Map/Reduce is the programming model in cloud computing. It enables the processing of data sets of unprecedented size, but it also delegates the handling of complex data structures completely to its users. In this paper, we apply Map/Reduce to EMF-based models to cope with complex data structures in the familiar an easy-to-use and type-safe EMF fashion, combining the advantages of both technologies. We use our framework EMF-Fragments to store very large EMF models in distributed key-value stores (Hadoop’s Hbase). This allows us to build Map/Reduce programs that use EMF’s generated APIs to process those very large EMF-models. We present our framework and two example Map/Reduce jobs for querying software models and for analyzing sensor data represented as EMF-models.

Keywords
Cloud Computing, EMF, Meta-Modeling, Big Data

1. INTRODUCTION
Two of the most dominant words in 2012 are probably Cloud Computing and Big Data. Or as Borkar et al. [10] put it: “Virtually everyone, ranging from big web companies to traditional enterprises to physical science researchers to social scientists, is either already experiencing or anticipating unprecedented growth in the amount of data available in their word, as well as new opportunities and great untapped value that successfully taming the Big Data beast will hold.”

There are two requirements that have to be balanced: unlimited and cheap scalability of clusters/clouds and a safe and flexible programming model. Among others, there are two major approaches: parallel relational data bases and SQL and distributed hash-table based file-systems (plus key-value stores) and Map/Reduce. SQL databases provide a structured approach and a well known, intuitive query language, but it is hard to build and maintain large clusters. Map/Reduce scales well, servers can come and go, but dealing with complex data structures is delegated to users, and it is assumed that Map/Reduce covers embarrassingly parallel problems only. [10, 8]

How can software modeling help? Dealing with complex, interconnected, well typed data-structures is the core trait of model driven software engineering. Meta-models allow to define fine grained object oriented types and references; constraints can elaborate meta-models, and semantics can be assigned to data structures. There is a large zoo of model transformation languages, programming frameworks, and other methodologies. Software modeling (especially model driven architecture) provides the tools to integrate different typed structures (typically called languages). Furthermore, software modeling (especially EMF-based) integrates well with other core technologies like XML, ontologies, or even databases (e.g. via ORMs like CDO). But, software models are usually small enough to fit into main memory and scalability is less of an issue.

In this paper, we discuss how EMF and Map/Reduce (i.e. with Apache Hadoop) can be combined. Our approach consists of two parts: storing large EMF models in a distributed key-value store (Hadoop’s HBase) and then using those models from within the Map/Reduce programming model using EMF’s generated APIs. The paper is organized as follows: The next section 2 describes our general architecture, all the frameworks we use, and how they interact. The examples section 3 describes our experience in building two Map/Reduce algorithms: a simple one and one more complex one. We close the paper with sections on related work and conclusions.

2. ARCHITECTURE
Our proposed architecture uses EMF on top of a cloud/grid computing infrastructure (refer to Fig. 2). We use Hadoop/HBase, but the approach should work for other frameworks as well. The next two subsections describe the two parts of our approach: First, EMF-Fragments—an extension to the EMF resource API that allows us to store large models in a distributed key-value-store (HBase). Secondly, new Hadoop base classes for the definition of map and reduce functions based on EMF models rather than raw files or datastore entries.

2.1 EMF and Key-Value Stores
We build a model persistence framework for EMF [16] called EMF-fragments [12, 13]. EMF-Fragments is different from...
frameworks based on object relational mappings (ORM) like Connected Data Objects (CDO) [17]. While ORM mappings map single objects, attributes, and references to database entries, EMF fragments map larger chunks of a model (fragments) to URIs. This allows to store models on a wide range of distributed data-stores including distributed file-systems and key-value stores (e.g., Hadoop’s HBase).

