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

1.1 Problem Statement

Large models (directed labelled graphs with a inherent spanning tree) cannot be maintained in computer memory. Large models need to be fragmented into smaller models. These smaller parts can be loaded and worked with. This is practical, since most applications only require to handle a small part of a large model at a time.

We distinguish between large and *extra large* models. Large models are to big for computer memory, but still small enough to be served from a single computer: large models are small enough for common hard drives and a single computer can bear the load from accessing the model (small number of users, infrequent access, only small chunks are accessed). Extra large models require a distributed data store, because they are either to big or are used in a way that a single computer can not handle the load. Safety, redundancy, dependability, etc. are other qualities that we associate with extra large models and their storage.

Modern key value data stores, can (in principle) store extra large models. They provide all necessary qualities associated with extra large models (distributed, safe, redundant, dependable, scalability, etc.). The problem is that models have to be fragmented so that they can be accessed in part. The concrete fragmentation of a model and its granularity directly influences the performance of storing, accessing, and working with the model. Here, performance regards execution time, CPU load, and RAM memory usage.

1.2 Fragmentation

Models (or data, or information) in this paper are directed labelled graphs (with an inherent spanning tree). Common model representations are EMF-based models or XML documents. We assume that nodes and all their outgoing edges always

belong together and are inseparable[1]. Often we will only write about nodes, but also mean all their outgoing edges (references to other nodes). For that matter a model is just a set of nodes (objects).

A *fragmentation* of a model is a set of sets of objects. The union of all sets in a fragmentation is the model; all sets are disjunct. The sets in a fragmentation are called *fragments*. The objects in each fragment contain a single spanning tree (sub tree of the models spanning tree). Each fragment is identified by a unique key. The fragment key identifies the root of that tree. Model edges can be associated with keys to reference objects in a different fragment.

1.3 Existing Storage Solutions and Fragmentation Strategies

There are three existing fragmentation strategies:

- *No fragmentation*, the model is not fragmented. Large models are not possible.
- *Total fragmentation*, each object of the model (a node with all its outgoing edges) is stored in its own fragment.
- *Manual fragmentation*, the user determines which objects of a model comprise which fragments.

1.4 Key Value Stores

Key value stores technical systems that persist large (hash-)maps. They can store arbitrary values (string or byte arrays) under arbitrary keys (string or byte arrays). There are different mature solutions for different environments (grid, cluster, cloud). A key-value combination is called *entry*.

Key value stores can be *distributed* or *non distributed*; they can be *ordered* or unordered. Generally values access complexity is logarithmic to the size the stored map (number of keys). This is true for both non distributed and distributed stores. In distributed stores complexity also depends logarithmically on the size of the store (number of computers (nodes) in the store). Reading an access value is linear to the size of the value. In distributed stores is a possibility to read values in parallel. In ordered stores, subsequent keys of an already accessed key can be accessed in constant time (scan). In general models are serialized and need parsing to be used. A loaded value (model fragment) is parsed in linear time (depending on the fragments size). We assume that parsing can only be performed on the client and no parallel parsing on different computers is possible.

1.5 Detailed Problems

There are several distinct problems with fragmentation of (extra) large models:

[1] This is in direct contrast to ER/SQL based models and stores, where links between entities are stored in separate tables

- Trivial fragmentation (no or total) might not produce optimal or even functional results.
- In extreme broad models one object might be too big (too many outgoing edges). Too big to be stored in RAM memory, or too big for reasonable and efficient use. This happens for example with sensor data collected over long periods of time. On sensor (object, node) holds references (edges) to thousands of thousands of values.
- There are several parameters that influence optimal fragmentation: parsing speed, data store access speed, whether parsing or accesses different entries can be parallelized (distribute scenario). Random access (random read) vs. sequential access (scan).
- Fragmentation in general depends on model usage patterns. These cannot be known at model creation and might change over time.

1.6 Terms

Specific terms are: *model*, *large model*, *extra large model*, *fragmentation*, *fragment*, *no fragmentation*, *total fragmentation*, *manual fragmentation*, *key-value data store*, *entry*, *key*, *value*, *(un-)ordered store*, *(non-)distributed store*. Further terminology may be introduced later.

