, the plot for all fragment sizes are parallel.

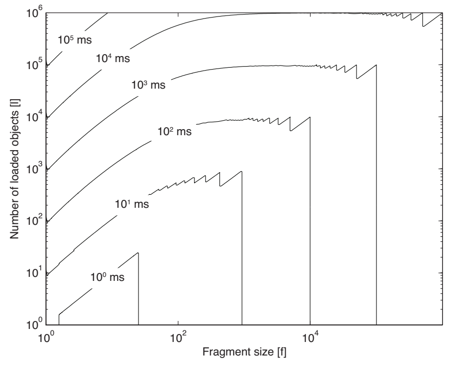

Fig. 7. Contour plot. Shows which (f, l) have the same costs t.

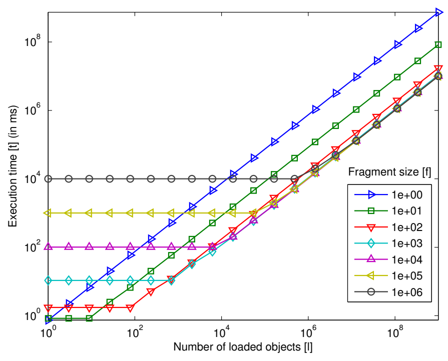

Fig. 8. Computed partial load times for a model of size 10^9.

6 Implementation of Model Fragmentation

In previous sections, we introduced model fragmentation, a specific fragmentation strategy, and the theoretical merits of fragmentation. In this section, we present a framework that realizes model fragmentation and implements the meta-model based fragmentation strategy. The framework is called *EMFFrag*.

The rational behind EMFFrag is to (re)use EMF resource as much as possible. EMF resource already realize partial model persistence, they manage inter-resource references through proxies, they lazy-load, can be added and removed, objects can be moved between resources, etc. EMFFrag only uses and specialized the existing implementations of EMF resources.

Fig. 9 illustrates EMFFrag's architecture and operation.

In EMFFrag a fragmentation is realized as a resource set, and fragments as resources. Different to EMF resources, fragmentations and fragments are completely hidden from the framework user (transparency). The fragmented model is internally realized though dynamic (meta-model independent) EMF objects. These internal objects are also hidden from users. Users access the model through generated (i.e. meta-model based interface) and stateless delegates. The delegates delegate all feature accesses to an EStore implementation, which delegates all calls to the corresponding internal object. The store translates between delegates and internal objects: parameters are unwrapped from delegates to internal objects, and return values are wrapped from internal objects to delegates.

If a delegate was created by the user, the initial internal object is created, the first time the user gives the delegate to the store. Delegates are cached by fragments through Java weak references. If the store has to deliver an internal object to the user as delegate, the delegate is either cached in the fragment or created and cached.

This rather complex internal object and delegate architecture allows to recognize fragments that are no longer used. Once the weak reference cache of an fragment has lost all reference to cached delegated, it can be assumed that the