EMF-Fragments uses and extends the regular EMF Resource API. Clients designate references that shall fragment the model in the meta-model. Fig. 2.1 exemplifies fragmentation on meta and model level. EMF-Fragments then automatically and transparently creates and manages resource and their content. This allows to control fragmentation without the need to trigger it programatically. Fragments/resource are continuously managed in the background, i.e., resources are loaded and also unloaded as necessary. Each fragment is backed by a EMF resource and identified by its URI. Resources and URIs are canonically mapped to keys and values in a key-value store (e.g., Hadoop’s HBase). From the client perspective one just uses the regular reflective (refl.) or generated (gen.) EMF-APIs.

There are two general ways to describe fragmentation in the meta-model. The first one is to mark containment reference in the Ecore meta-model with annotations. This tells EMF-fragments to create a new resource for each value in those references. This works well when the number of anticipated values per feature is relatively low. This is usually the case in software models. In a large Java code base for example, we have a large number compilation units (i.e., Java class files), but a single package only contains a small set of sub-packages and compilation units (example in Fig. 2.1).

With the described method the container has to keep references to all its contents. If we have millions of values the references alone require too large resources even though the values are stored in different resources. Therefore, we also need a second approach: clients can use predefined index classes to define relationships with large value sets. This happens for example, if we want to store sensor readings in EMF: each sensor might have million of readings depending on the period of time and sample rate (example in Fig. 2.1).

2.2 EMF and Map/Reduce

The Hadoop web site describes its Map/Reduce capabilities like this: "Map-Reduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner." Map/Reduce has proven itself as a successful programming model. But, Map/Reduce has two known disadvantages. First, Map/Reduce is only suited for so called embarrassingly parallel problems. Engineers struggle to implement algorithms for problems that do not canonically fall into independent parts. Secondly, Map/Reduce deliberately ignores the structure of the data that it is used to analyze. This issue delegated to users. The consequences are many fold; examples are hardly reusable algorithms based on proprietary data-structures or slow, defective, and proprietary parsers.

We use EMF to help with the second problem: the EMF-Fragments framework introduced in the prior section stores...
3. EXAMPLES

3.1 Querying Large Software Models

In this example, we take a large Java code base represented as an EMF model. The goal is to execute a query: find all public and static methods that return a value with the same type than the class that contains them. This example is taken from the 2009 Grabats Workshop contest [3]. A simplified excerpt of the Java meta-model and its fragmentation is shown in Fig. 2.1.

We consider this problem as one of the *embarrassingly parallel problems*: the query can be executed on each CompilationUnit individually. Each map task takes a CompilationUnit (which are also always the root elements in a fragment/resource). If a task finds a method that fulfills the query, an id of that method is added to the results. A standard reduce task (e.g. identity) simply accumulates the results.

3.2 Seismic Sensor Networks and Road Traffic Analysis

In this example, we want to analyze time series data. At our lap, we work with Wireless Sensor Networks (WSN), each node of our WSNs is equipped with a 3-axis accelerometer sensor. We use these networks for our research in earthquake early warning [4] and traffic surveillance. A simplified excerpt of the meta-model used to represent sensor data is given in Fig. 2.1. Each sensor reading is stored in an individual fragment, hence in its own key-value-entry.

This problem is not as *embarrassingly easy* as the former one: during time series analysis consecutive values have to be computed with each other. A moving average for example operates on a sliding window of values and not on individual values. In our concrete analysis, we are applying the following operations on our data (refer also to Fig. 3.2). First, we start with the unfiltered raw time series for each axis. Secondly, we calculate a moving average and subtract that average from all values for normalization. Thirdly, we apply a Fast Fourier Transformation (FFT) to a sliding window. Fourthly, we accumulate the FFT values of certain frequencies to create a measure of the intensity in certain frequencies for each point in time.

This analysis obviously operates on a series of sequential values. To compute this with Map/Reduce, we have to apply a non trivial mapping (the solution is still quite simple): Within each map task, we analyze a sub-sequence of sensor readings. The size of the sub-sequence is determined by the overall number of readings and the amount of computing nodes in our Hadoop network. HBase is a sorted key-value-store and each node in the network will hold a set of sequential sensor readings. The goal of our mapping is to let each map task to (almost) only analyze the readings sequence on a single node. Due to the nature of the analysis and the use of sliding windows a small overlay has to happen and each map task has to consider a small amount of non local data. Fig. 3.2 depicts this mapping. The reduce task, takes the values of each map tasks and accumulates a seamless sequence.