2 Related Work

3 Examples for (Extra) Large Models

3.1 Software Projects – e.g. Linux Kernel

What are software code models? We use the term model to describe MDSD artifacts. Originally these artifacts were models on a high level of abstraction, but today programming code, can also be understood as models. For example IDEs such as eclipse JDT or eclipse CDT internally maintain programming code as ASTs (primitive models) in order to provide advanced IDE features such as outlines, error annotations, type and call hierarchies, and code completion. It is safe to assume that programming code contains far more information compared to high-level models. Since we are looking for extra large models, we will concentrate on software code models.

Advances in language workbenches suggest to replace these IDEs with eclipse EMF based and generated tools. Advances in comparison and versioning of models will soon allow to replace per-line-text-file-based versioning systems (e.g. CSV, SVN, GIT) with model based (e.g. EMF-based) systems that version on a per mode-object- or even per-object-attribute basis.

How big is are existing software code models? How large are these software code models? 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.

In order to determine corresponding model sizes, we need to know how much model objects each LOC represents. To estimate this, we used the eclipse CDT parser as tool and the linux kernel as sample. The CDT parser is specialized to parse C/C++ code in its un-preprocessed form. Thus, the resulting ASTs are not bloated with information injected (with lots of duplicates) during preprocessing. Of course ASTs are not software models as they are understood in the OMG/EMF world, but we can assume that proper C/C++ models will contain one object for each AST node. Markus: This needs a reference. Parsing the current kernel version, counting its AST nodes, and comparing this number to the kernel's SLOC number gives us a good estimate for objects per SLOC (at least for the kernel code, probably for all C code, and maybe it is also a good estimate for other similar programming languages, e.g. C#, Java). We will use this estimate to extrapolate the model size of other kernel versions and other software projects.

Eventually, we want to store software models and its versions. Of course, we only want to store the differences between versions. A software projects model size is determined by the size of the differences that produced the current software model. To determine this size, we need to know how much LOCs were added, removed, and modified in a software project. To estimate this, we use the versioning system GIT as tool and the linux kernel GIT repository as sample. With gitstats we can determine the number of added, removed, and changed lines throughout the repository history. With these numbers, the estimated objects per SLOC and the recorded SLOCs for different kernel versions we can estimate the model size of a kernel software code model repository. We can also assume that other software projects have a similar proportion of added, removed, and modified lines, and transfer these estimates to other software projects.

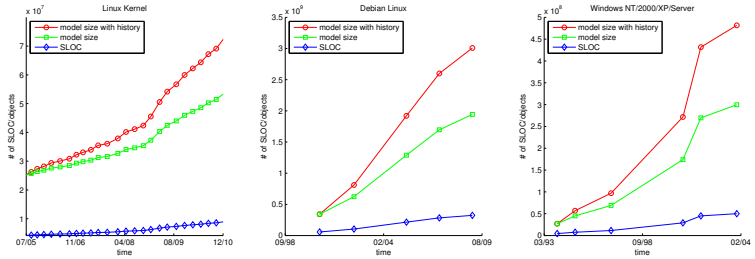

Fig. 1. SLOC and estimated number of objects for popular large software projects.

Fig. 1 shows the LOC numbers and estimated object numbers in corresponding software code models. The charts present different software projects and how size developed over time. Markus: kernel, linux(debian)?, windows - numbers based on kernel git alone, SLOC numbers and transferred kernel estimates.

How big are software code models in the future? Some believe that the average size software (and software code models for that matter) grows faster than linear following Moore's Law. Maciej Soltysiak for example has analysed the grows of the linux kernel code base and observed exponential growth. A counter argument is that software not only bound by hardware limitations (i.e. Moore's Law) but increasingly by software complexity and therefore will not grow exponentially. Never the less, it is safe to assume that software code models will be larger in the future.

