White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of York.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/211856/

Version: Accepted Version

Proceedings Paper:
Barmpis, Konstantinos, Shah, Seyyed and Kolovos, Dimitrios S. (2015) Towards incremental updates in large-scale model indexes. In: Modelling Foundations and Applications - 11th European Conference, ECMFA 2015 Held as Part of STAF 2015, Proceedings. 11th European Conference on Modelling Foundations and Applications, ECMFA 2015 Held as Part of International Conference on Software Technologies: Applications and Foundations, STAF 2015, 20-24 Jul 2015 Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer, ITA, pp. 137-153.

https://doi.org/10.1007/978-3-319-21151-0_10

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UNIVERSITY OF LEEDS
University of Sheffield
UNIVERSITY of York

Towards Incremental Updates in Large-Scale Model Indexes

Konstantinos Barmpis, Seyyed Shah and Dimitrios S. Kolovos

Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK
{kb,s.shah,dkolovos}@cs.york.ac.uk

Abstract. Hawk is a modular and scalable framework that supports monitoring and indexing large collections of models stored in diverse version control repositories. As such models are likely to evolve over time, responding to change in an efficient manner is of paramount importance. This paper presents the incremental update process in Hawk and discusses the efficiency challenges faced. The paper also reports on the evaluation of Hawk against an existing large-scale benchmark, focusing on the observed efficiency benefits in terms of update time; it compares the time taken when using this approach against the naive approach used beforehand, and discusses the benefits of combining the two, gaining improvements averaging a 70.7% decrease in execution time.

1 Introduction

As discussed in [9, 11], the ability of modelling and model management tools to scale up in a collaborative development environment is essential for the wider adoption of MDE. This paper contributes to the study of scalable techniques for collaborative modelling, by presenting and empirically evaluating an incremental approach for indexing evolving models stored in file-based repositories (such as Git or SVN). Section 2 introduces the rationale behind model indexing and provides a brief overview of an existing model indexing framework (Hawk [1]). Section 3 presents a naive and an incremental approach for updating model indexes in response to changes to models stored in file-based repositories that Hawk monitors. Section 4 evaluates their performance on various mutants of models from an existing large-scale benchmark. Section 5 discusses related work and Section 6 concludes the paper and identifies directions for further work.

2 Background

This section overviews model indexing, providing motivation for using it in this domain, and introduces the reader to Hawk, our model indexing framework.

2.1 Model Indexing

In a collaborative environment, models need to be version-controlled and shared among many developers. The default approach for doing this is to use a file-based version control system such as Git or SVN. This has certain advantages as such version control systems are robust, widely-used and orthogonal to modelling tools, the vast majority of which persist models as files. On the downside, since such version control systems are unaware of the contents of model files, performing queries on models stored in them requires developers to check these models out locally first. This can be particularly inefficient for global queries (e.g. is there a UML model in my repository that contains a class named "Customer"?) that need to be executed on a large number of models. Also, file-based version control systems do not provide support for model-element level operations such as locking or change notifications. To address these limitations, several open-source and proprietary model-specific version control systems such as CDO, EMFStore and MagicDraw's TeamServer have been developed over the last decade. As discussed in detail in Section 5, while such systems address some of the limitations above, they require tight coupling with modelling tools, they impose an administration overhead, and they lack the maturity, robustness and wide adoption of file-based version-control systems.

In what can be seen as a happy medium between the two approaches to model version control, *model indexing* is an approach that enables efficient global model-element-level queries on collections of models stored in file-based version control systems. To achieve this, a separate system is introduced which monitors file-based repositories and maintains a fine-grained read-only representation (graph) of models of interest, which is amenable to model-element-level querying. Previous work [2, 3] has demonstrated promising results with regards to query execution times, with up to 95.1% decrease in execution time for querying model indexes [2], compared to direct querying of their constituent EMF XMI-based models, and up to a further 71.7% decrease in execution time, when derived (cached) attributes were used [3]. This motivates us to improve upon this technology by improving the efficiency of handling model evolution in such model indexes.

2.2 Hawk

Hawk is a model indexing system that can work with diverse file-based version control systems (VCS) and model persistence formats whilst providing a comprehensive API through which modeling and model management tools can issue queries. Hawk needs to be scalable so that it can accommodate large sets of models, and non-invasive (the VCS repositories, where the primary copies of the monitored models are stored, should not need to be modified or configured). In order for Hawk to be able to index heterogeneous models in a back-end agnostic manner, it provides two abstraction layers:

– **Model Abstraction Layer** This layer provides a set of abstractions for representing heterogeneous models and metamodels in memory. Inspired

by EMF's respective abstractions, *metamodel resources* contain types/metaclasses (grouped in packages), which have typed attributes and references, as well as annotations. *Model resources* contain objects representing model elements, which have values for the attributes and references of their type.

