Model Fragmentation For High Access Performance of Models Persisted in Key-Value Stores

Markus Scheidgen

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

Abstract. Markus: TODO

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 citemodelsAsCode, 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). [19] 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.

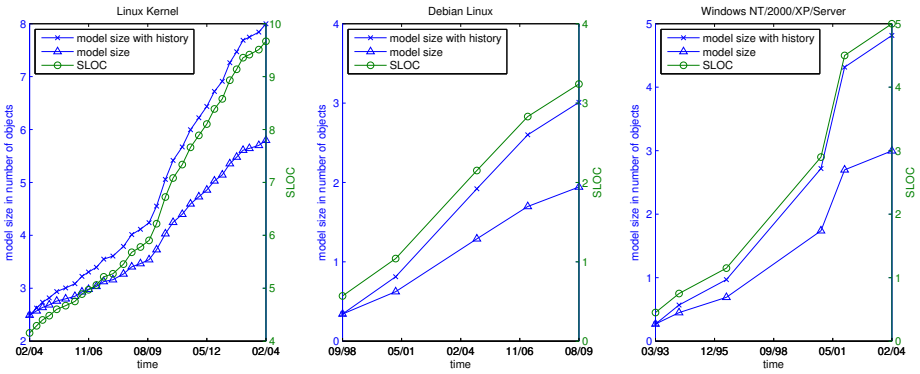

Fig. 1. Rough estimates for software code model sizes based on actual SLOC counts for existing software projects.

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.

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* [20], 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[1]

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

[1] Plus and minus signs characterise the importance of time efficiency for the corresponding property or task from 0 to 3.

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.

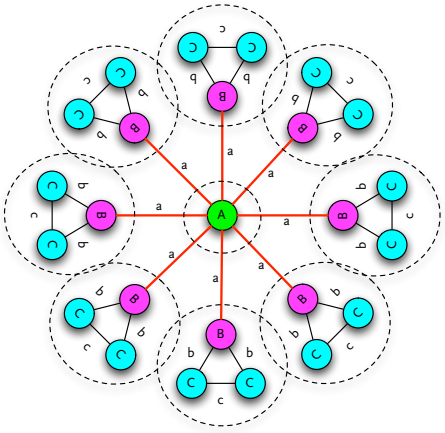

Fig. 2. A fragmented example model with classes A,B,C and reference features a,b,c. Fragments are denotes with circles, references with lines: black for intra- and red for inter-fragment references.

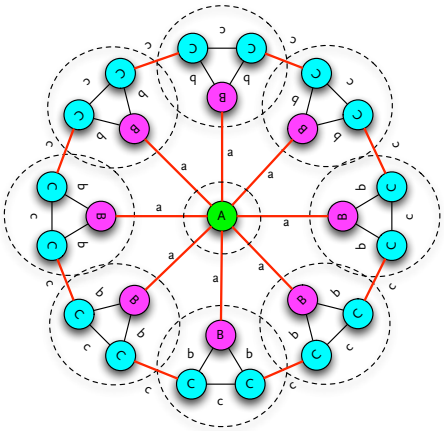

Fig. 3. This example contains inter-fragment cross-references (feature c), i.e. inter-fragment references *by accident*. Feature a produces inter-fragment containment references.

3.2 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.

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[2] 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).

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. [3]

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

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

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. [4] 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 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

[4] 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.

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 performance gains from optimal model fragmentation. Even though these gains will never be achievable for real applications (in most cases), the results in this section provide a theoretical lower bound. The results also help to benchmark (existing) fragmentation strategies.

In this section, we will only analyse the time efficiency for partially loading a fragmented model. Partially loading a model is needed for the modelling traversing, querying, and loading parts of a model.

First, we need to define a few basic operations and performance functions for these operations. The operations needed to load a fragmented model are accessing a model fragment and parsing the fragment (this includes reading the persistent version of the fragment).

We define the *parse* function that determines how long it takes to parse a serialized model fragment of a given *size* as a linear function: $parse(size) = \mathcal{O}(size)$. The next function *access* determines how long it takes to find an entry

in a key-value data store based on the number of key $\#keys$, i.e. number of entries: $access(\#keys) = \mathcal{O}log(\#keys)$. Note that all data stores (key-value, files-systems, relational data bases) perform logarithmically at best Markus: cite?.