3.2 Geospatial Models – e.g. a 3D Model of Berlin in CityML

What are geo-spatial 3D city models? 3D city models are a good example for structured desecrate geo spatial information. In these models geo spatial features like windows, rooms, apartments, houses, streets, boroughs, and cities are composed to form a complex graph. 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 distinguished 5 LODs, 0-4).

How big is an existing 3D city model? 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.

How big will 3D city models get in the future? 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*.

3.3 Heterogenous Sensor Data – e.g. HWL and Smart City

What are sensor data models? 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 heterogenous: 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. Components provide data as structured XML documents. Software components are themselves structures according to the software architecture.

We build a software that collects this XML data, transforms it into an EMF model, stores it in a database, and allows analysis with EMFs generated Java APIs and model transformation languages (e.g. [**?**]).

How big are existing sensor data models? 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 $5 * 10^9$ objects.

How big can sensor data models get in the future? Sensor networks are currently only limited by the limitations of the technical infrastructure that supports them. Will, infrastructure, and money provided, sensor networks can produce models of virtually unlimited size.

4 Possible performance gains for loading models from local key-value data stores

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.

4.1 Optimal Fragmentation for non Distributed and Unordered Data Stores

First, we need to define a few functions that provide the performance of all operations involved in loading a model. We define the *parse* function that determines how long it takes to parse a serialized model of a given *size*.

$$parse(size) = \mathcal{O}(size)$$

The next function *access* determines how long it takes to find an entry in a local key-value data store based on the number of key $\#keys$, i.e. number of entries:

$$access(\#keys) = \mathcal{O}log(\#keys)$$

We will use the following parameters: The total model $size$, the average number of objects per entry ope, the size of the model to load $lsize$, and $part$ the average percentage of an entry's objects that belong to the loaded model if at least one object is part of the loaded model.

The time t to load a model with this parameters is:

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

The two extreme examples are: (only one big entry) $ope = size$, $part = lsize/size$, $t = \mathcal{O}(size)$, and (one object per entry) $ope = 1$, $part = 1$, $t = \mathcal{O}(lsize\,(log(size) + 1))$.

Analysis In the optimal model distribution (one only wants to load models that exactly constitute one entry) $ope = lsize$ and $part = 1$ the time is $t = \mathcal{O}\left(log(\frac{size}{lsize}) + lsize\right)$.

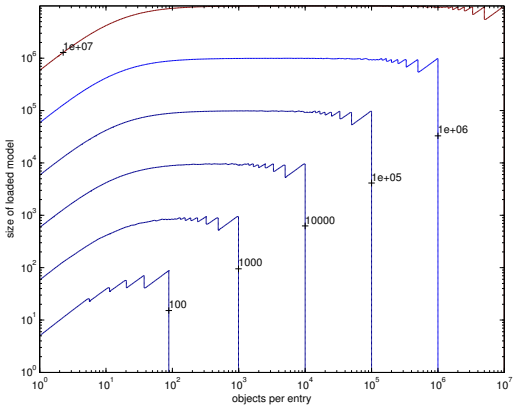

Fig. 2. Load times

The plot in Fig. 2 shows the relation between objects per entry, load size, and the time it takes to load. The contours show loads that take the same time. The plot does not account for any $parse = m * size + n$ and $access = m * log(\#keys) + n$ factors (m, n). Depending on actual factors (see next section) 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 object per entry numbers are more desirable (logarithmic parts of contours are longer).

Measurements 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. 2. As expected the parsing performance is linear and the data store access behaves logarithmic. The plot in Fig. 4 is similar to Fig. 2, but uses $parse$ and $access$ factors based on the measurements (actually the interpolated functions shown as lines in the plots).

From this plot, we can see what fragmentation allows compared to no fragmentation ($ope = size$) or complete fragmentation ($ope = 1$). No fragmentation always takes the full time (10s in this scenario). Of course optimal results can be obtained if ($lsize = ope$). This optimal result allows to load models a 1000 times bigger at the same time than total fragmentation.

