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. EMF or kermeta) can only work with models if they are loaded into a computer's RAM. This limits the possible size of a model. Modeling frameworks provide only limited capabilities to deal with large models (i.e. lazy loading in EMF). Other frameworks (e.g. CDO) persist models in data bases and load and unload single model objects on demand. This allows to work with models of arbitrary size. Data base persistence is RAM efficient and practically solves today's model size issues, but we claim that existing approaches are not time efficient. Here, time efficiency relates to the execution time for common modeling tasks. In this paper, we look at the following tasks (1) creating and extending models, (2) traversing models (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 many modeling tasks (especially traversing models and loading parts of models) require to load large chunks of the model. Existing data base 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.

Our hypothesis is that modeling tasks can be executed faster, if models are represented as larger aggregates within an underlying data base. Storing models as aggregates of objects and not as single objects reduces the number of 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"*, []. This already raises the three main questions that we want to answer with this paper: Are there aggregates that are *often* accessed together? How can we create them 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 **??**), we look at typical modeling applications: which model sizes they work with and what concrete modeling tasks are used 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 **??**), we will present our aggregation approach. This approach is based on fragmenting models along their inherent spanning tree (composite tree). We will reason, that most modeling tasks need to load sub-trees of this composite tree, and that such sub-trees (model fragments) make reasonable aggregates. In the related work section 4, we present existing model persistence frameworks and interpret their strategies with respect to the idea of fragmentation. In the next section, we provide a theoretical analysis of performance under optimal fragmentation to provide some bounds for efficiency expectations. In section **??**, we finally present our fragmentation strategy and a framework that implements it. The next section is the evaluation section, where we compare our framework to existing frameworks with respect to time and memory efficient execution of common modeling tasks. Furthermore, we use our framework to analyse the influence of the amount of fragmentation on performance. We close the paper with conclusions and further work.

2 Applications for Large Models

In this section, we look at three modeling applications. The first one is software modeling, which is what most of the MODELS community is concerned with and what the modeling frameworks mentioned in this paper were designed for. But techniques originally developed for software modeling are also used for other applications. More abstractly, software modeling can be explained as the representation, analysis, and manipulation of structured data.

Our research group, for example, uses EMF to represent and analyse sensor data received from our wireless sensor network test-bed (*Humboldt Wireless Lab* [**?**]). Models of heterogeneous sensor data will be our second application. This application is what actually motivated us as authors for this work, because existing approaches where either impractical (lazy loading of EMF resources), or far to slow to store the data at the rate it was produced (CDO and manual relational data base persistence). Refer to our publications [**?**,**?**] for more details.

The third application that we want to analyse, are geo-spatial models and more specifically 3D object models of cities. Such models represent the features of a city (boroughs, streets, buildings, floors, rooms, windows, etc.) as a containment hierarchy of objects and their properties. These models are closely related to sensor data analysis.

For all three applications, we estimate the size of typical models. Furthermore, we determine the most important use cases for all three applications. From these use cases we derive the commonly used modeling tasks and their characteristics.

2.1 Software Models

In modern Model Driven Software Development (MDSD) all artifacts including software code are understood as models Markus: cite needed. IDEs (e.g. eclipse's

JDT or CDT) already represent code as models (ASTs). Advances in modeling frameworks (e.g. model comparison) suggests that code is also persisted as model.

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; refer to Wheeler [**?**], and LLOC (logical LOC, like SLOC but normalized to one statement per line). 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 CDT to parse the current version of the Linux Kernel (Markus: version) and counted the number of AST nodes required. We assume that model objects and AST nodes are equivalent for our estimation. We also counted the LOCs of the Kernal Code with Markus: tool name. This provides an model objects per LOC ratio 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. We extrapolate these numbers for the further history of the Kernel. Based on the actual LOC grows over the last years and our model objects per LOC ratio, we calculated the average number of modified objects in a modified line of code, and can finally estimates the number of all objects in the Kernel's code and its history (a model that only contains the changes).

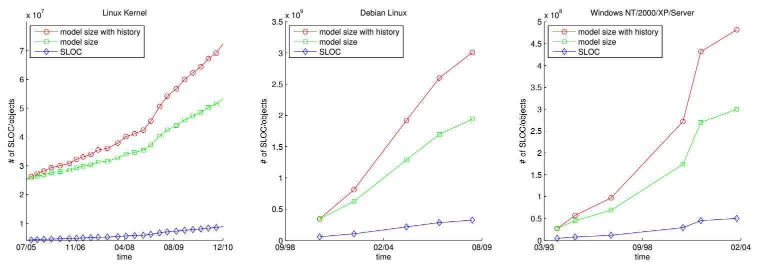

Fig. 1. SLOC and estimated number of objects for popular large software projects.