To replace the $\mathcal{O}$-notation with actual function parameters, we measured the performance of EMF parsing (depending on model size) and HBase data store access (depending on number of data store entries). The results are shown in Fig. 5. As expected the parsing performance is linear and the data store access behaves logarithmic.

We can estimate the execution times for loading a part of a fragmented model for different fragment sizes. We will use the following parameters: The total model $size$, the average number of objects per fragment $fsize$, and the size of the model part that is loaded $lsize$. We assume *optimal fragmentation*. In an optimal fragmentation (with respect to loading a model part), the number of objects that are actually loaded is minimal. We only need to load as many fragments as are needed to accumulate the size of the loaded model part. The time t to load from an optimally fragmented model is:

$$t = \frac{lsize}{fsize}\left(access(\frac{size}{fsize}) + parse(fsize)\right)$$

With functions $parse$ and $access$ as determined by our measurements, we get the times presented in Fig. 5.The contour plot shows the relation between fragment size, load size, and the time it takes to load. The contours show loads that take the same time. The linear or logarithmic parts of the contours are more or less dominant: if parsing is relatively slow, fragmentation becomes more important (linear parts of contours are longer), if accessing the data-base becomes relatively slow, fragmentation becomes less relevant and generally large fragment sizes are more desirable (logarithmic parts of contours are longer).

From this plot, we can see what fragmentation allows compared to no fragmentation ($fsize = size$) or complete fragmentation ($fsize = 1$). No fragmentation always takes the full time. Of course optimal results can be obtained if ($lsize = fsize$). This optimal result allows to load models a 1000 times bigger at the same time than total fragmentation.

In this analysis, we only considered unsorted, non distributed key-value stores. Sorted key-value stores further allow to organise fragments in the order that they are most often loaded in. This reduces the number of necessary logarithmic accesses and replaces them with time constant accesses (scans). Distributed key-value stores allow to access and read fragments in parallel, thus allowing shorter execution times. Markus: Refer to future work?

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 metamodel based fragmentation strategy. The framework is called *EMFFrag*.

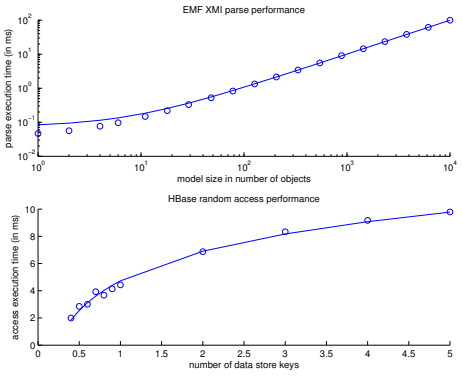

Fig. 4. Measure for liner parsing performance of EMF and logarithmic access performance of HBase.

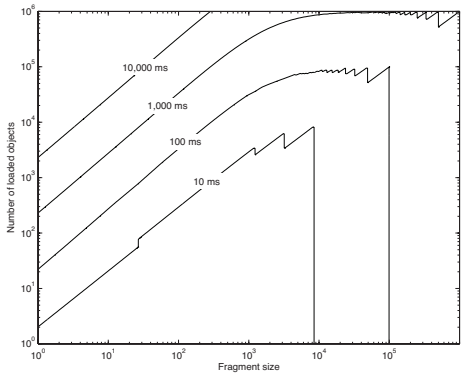

Fig. 5. Load times based on actual EMF parsing and HBase access measurements.

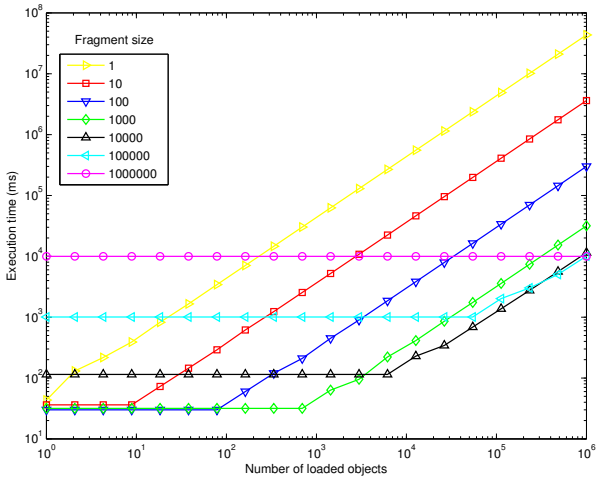

Fig. 6. Load times based on actual EMF parsing and ...