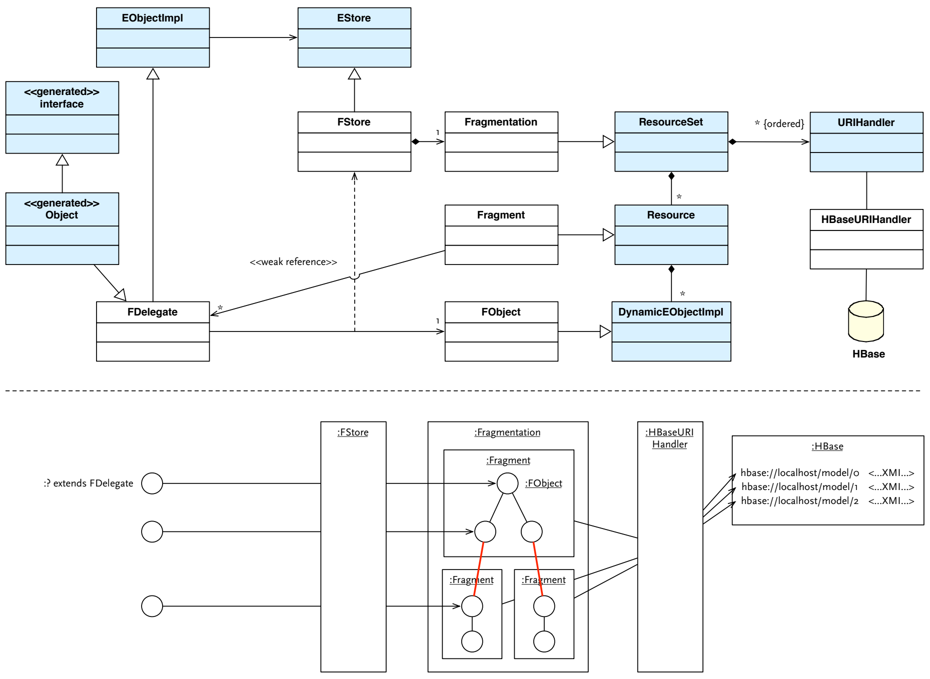

Fig. 9. EMFFrag architecture and example fragmentation.

user lost all references to all fragments. The fragment can be safely unloaded without destroying remaining delegates.

The store recognizes when an object is added to (or removed from) a feature that is marked as inter-fragment reference feature in the meta-model. In this cases, EMFFrag creates a new fragment (or deletes a fragment) and moves the object into the new fragment. EMF's containment proxies handle everything else automatically.

All fragments persist themselves through the regular URIConverter and URIHandler API. EMFFrag registers a specific URIHandler that maps fragment URIs to entries in an HBase key-value store. Fragments are lazy loaded through the regular EMF implementations. When a fragment is unloaded, it is saved (if changed). We extended the regular EMF unload behavior: EMFFrag does not only proxyfy all objects, but also removes all intra-fragment references. This allows the Java gargabe collecter to completely remove unloaded fragments.

Further fragment caching allows maintain unloaded caches on a LRU-basis for potential re-use.

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 Computer with Intel Core i5 2.4ĠHz 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 [**?**].

7.1 Fragmentation Compared to other Persistence Frameworks

To compare fragmentation to EMF XMI, CDO, and Morsa, we measured execution time for the abstract modeling task. To analyse traverse and query, we used example models from the Grabats 2009 contest [**?**] as benchmarks. Those were already used to compare Morsa with EMF XMI and CDO here [13]. There are five example models labelled *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 growths is not linear. To measure the last task create/modify, we use simple test models, because importing Grabats' large XMI models files does not allow effective separation of loading the models with EMF and storing them with the respective persistence framework. We don't provide any comparative measures for partial loads. Partial load performance is indirectly covered by the measures on queries, which required to partially load the Grabats models. Furthermore, partial loads are extensively measured for EMFFrag in the next section.

Fig. 13 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 EMF XMI there is always only one fragment, because it implements no fragmentation. For EMF-Frag, we provided two different meta-model based fragmentations. The first one puts each Java compilation unit and class file into a different fragment (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 obviously contain a lot of method definitions.

We could not measure CDO's performance for *set3* and *set4*: the models are to large for a single CDO transaction, and circular cross-references do not allow to import the model with different transactions.

Create/Modify Markus: Big TODO: I need to measure this, describe the test models, plot results and testmeta-model and describe the results.

Markus: Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Markus: Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed

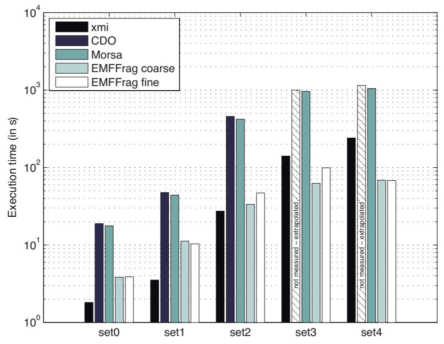

Fig. 10. Execution time for traversing the different Grabats models with the different persistence solutions.