We also transferred all ratios from the Kernel to other OS software projects and publicly reported LOC counts. The results 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 grow 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.

Modeling Tasks There are two major use cases in today's software development: editing and analysing/transforming/compiling. The first is done based on diagrams (graphical editing) or 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 whole containment tree of the software model. Analysing, transformation, and compiling is usually either done for the whole model or again on a per package or compilation unit basis. Within these aggregates, the use-case usually requires to traverse the model. An exception can be the analysis based on single queries. But due to performance issues, model analysis is usually performed by traversing the model and executing queries more like 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. Markus: cites, cites, cites.

We build the *Humboldt Wireless Lab* are 120 node wireless sensor network. Nodes are equipped with 3 axis accelerometer, but more essentially for our research also creates data from monitoring all running software components (mostly networking protocols), and other system parameters (CPU, memory, or radio activity, etc.). We represent analyse this data as EMF ecore based models (e.g. [**?**]).

Model Size HWL network protocol and system software components provide 372 distinct data sets (containing data such as WLAN radio data, network statistics, CPU load levels, etc.) Each data set is represented in an XML fragment of variable size. 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. Such an experiment produces a model of $5 * 10^9$ objects.

In general, sensor networks are currently only limited by the limitations of the technical infrastructure that supports them. Ambiguous computing and Smart Citys Markus: cites suggest sensor data models of arbitrary sizes.

Modeling Tasks 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 individual data sets (sub-trees) and to traverse these data sets (mostly time series). Analysis is usually done 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 standard, provides a set of xml-schemas (building upon other standards, e.g. GML) that function as a meta-model. CityGML, compared to other quasi standards (e.g. google KML) does not solely concentrate on the 3D measures, but allows to extend the covered information by more semantic attributes (e.g. materials, age, inhabitants, existing infrastructure, etc.). Geo spatial models usually come with different levels of details (LOD); CityGML distinguishes 5 LODs, 0-4) Markus: cite.

Model Size Like many cities, Berlin is currently establishing such a model. The current model of Berlin comprises 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.

Based on data published in [?] a LOD 1 building comprised of 12.5 objects, a LOD 2 building of 40 objects, a LOD 3 building of 350 objects. A LOD 3-4 building 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.*

Modeling Tasks Geo-spatial models are accessed by many people at the same time. Compared to model manipulation, model access (partial load) 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). Queries for distinct feature characteristics within a specific geographic location (i.e. with-in such containment hierarchies) is also common.

3 Model Fragmentation

3.1 Fragmention in General

All models considered in this paper have an inherent containment hierarchy. This means the models are graphs, but within these graphs there is a single distinct subset of edges that spans a tree. In EMF ecore based models, these trees consist of containment references. Hence, we also call these spanning trees containment trees.

In the large section, we saw that typical model applications mostly use aggregates of model objects, more precisely sub-trees of the model's containment trees. The model fragmentation approach, divides (i.e. *fragments*) a model along its containment tree. 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 **??**, where we use resources to realize fragments.

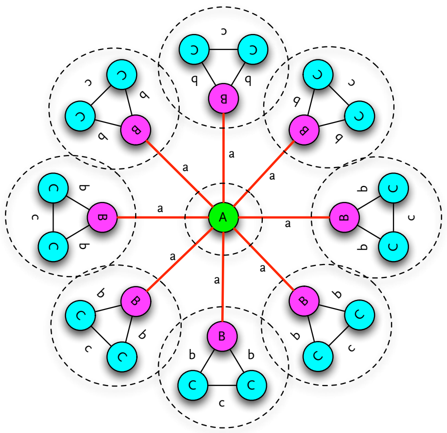

Fig. 2. An example pattern in a model and a fragmentation. The red links are inter fragment links.

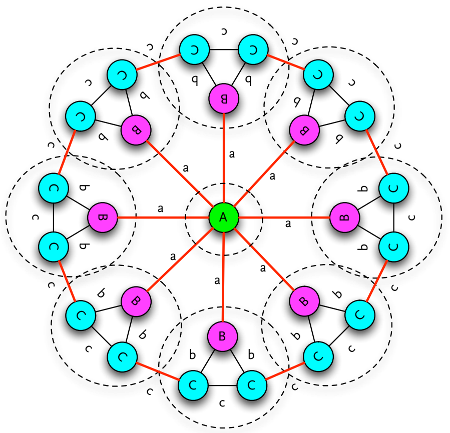

Fig. 3. Another example pattern. For a single feature there are two types of instances: inter- and intra-fragment links.

Fig. 2 and Fig. 3 present two examples.

3.2 Fragmentation Strategies

Of course a model has no fragmentation originally, further do we need to maintain a fragmentation when the model is manipulated, and we need 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.

There are other strategies imaginable. We will analyse one in more detail.