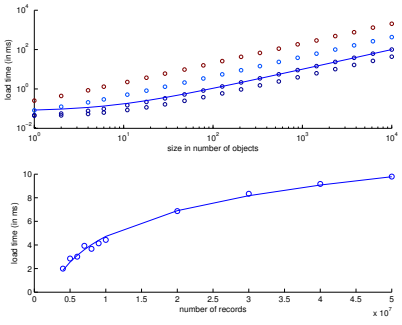

Fig. 3. Measure for liner parsing performance of EMF and logarithmic access performance of HBase.

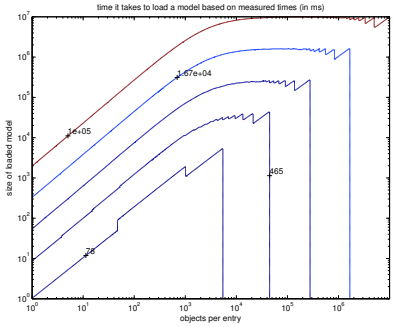

Fig. 4. Load times based on actual EMF parsing and HBase access measurements.

4.2 Optimal Fragmentation in Ordered but Non Distributed Stores

Analysis In ordered stores, entries can be scanned. If keys are chosen intelligently, a loaded model consisting of multiple fragments can still be loaded with only one access and subsequent scans. Assuming that all keys are always chosen in the optimal way and that the time between scans is close to 0: as long as $ope < lsize$ the required load time equals $parse(lsize)$. We can have more fine grain fragmentation without loosing performance if larger models are loaded.

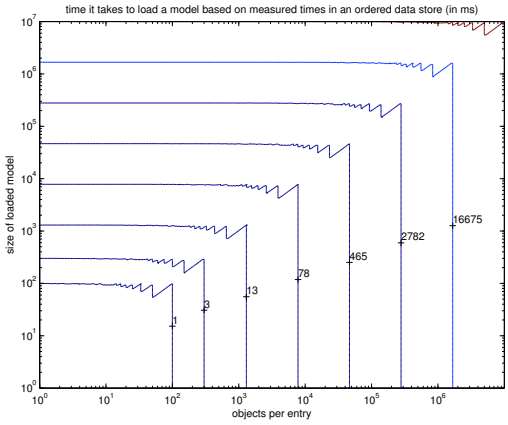

Fig. 5. Load times based on actual EMF parsing and HBase access measurements in an ordered data store.

Measurements Fig. 5 shows that the time it takes to load models of a certain size, is independent from the fragmentation as long as the fragments are small enough, i.e. $ope < lsize$.

4.3 Optimal Fragmentation in Unordered but Distributed Stores

Analysis In a distributed store serialized model fragments are loaded from nodes in a distributed store network. The serialized model fragments are then parsed on the client. To model this, we need to distinguish between load and parse time:

$$parse(size) = \mathcal{O}(size)$$
$$load(size) = \mathcal{O}(size)$$

Furthermore, access times in a distributed data store are different; they do not only depend on the number of keys $\#keys$, but also on the number of nodes $\#nodes$:

$$access(\#keys, \#nodes) = \mathcal{O}log(\#nodes) + \mathcal{O}log(\frac{\#keys}{\#nodes})$$

In an optimal fragmentation and distribution of fragments, model fragments can be accessed and loaded in parallel. The number of parallel access and load processes is determined by the number of nodes in the data store. The average overall model load time (for $part = 1$) is:

$$t = \frac{lsize}{ope}\left(parse(ope) + \frac{access(\#nodes, \frac{size}{ope}) + load(ope)}{\#nodes}\right)$$

Note that this model does not allow parallel loading and parsing of the model.

5 Fragmentation Strategies

5.1 Manual Fragmentation

5.2 Metamodel Defined Fragmentation

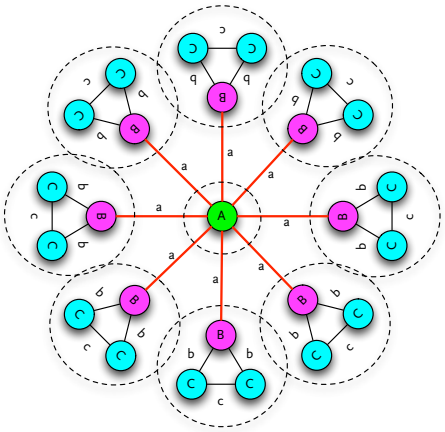

Fig. 6. An example pattern in a model and a fragmentation. The red links are inter fragment links.