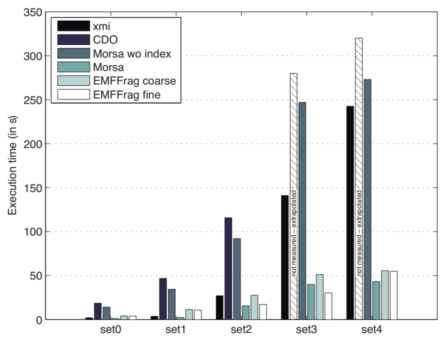

Fig. 11. Execution time for querying the different Grabats models with the example query.

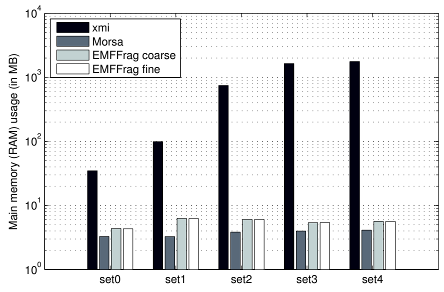

Fig. 12. Memory usage time for traversing the different Grabats.

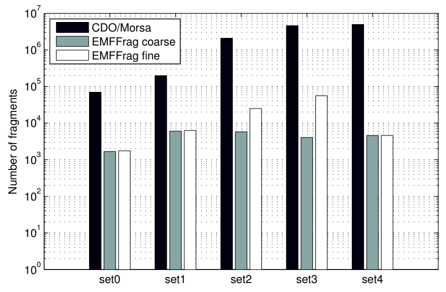

Fig. 13. Number of fragments used by the different persistence solutions.

diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Traverse The execution times of CDO, Morsa, and EMFFrag are proportional to the size of the model (Fig. 10). Times for EMF XMI grow faster than the model's sizes. Interestingly, Morsa and CDO both use total fragmentation and their respective execution times are very close. EMFFrag performs significantly faster: upto 10 times as fast as Morse and CDO. For models *set2* and *set3* (where the number of fragments differ largely between fine and coarse fragmentation) the fine fragmentation performs (not surprisingly) worse, since more fragments have to be loaded.

Query The Grabats contest also provides an example query: find all Java type declarations that contain a static method which has this declared type as return type. Depending on the persistence framework, queries can be implemented in different ways. With EMF XMI and EMFFrag there are not indexes 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 (Fig. 11: EMF XMI performs badly for large models. CDO and Morsa with SQL and meta-class index performs best. But even though EMF-Frag 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. 12). EMF/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 a simple meta-model (ref. to Fig. **??**) 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. 14.

The plots show a similar picture with comparable values. Although, the measured times are generally larger probably 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 two peer networks. This is done for two scalability reasons: replication (allows more users to access the

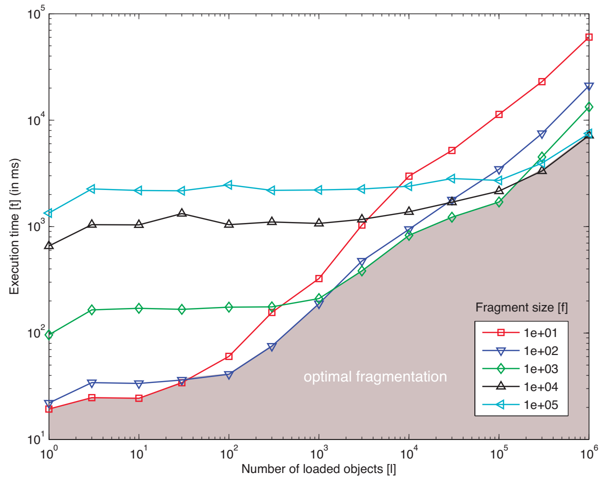

Fig. 14. Execution times for loading model parts with different fragmentation granularities.

same data in less time) and sharding (distributing data to allow faster and larger storage). Fragmentation can have an influence on both.