3.3 Meta-Model based Fragmentation

A meta-model defines possible models (graph of objects and links between objects) by means of classes and associations (i.e. structural features in EMF ecore). A good meta-model only allows models that are suitable for the use-cases of the envision model application. Containment references (indirectly the containment hierarchy) are already used by the meta-modeller to aggregate (e.g. in UML composition is a special case of aggregation) closely related objects. For the *meta-model based fragmentation* fragmentation strategy, we ask the meta-modeller to determine inter-fragment reference features as specific containment references. This way, the meta-model determines where the spanning-tree is broken into fragments.

Only containment reference 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 refernces) can become inter-fragment references *by accident* (like c-references in Fig. 3).

One the inter-fragment reference features are designated within the meta-model, it is easy to create and maintain fragmentations automatically and transparently; ref. to section **??**.

4 Related Work

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 and other object relational mappings (ORM) for Ecore; (4) morsa a EMF data-base mapping for non-relational data bases. [1]

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: 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 Markus: version 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.

[1] Mentioned memory and time efficiency in this section are taken from the evaluation section **??**.

Thirdly, CDO: CDO is a ORM for EMF. [2] 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: Different to CDO, morsa uses so mongoDB, 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 then with CDO: fast queries, slow creation, traverse, and load. Markus: You need decide where key-value stores are introduced and explained. Markus: Need to add cites.

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: $parse(size) = \mathcal{O}\left(size\right)$. 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

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

Fig. **??**. 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.

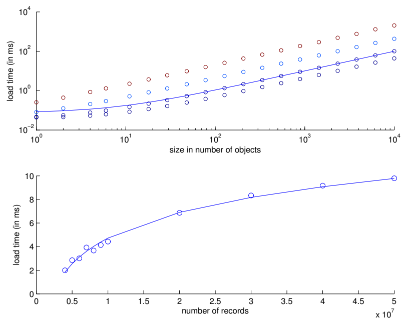

Fig. 4. Measure for liner parsing performance of EMF and logarithmic access performance of HBase.Markus: The dots in the EMF chart refer to different object sizes. either explain or remove.

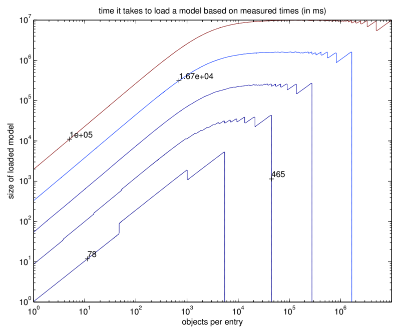

Fig. 5. Load times based on actual EMF parsing and HBase access measurements.

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 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.

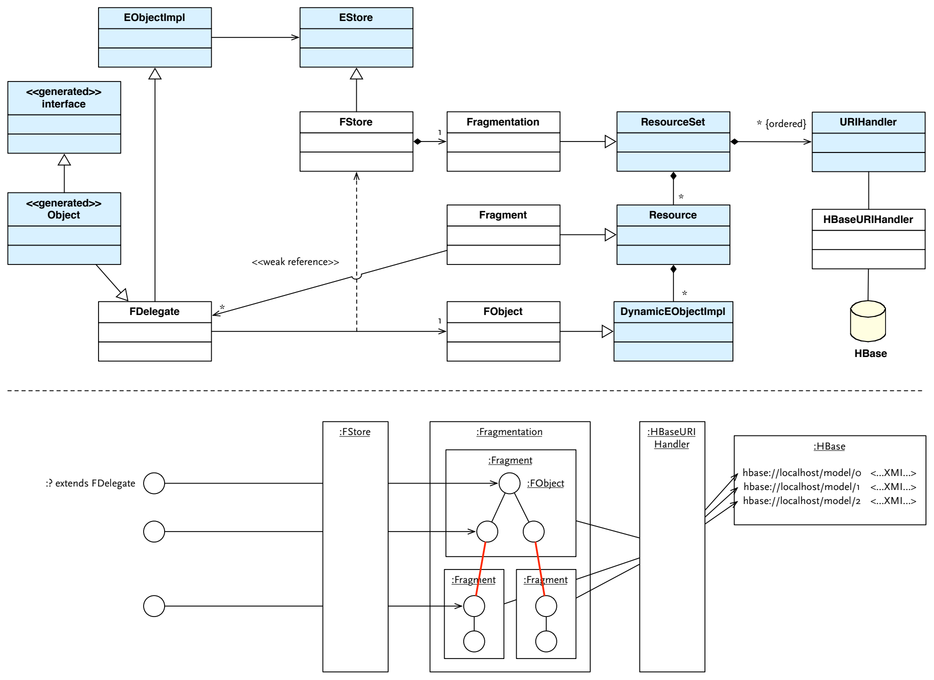

Fig. 6. EMFFrag architecture and example fragmentation.