4. RELATED WORK

4.1 Structured Data in Cloud Computing

There a several approaches to compute structures data in *share nothing* grids/clouds. Pig (Yahoo!), Hive (Facebook), and Jaql (IBM) borrow from relational databases and try to bring a structured data approach to Map/Reduce based computing. All three raise the level of abstraction: they translate from their respective high-level languages to Map/Reduce jobs, but also allow Java-programmed user defined functions. All three approaches mimic relational schemes and operators like *filter, join, group, union, etc.*
normalization FFT on each window (here two example windows)
energy in each window for certain frequencies
the particular phenomena here, is a public bus running by our WSN

Figure 5: Example frequency spectrum analysis of seismic acceleration readings for the traffic monitoring.

Figure 6: Time series analysis with Map/Reduce. The use of sliding windows requires small overlaps depending on window size. (*) depending on window size the first sensor readings do not produce usable results.

Another structured data approach for grid/cloud computing are graph databases. In contrast to relational approaches, graph databases focus on efficient storage and navigation of neighbor relations (known as index-free adjacency). The meta-model is usually fix (vertices and edges), but custom attributes can be provided. The programming model corresponding to Map/Reduce is called Bulk Synchronous Parallel (BSP). Google’s Pregel is a well known implementation of BSP [6].

With EMF-Fragments we basically mimic both approaches. The index classes (refer to Fig. 2.1) allow to create large relations with index (relational approach) and normal EMF references mimic the graph database approach. We currently map EMF data to key-value stores and apply the Map/Reduce programming model. Mapping EMF to graph databases and using BSP should also be feasible.

4.2 EMF Database Persistence

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

There are at least three different approaches to deal with large EMF models: (1) EMF resources, where a resource can be a file or an entry in a database; (2) CDO [17] and other object relational mappings (ORM) for Ecore; (3) morsa [7] a EMF data-base mapping for non-relational databases.

First, EMF resources [16]: With EMF clients can store one model in multiple resources. Originally, objects in different resources could only reference each other via non containment references. Since EMF 2.2 containment proxies are supported. EMF support lazy loading: resources do not
have to be loaded manually, EMF loads them transparently once objects of a resource are navigated to. Model objects have to be assigned to resources manually (manual fragmentation). To actually save memory the user has to unload resources manually too. The framework MongoEMF [5] maps resources to entries in a MongoDB [9] database.

Secondly, CDO [17]: CDO is a ORM for EMF. It supports several relational databases. Classes and features are mapped to tables and columns. CDO was designed for software modeling and provides transaction, views, and versions. Relational databases provide mechanisms to index and access objects with SQL queries. This allows fast query execution, if the user understands the underlying ORM.

Thirdly, morsa [7]: Different to CDO, Morsa uses mongoDB [9], a NoSQL database that realizes a key-value store (see below). Morsa stores objects, their references and attributes as JSON documents. Morsa furthermore uses mongoDB’s index feature to create and maintain indices for specific characteristics (e.g. an objects meta-class reference).

5. CONCLUSIONS
We presented the framework EMF-Fragments for the distributed storage of very large EMF models and we described how Map/Reduce jobs can be used to analyze those models. We therefore showed the general usability of EMF as an approach for structured data and data analysis in grid/cloud computing. The possibility to store large graphs and large sets of relational data covers a large set of possible problems.

Map/Reduce is a very simple programming model and critiques say it is only feasible for embarrassingly easy parallel problems. Especially our second example showed, that (despite a structured data representation) Map/Reduce algorithms have to be designed carefully and not all problems are easily mapped onto this programming model.

As future work, we want to examine how Map/Reduce jobs can be generated from rule-based model transformations. This would provide a further step in hiding the complexity of cloud computing. We plan more realistic case studies based on our experimentation framework ClickWatch [14, 11] and our wireless sensor network test-bed HWL [15].

6. REFERENCES