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. 7 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.

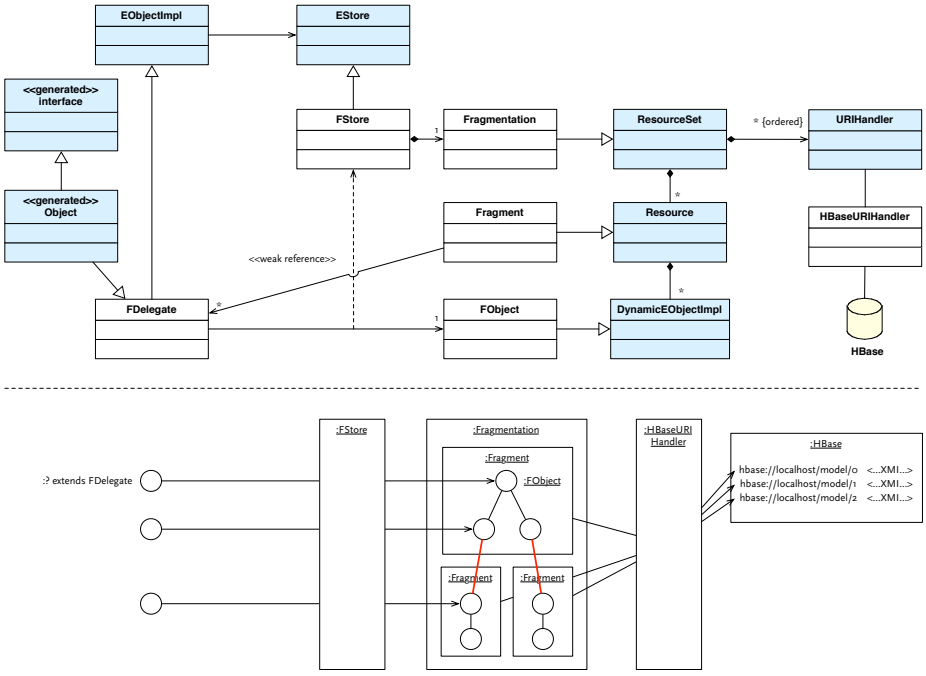

Fig. 7. EMFFrag architecture and example fragmentation.

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 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

We provide and evalutation based on two different models, following two different goals. First, we want to compare EMFFrag to its related world from the software modeling world using a software models. Secondly, we want to show the influence of different granularity of fragmentation on different performance parameters.

All experiments were conducted on a Markus: TODO. All measures were repeated at least 20 times, and all present results are respective averages.

7.1 Grabats – Actual Software Models

In this section, we will evaluate our approach to EMF/XMI, CDO, and Morsa. To cover as many use cases as possible, we will look at four abstract parameters: execution time of creating/manipulating, accessing complete models, accessing specific parts of models (query), and memory usage.

The evaluation models are taken from the Grabats 2009 contestMarkus: reference and some words. There are five different software models. All covering Java projects. The models include the project structure, type structure, down to AST models of Java code. The five models have different sizes and complexities. Fig. 8 shows the number of fragments that the different approaches used. Where EMF/XMI always uses only one fragment and morsa one fragment per object (i.e. the number of fragment equals the size of the model in number of objects). For EMFFrag, we used two differently strong fragmentations. First (Frag1), each class and compilation unit is put into a single fragment, and secondly (Frag2), additionally each method block is put in its own fragment. This shows the internal structure of sets zero through four: sets zero and one are pretty small, sets two and three containing lots of methods and its implementations, where set four seems to only contain information about projects, types, and members, but no implementation of members.