– **Graph Database Abstraction Layer** Extensive benchmarking showed that graph databases such as Neo4J and OrientDB perform significantly better than other technologies (e.g. relational databases) [2, 14] for the types of queries of interest to a system like Hawk. To avoid coupling with a specific graph database, this layer aims at providing a uniform interface for querying and manipulating graph databases in an implementation-independent manner. It is worth noting that implementations of this layer can conceptually be used to connect to any back-end technology, but will suffer in performance if the data model is not similar with the graph model used here.

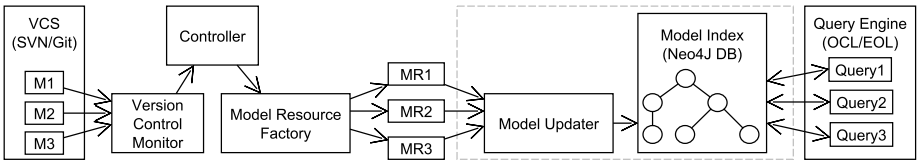

Fig. 1. Overview of the relevant Hawk architecture

Hawk comprises components which can monitor a set of version control repositories, parse and index models of interest stored in them. Figure 1 shows some of the key components of Hawk and their interactions; in the figure, M1–M3 represent model files and MR1–MR3 represent in-memory *model resources* obtained by parsing M1–M3. Below is a brief description of these components:

At the center of Hawk lies the *Controller* of the system, which knows which Hawk components are currently active and is responsible for synchronizing with any changes made to monitored files.

Resource Factories are used to parse metamodel and model files and create the relevant metamodel and model *resources* described above, which are given to the *Controller* to be propagated to the index. They know which files they can parse and also provide a way to give any statically available metamodels to the *Controller* (such as a UML metamodel – without them having to be manually registered).

Version Control Managers are used to poll monitored version control systems in order to get any changes (deltas) to model files of interest. If any such delta is found, it passes it on to the *Controller* so that the changes can be propagated to the model index.

Metamodel Updaters are used to insert *metamodel resources* to the index. As Hawk does not currently deal with metamodel evolution, only the first version of a metamodel is indexed. Model *Updaters* are used to update the index with a new version of a model. In the sequel we examine how such updates can be optimized, focusing on the components found inside the dashed box of Figure 1.

Prototype implementations of the various components exist as Java plugins[1] and support: XMI EMF, BIM[2] EMF and Modelio[3] UML models and metamodels, monitoring of folders on the local machine and on SVN version control repositories, using Neo4J (version 2) Graph NoSQL database for persistence and using Epsilon's EOL [10] as a query engine.

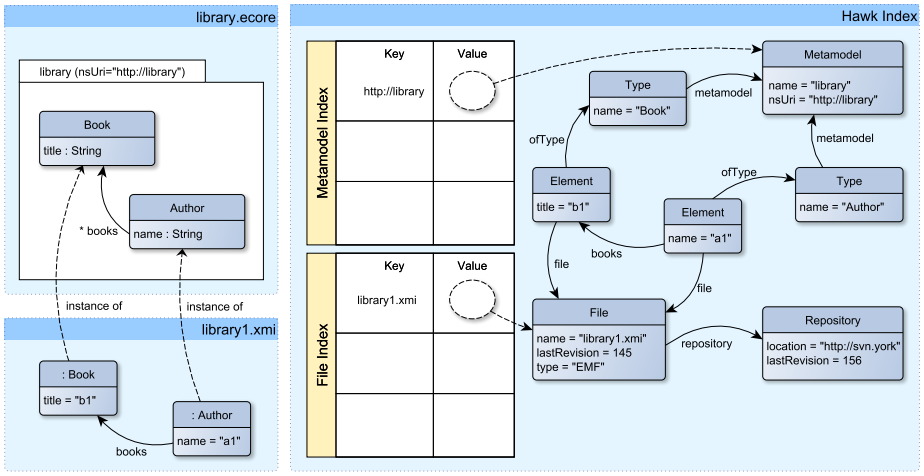

Fig. 2. High-level overview of the contents of a library model index (persisted in a NoSQL graph database), adapted from [3].