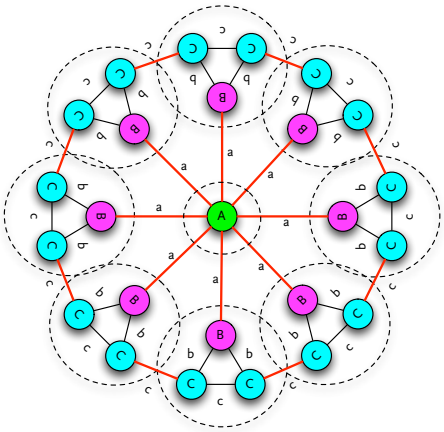

Fig. 7. Another example pattern. For a single feature there are two types of instances: inter- and intra-fragment links.

Theory Access patterns for a model are strongly influenced by its metamodel. Metamodels are tiny in comparison to their large instances. If you imagine looking from above onto a large model, the metamodel types of its objects form patterns. How we access a model is also influenced by its metamodel, since all algorithms doing something with a model are programmed against its metamodel. Hence, optimal fragmentation goes along this patterns. Most fragments will have the same structure, and fragments are connected through structural features of only few different types. One way to define fragmentation is to mark these fragment crossing structural features.

Fig. 6 shows a simple example metamodel type pattern. The instances (links) of feature a cross fragment borders (inter fragment links). All other links are intra-fragment links. The situation is a little more complicated in Fig. 7. Here the links of feature c have both inter- and intra-fragment instances.

If we want to describe fragmentation by marking features as inter- or intra-fragment features, it would work for the example in Fig. 6, but not for the example in Fig. 7. Obviously, we need further restrictions.

Models (as used in this paper) always have a inherent spanning tree. The spanning tree is formed from links that are instances from containment features. All instances of containment features are part of the spanning tree. If we only allow containment features to be inter-fragment features then the instances of inter-fragment feature will always define a unique fragmentation. Markus: Proof?

Implementation This describes an implementation based on EMF.

5.3 Automated Fragmentation based on Expected Range Queries

5.4 Automated Fragmentation based on Access Patterns

5.5 Fragmentation of Even Models

Analysis At the beginning, we will look at *even* models. A model is even, if its inner structure suggest fragmentation into equal pieces. For example, an intuitive way to fragment a OO software model is to put each package into one entry. This is an uneven model, since packages have different sizes. Another example is sensor data, sensor data produced at each point in time or on each node has the same size. If one puts each sensor reading or each node into one entry, the entries will have similar size.

Previously, we were looking the gains achieved with optimal fragmentation. While optimal fragmentation is plausible in manually fragmented models for a single specific loaded model (e.g. accessing single sensor readings in ClickWatch). Optimal fragmentation is unlikely for different loaded models (even impossible for models of different size).

In general, we can assume that the smaller *ope* is compared to *load*, the more likely it is that much of each entry is part of the loaded model. In other words, the smaller my entries are, the more likely it is that much or all if a single entry is part of the loaded model. We will model *part* accordingly.

6 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.

6.1 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:

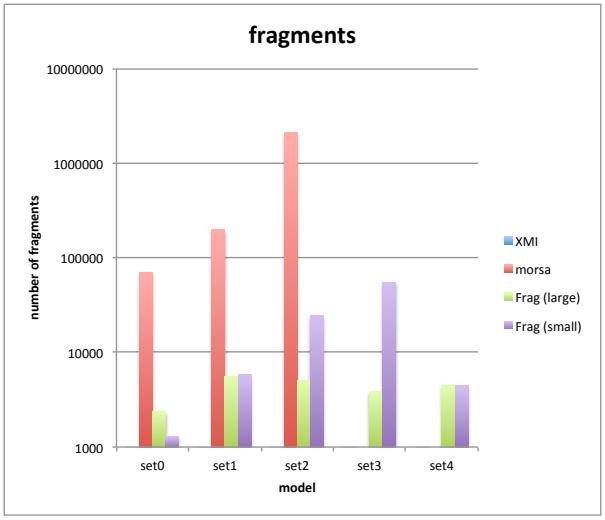

Fig. 8. Number of fragments used

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. **??** 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. **??**. 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.