Model creation time

Memory usage We measured the memory usage, needed for a full model access; the results are shown in Fig. 9. EMF/XMI's memory usage is proportional

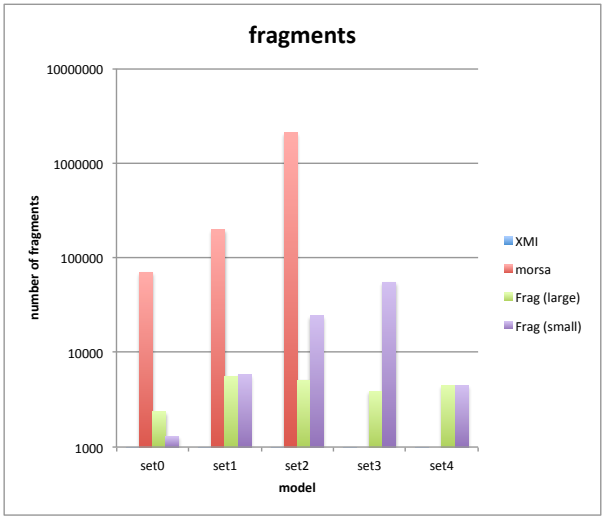

Fig. 8. Number of fragments used

to model size, because it needs to load full models into memory. All other approaches need constant quantity of memory independent of model size. This quantity varies between the approaches, but is of an practical amount for all approaches.

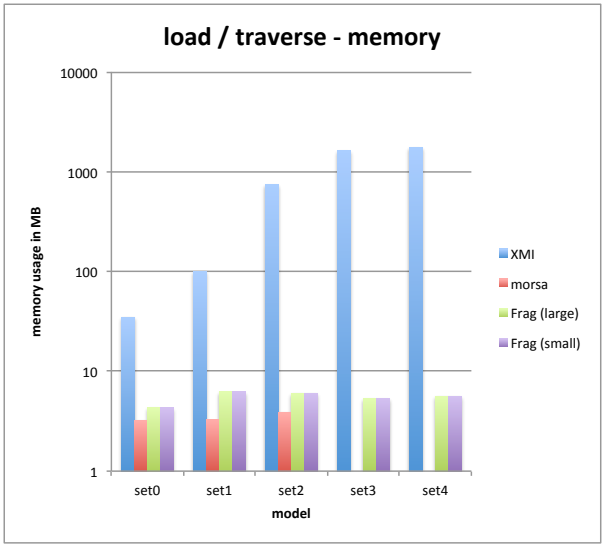

Fig. 9. Memory usage for a full model traversal

Full model access The execution times for accessing the complete models are shown in Fig. 10. Fist of all, because the whole model is accessed, the execution time for all approaches grows with model size. EMF/XMI execution time grows more then linear due to pointless garbage collection. Morsa also shows sub linear performance, since each object has to be accessed individual, and even the best index only performs logarithmically depending on model size. EMFFrag performance is proportional to model size and number of fragments; as expected it takes longer to access the whole model on a more fragmented store.

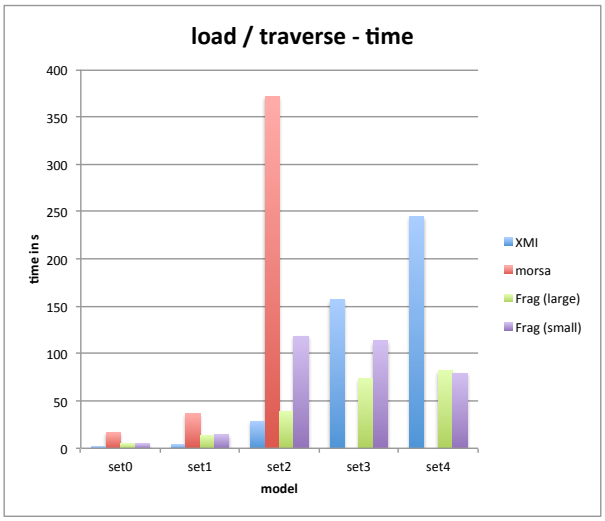

Fig. 10. Execution time for a full model traversal

Selective model access In this section, we present the execution times of model queries. The query is also taken from the grabats 2009 contest. The query is to find those *TypeDeclaration* instances in the Java model that contain a static method with the containing type as return type. Morsa can profit from its index on the objects meta-class to find all instances of *TypeDeclaration*. All other approaches have to traverse the model to find all instances of *TypeDeclations*. For the Morsa approach, we used two implementations of the query: one using the meta-class index, and the other traversing the model as in all other approaches. The results are shown in Fig. 11.