Overview of a Hawk model index An example of a Hawk index, containing a simple library metamodel and a model that conforms to it, is illustrated in Figure 2. In general, a model index typically contains the following entities [3]:

- **Repository nodes**. These represent a VCS repository and contain its URL and last revision. They are linked with relationships to the *File*s they contain.
- **File nodes**. These represent files in a repository and contain information on the file such as the path, current revision and type. They are linked with relationships to the *Element*s they contain.
- **Metamodel nodes**. These represent metamodels and contain their names and their unique namespace URIs (in EMF, these would be *EPackage*s[4]). They are linked with relationships to the (metamodel) *Type*s they contain.
- **Type nodes**. These represent metamodel types (*EClass*es in EMF terminology) and contain their name. They are linked with relationships to their (model) *Element* instances.
- **Element nodes**. These represent model elements (*EObject*s in EMF terminology) and contain their attributes (as properties) and their references (to other model elements) as relationships to them.

[1] `https://github.com/kb634/mondo-hawk`
[2] `http://www.openbim.org/`
[3] `http://www.modelio.org/`
[4] We choose to draw parallels with concepts from EMF as they are well-understood

– **Database Indexes**. Metamodel nodes and File nodes are indexed[5] in the store, so that their nodes can be efficiently accessed for querying (commonly used as starting points for complex graph traversal queries).

It is worth noting that a model index such as the one presented above may end up being a full copy of the actual models found on the relevant version control system but it does not have to be. In principle, if some contents of the model are not deemed useful they can be omitted in order to gain an improvement in injection and possibly query time.

2.3 Updating Model Indexes

As models can evolve over time, appropriate mechanisms need to be in place for efficiently synchronizing the index with any changes in models it monitors. Two alternative approaches that can be used for this:

Naive Synchronization In this approach, when a model file changes in a repository, Hawk propagates this change by removing all its model elements in the model index and then re-inserting them.

This approach, while seeming inefficient, has various potential benefits:

– The overhead of comparing the contents of the two versions (in two different formats) is avoided. In order to perform an incremental update an element-by-element comparison of the old and new versions of the models has to be performed, which can be very costly.
– The index is expected to be capable of performing massive naive inserts efficiently (such "mass inserts" are used by various database technologies in order to rapidly store an initial version of the data, they may make the model index unavailable during this process). When performing changes to elements in the index (such as when performing an incremental update), this cannot happen, so the time for each individual change to be propagated can substantially increase.

The drawbacks of this approach are:

– The lack of knowledge about changes means that even a change in a single element will require all elements in the model file to be removed and re-inserted into the index. This gets more costly the smaller the change and the larger the model file gets.
– The act of performing a "mass insert" into the index will require heavy usage of its resources and may also limit its availability for use while the update happens. Furthermore, if the index is inefficient in deleting elements, then deleting such large contents may become a bottleneck.

Incremental Synchronization In this approach, model changes are identified on a model element level by performing a comparison of the old version found in the index with the new version found in the repository. As such, only affected elements (added, removed, changed) have to be updated in the index for achieving synchronization.

[5] `http://components.neo4j.org/neo4j-lucene-index/snapshot/`

The benefits of this approach are:

- As each change is identified on a model element level, avoiding having to delete and insert the entire model file into the index can potentially compensate for the overhead of performing a full model comparison. Hence it can be more efficient both in terms of time as well as resources and availability, as only the relevant subsets of the index will be touched.

The drawbacks of this approach are:

- The overhead of model comparison may be larger than the gain of a fine-grained update if the update is relatively large when compared to the model.
- Such updates impose various requirements on the model files, in order to enable the required comparison (such as the need for model elements to have a unique (non-volatile per file) identifier), which may not be satisfied by some model representation formats.

3 Updates in Hawk

This section presents the process used for updating Hawk model indexes when monitored models change, a combination of naive insertion and incremental updates (using model element signatures – described below), and discusses how this improves the efficiency of updates performed in large model indexes.