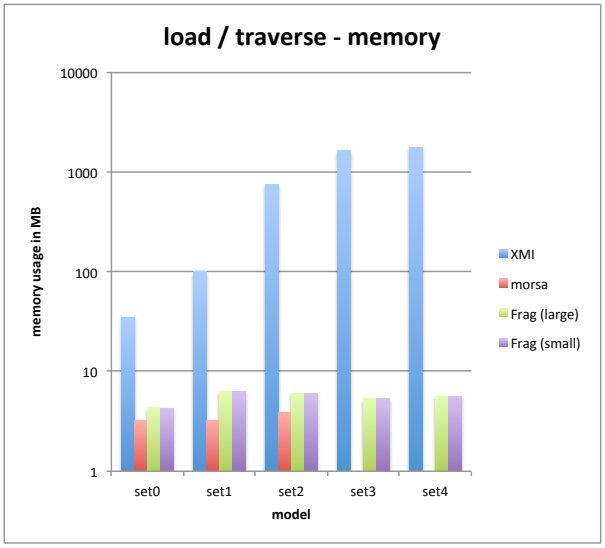

Fig. 9. Memory usage for a full model traversal

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 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. **??**.

Markus: TODO explain the query

6.2 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

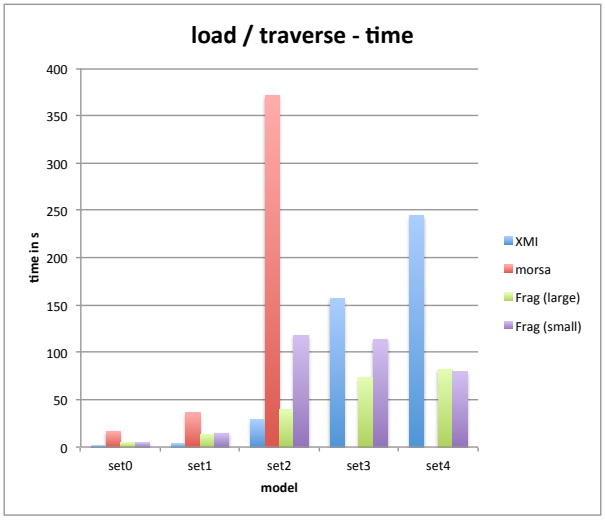

Fig. 10. Execution time for a full model traversal

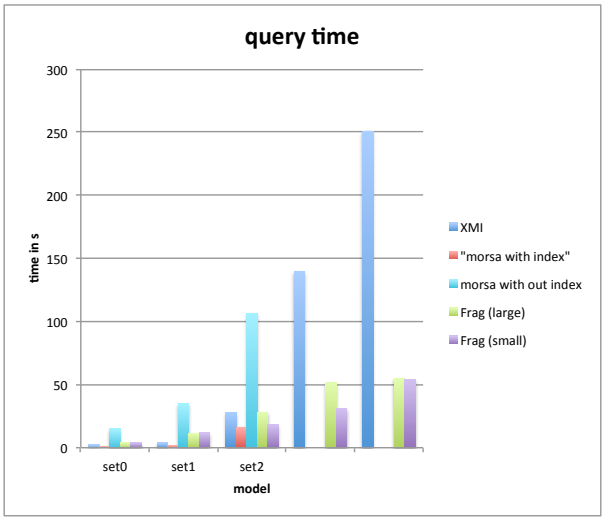

Fig. 11. Execution time for a specific query

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 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

7 Conclusions