Fig. 6 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 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.

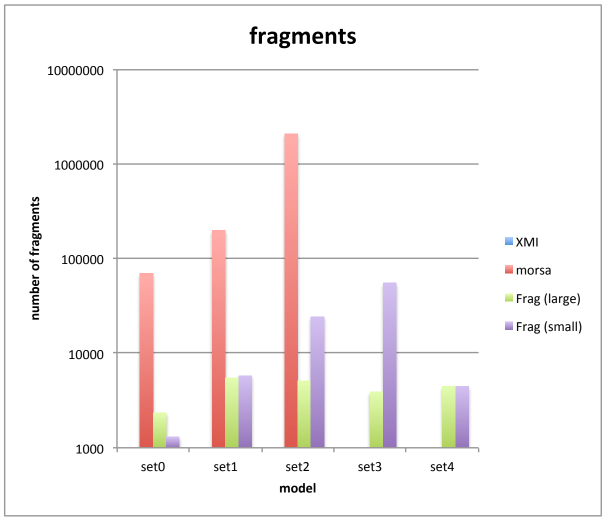

Fig. 7. Number of fragments used

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. 7 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. 8. EMF/XMI's memory usage is proportional 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.

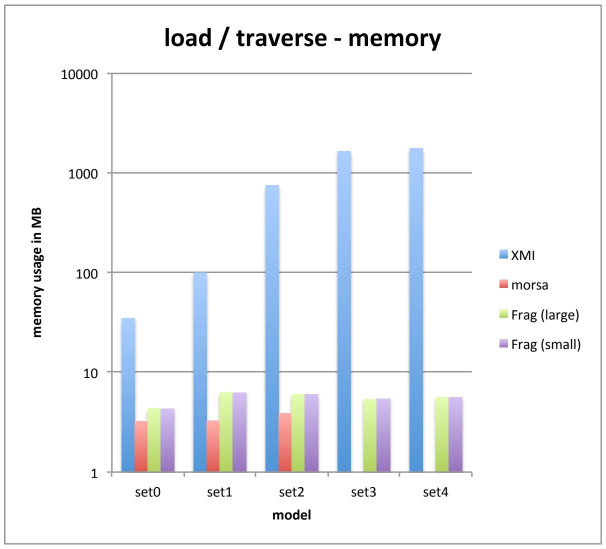

Fig. 8. Memory usage for a full model traversal

Full model access The execution times for accessing the complete models are shown in Fig. **??**. 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.

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

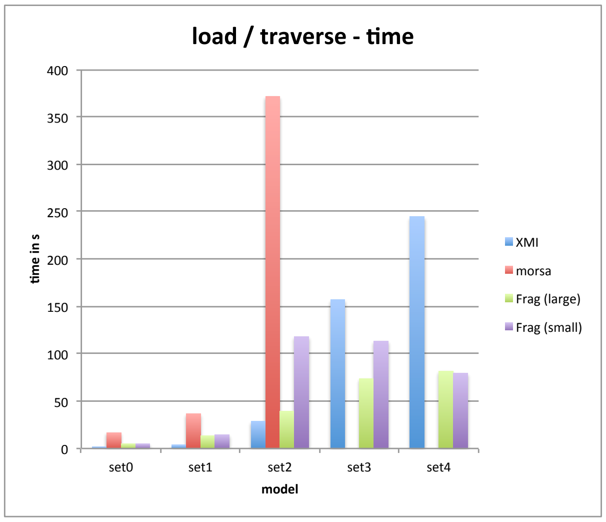

Fig. 9. Execution time for a full model traversal

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. 10.

Markus: TODO explain the query

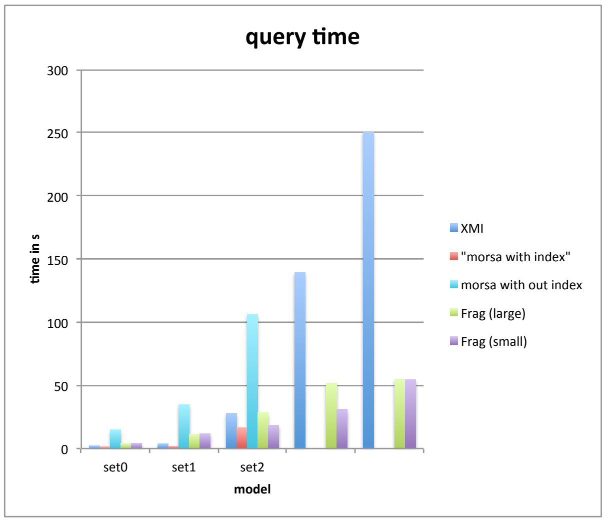

Fig. 10. Execution time for a specific query

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 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 callaed *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.

Markus: Measure first and then describe

8 Conclusions