3.1 Overview of Hawk Updates

Hawk performs Algorithm 1 every time it finds a changed (added, removed, updated) model file from any of its monitored repositories. As demonstrated in the sequel, using this incremental updating when a model file is already indexed provides a large performance gain when compared to naively deleting and re-indexing it every time it is modified.

```
if model file already indexed then
|  if change of type added/updated then
|  |  incremental update (see Section 3.4)
|  else if change of type removed then
|  |  delete indexed elements of file, keeping cross-file references to these
|  |    elements as proxies
|  end
else
|  if change of type added/updated then
|  |  naive insertion (see Section 3.3)
|  end
end
```

Algorithm 1: Hawk Update Overview

3.2 Signature Calculation

In order to update a model, an efficient way to determine whether a model element has changed is needed. A signature of a model element is a lightweight proxy to its current state. In order to calculate a meaningful signature for model elements indexed in Hawk (in order to enable support for incremental updates of the model index, as models in it evolve), every mutable feature of the element needs to be accounted for. As such, the following features are used to calculate the signature of each element:

- all of the names and values of its attributes
- all of the names and the IDs (of the target elements) of its references

This works under the assumption that model elements cannot be re-typed during model evolution, which is the case for the popular modeling technologies such as EMF, as well as that model elements have immutable and unique IDs.

A signature can be represented as either a String containing the concatenation of the values listed above or as a hash-code of this String. The String representation ensures that a unique signature exists for any model state, but suffers in terms of comparison performance as potentially very long Strings will have to be compared. On the other hand, the hash-code (Integer) representation allows for rapid comparisons but has a chance (albeit small) for clashes, which show different model states as having the same signature. We decided to use the integer representation as this identifier, to allow rapid comparisons. This signature is used to efficiently find changes in model elements, as detailed below.

3.3 Naive Insertion

For a naive insertion of a model file into Hawk a process outlined in Algorithm 2 is followed. In this process, the elements of the model file are firstly loaded into memory as a *model resource*. Then, for each such element a node is created in the index graph with its attributes as properties, and linked (using relationships) to its file and type/supertypes. Finally, for each element its references are used to link the node with other nodes in the graph.

As this process often requires intense resource consumption, the batch mode of Hawk's back-end is used (if the specific back-end used supports it). This mode makes the database unavailable until the process is completed, but Neo4J has at least an order of magnitude better performance in terms of execution time when compared to on-line (transactional) updating.

3.4 Incremental Updating

For incremental updating of a model file into Hawk, the process outlined in Algorithm 3 is followed. In this process the signatures of each element are used to efficiently determine which elements have changed. Then, for each new element a node is created, for each changed element its properties and relevant references are updated (keeping dangling cross-file references as proxies in Hawk for consistency), and for each removed element its node is deleted. The complexity of this

```
use relevant factory to parse the file into a model resource
foreach element in the model resource do
    create model element node in graph
    create signature attribute in node
    create a relationship from this node to its file node
    create a relationship from this node to its type node (and relationships to
     all its supertype nodes)
end
foreach element in the model resource do
    foreach reference in the references of the element do
        if reference of element is set then
            foreach referenced element do
                if referenced element is not a proxy then
                    create relationship from this node to the node of the
                    referenced element
                else
                    add new proxy reference
                end
            end
        end
    end
end
```

Algorithm 2: Naive (batch) insertion algorithm

algorithm is $\mathcal{O}(m + n + d \times r)$ where m is the number of model elements in the updated model file, n is the number of model nodes in the model index linked to the (previous version of the) updated file, d is the number of changed elements and r is the number of target elements referenced by the changed element.

This process only alters the part of the model index which has actually changed and as such, it does not need to use more resources than required by the magnitude of the change, potentially saving on memory and execution time.

3.5 Derived Attributes

Derived attributes are used in Hawk in order to speed up certain types of queries [2]. Such attributes are computed using expressions formed in the expression language of a known *Query Engine*. A query engine in Hawk allows for expression languages (such as OCL or Epsilon's EOL [10]) to be used as a query mechanism for a Hawk model index. Such derived attributes are hence pre-computed and cached at indexing time and need to be maintained as the model index evolves.

A simple example is shown in Figure 3; here, the number of books each author has published (named *numberOfBooks*) is pre-computed (using the EOL expression *return self.books.size()*) and stored in a new *DerivedAttribute* node[6] with the attribute name as the relationship linking it to its parent *Element* node. This derived attribute is handled seamlessly with regards to querying, hence an EOL query used to get the number of books of a specific author a would change from *a.books.size()* to *a.numberOfBooks* (in both cases returning an integer).