Markus: TODO explain the query

7.2 The Influence of Fragmentation on Performance

In this section, we look hat query and full access execution times for models of the same size and characteristics, but with different fragmentations. For this

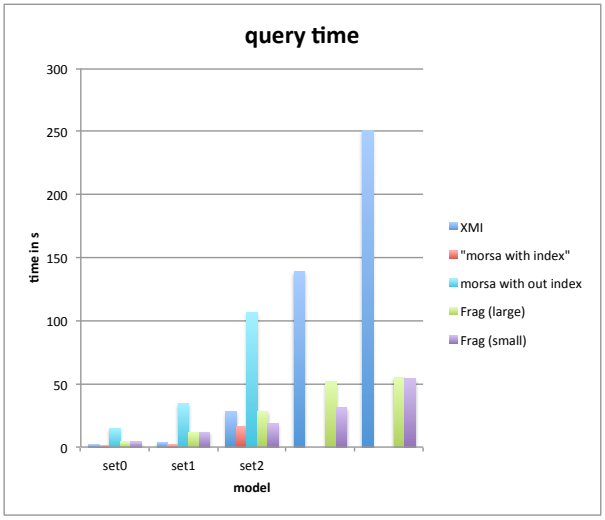

Fig. 11. Execution time for a specific query

purpose, we use a simple meta-model (ref. to Fig. **??**). We generated models with a simple algorithm that takes model size, fragment size, and a so called *depth factor* as parameters. The *depth factor* determines the proportion of average number of object children to the overall model size, i.e. the depth of the model.

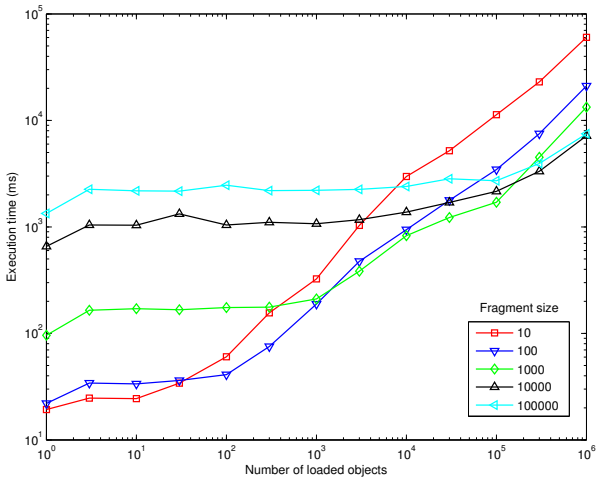

Fig. 12. Execution times for traversing parts of models with different fragmentations.

Markus: Measure first and then describe

8 Conclusions

- We proposed model fragmentation for model data base persistence with better performance characteristics that existing solutions (e.g. CDO, Morsa).
- For most modelling tasks (create, traverse, query, partial load) fragmented models perform better or similar to existing solutions. For create, traverse, and partial load performance enhancements are of magnitudes.
- Fragmentation can be achieved with meta-model based fragmentations, which already provides close to optimal performance for many applications and tasks (e.g. editing and compiling for software code models). Other strategies are imaginable and future work.
- The downside is that the underlying key-value stores (generally) do not provide a transaction mechanism. Future work: implement transaction within EMFFrag or use key-value stores with transactions (e.g. Scalaris).
- On the up-side, good scalarbility for both model size and demand for parallel access: e.g. in geo-spatial models. Good integration into parallel computing, i.e. with map-reduce frameworks.

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. Wheeler, D.A.: Counting Source Lines of Code (SLOC). http://www.dwheeler.com/sloc (2002)
20. Zubow, A., Sombrutzki, R.: A low-cost mimo mesh testbed based on 802.11n. In: IEEE Wireless Communications and Networking Conference. France (2012)