Transactions If multiple user access/modify transactions become a necessity. Transaction can either be provided by the underlying data store (e.g. with Scalaris [16]) or can be implemented into EMFFrag. On non-distributed data stores, a common transaction pattern could be used. More interesting is explore the influence of fragmentation on transactions (and versioning), because the maximum granularity is determined by fragment size.

Cross-References EMFFrag persists containment references with URIs with a fragment ID pointing to the fragment that contains the reference objects. This does not work for cross references: when an object is moved, its URI changes and all referencing object use invalid URIs. For this reason EMFFrag needs to use a secondary index that maps object IDs to containing fragments. EMFFrag uses URIs with object IDs to persist cross references. To resolve such an URI, the secondary index is used to find the fragment that contains the object. Object IDs and secondary index are only maintained for objects that are actually cross references to keep the index small.

In our analysis and evaluation, we did not examine the influence of cross references and corresponding indexes on performance. We have to expect that create/modify tasks are performed considerably slower, since two indexes have to be maintained. The impact on traverse, query, or partial loads, on the other hand, should be minimal.

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 so slowly. EMFFrag (and Morsa), on the other hand, consider objects as atomic entities and large object become a performance burden too. We need to extend the fragmentation idea to large value sets. Similar to all consideration in this paper, these 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 upto 10^9 objects. Models from other application can have a size upto 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 XMI, CDO and Morsa) and our framework performs significantly better for creating/manipulating, traversing, and partially loading models. Execution times are 5 to 10 times smaller. Model queries (that favour object-by-object based model persistence with indexes, such as in CDO and Morsa) can be executed with comparable execution times (section 7). 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 **??**). Our framework stores fragments in key-value stores. Those scale easily (both replication and sharding is supported) and integrate well in peer-to-peer computation scheme (e.g. map-reduce). Fragmentation therefore prepares very large models for modelling in the cloud applications (section 4).

References

1. Connected data objects (cdo). http://www.eclipse.org/cdo/
2. Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide Time to Relax. O'Reilly Media, Inc., 1st edn. (2010)
3. 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. SIGOPS Oper. Syst. Rev. 41, 205–220 (Oct 2007)
4. Estrin, D., Girod, L., Pottie, G., Srivastava, M.: Instrumenting the world with wireless sensor networks. In: IEEE International Conference on Acoustics, Speech, and Signal Processing. Salt Lake City, UT , USA (2001)

5. Fowler, M.: Aggregate oriented databases. http://martinfowler.com/bliki/AggregateOrientedDatabase.html (Jan 2012)
6. 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)
7. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. 15, 287–317 (December 1983)
8. Hunt, B.: Mongoemf. http://github.com/BryanHunt/mongo-emf/wiki
9. 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)
10. 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)
11. Lynch, J.P.: A summary review of wireless sensors and sensor networks for structurr al health monitoring. The Shock and Vibration Digest 38(2), 91–128 (2006)
12. 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)
13. 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. MODELS'11, Springer-Verlag, Berlin, Heidelberg (2011)
14. 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)
15. Scheidgen, M., Zubow, A., Sombrutzki, R.: ClickWatch – An Experimentation Framework for Communication Network Test-beds. In: IEEE Wireless Communications and Networking Conference. France (2012)
16. 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)
17. 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, Seoul, Korea, pp. 175–192. Springer Verlag, Berlin, Heidelberg, New York (2008)
18. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional, 2nd edn. (2009)
19. Thomas, D.: Programming with models – modeling with code. Journal of Object Technology 5(8) (2006)
20. Wheeler, D.A.: Counting Source Lines of Code (SLOC). http://www.dwheeler.com/sloc (2002)
21. Zubow, A., Sombrutzki, R.: A low-cost mimo mesh testbed based on 802.11n. In: IEEE Wireless Communications and Networking Conference. France (2012)