Expressions of arbitrary complexity are expected to be used in practice so that pre-caching the results of such expressions is actually worthwhile. A more

[6] A new node is used for overcoming a limitation found during incremental updating of derived attributes; further information on this can be found in Section 3.6

```
let nodes be the set containing the ids and pointers to all the nodes (in the model
  index – linked with the updated file)
let signatures be the set containing the ids and signatures of the nodes
let delta be the set containing changed elements
let added be the set containing new elements (to be added to the model index)
let unchanged be the set containing elements which are the same
foreach node from all nodes in the model index that are linked with the updated file do
    add node to nodes
    add signature of node to signatures
end
foreach element in elements of model resource do
    if element id exists in signatures then
        if element signature not equal to current signature then
            add element to delta
        else
            add element to unchanged
        end
    else
        add element to added
    end
end
/* add new nodes to model index                                                  */
foreach element in added do
    add this new element in model file to model index
end
/* delete obsolete nodes and change altered node attributes                      */
foreach node in nodes do
    if node id exists in delta then
        remove current properties of node
        set all model attributes of node as properties
    else if node id does not exist in unchanged then
        de-reference node (keeping dangling cross-file references as proxies)
        delete node
    end
end
/* change altered references                                                     */
foreach element in delta do
    foreach reference in references of element do
        if reference is set then
            foreach referenced element in referenced elements of reference do
                if referenced element is not proxy then
                    add id of referenced element to targetIds
                else
                    add new proxy reference to model index
                end
            end
            foreach relationship in relationships of node linked with the element do
                if relationship target has id which exists in targetIds then
                    remove target from targetIds
                else
                    delete relationship as new model does not have it
                end
            end
            foreach id in targetIds do
                add new relationship to model index
            end
        else
            foreach relationship in relationships of node, with the same name as the
            reference name do
                delete this relationship
            end
        end
    end
end
```

Algorithm 3: Incremental update algorithm

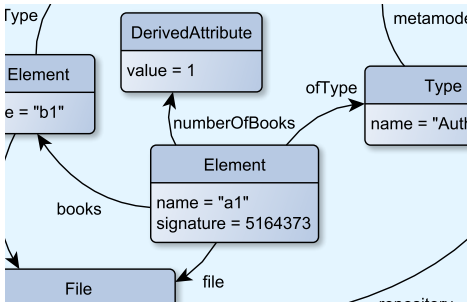

Fig. 3. Pre-computing the number of books of each author

realistic example (but one too complex to present in detail) would be to calculate (for each author) the names of the authors which have co-written at least three books with the author in question. This query can be presented in EOL as:

```
var coauthors = self.books.collect(a|a.authors.name);
var authormap = coauthors.flatten.excluding(self.name).asSet().mapBy(a|
    coauthors.select(s|s.contains(a)).size());
var atLeastThreeBooks : Sequence;
for(a in authormap.keySet()){
  if(a>=3) atLeastThreeBooks.add(authormap.get(a)); }
return atLeastThreeBooks.flatten();
```

This would return a Sequence of names of the other authors in question. Caching such complex expressions during inserts/updates can significantly reduce query execution time of relevant queries, as shown in [2].

3.6 Updating of Derived Attributes

A naive approach for maintaining such attributes would involve having to fully re-compute each one, every time any change happens to the model index. This is due to the fact that any such attribute can potentially depend upon any model element in the index, thus any change can potentially affect any derived attribute. Such an approach would be extremely inefficient and resource consuming.

As such, an incremental approach for updating derived attributes in Hawk has been used. In this approach, which is an adaptation of the incremental OCL evaluation approach discussed in [5], only attributes affected by a change made to the model index are re-computed when an update happens. In order to know which elements affect which derived attributes, the scope of a derived attribute needs to be calculated. The scope of a derived attribute comprises the current model elements (and/or features) in the model index this attribute needs to access in order to be calculated. When a derived attribute is added/updated in the model index, the query engine used to calculate this attribute also publishes an *AccessListener* to Hawk, providing the collection of *Accesses* this attribute performed. By recording these accesses (element and/or feature accesses), Hawk updates only the derived attributes which access an element altered during an incremental update. As the incremental update changes the minimal number of

elements during model evolution, the updating of derived attributes can be seen to be as efficient as possible with respect to the magnitude of the change.

In more detail, every time an update process happens in Hawk, it records the changes it has made to the model index. A change can be one of the following:

- A model element has been created / deleted
- A property of a model element has been altered
- A reference of a model element (to another one) has been created / deleted
- Note: complex changes (like *move*) are broken down to these simple changes.

Furthermore, every time a derived attribute is added or updated, it records the accesses it requires in order to be computed. An access can be one of:

- Access to a property / reference of a model element
- Access to the collection of model elements of a specific *type / kind*

By having recorded the above mentioned changes and accesses, Hawk can calculate which derived attributes need to be re-computed during a model update using Algorithm 4. As the derived attribute is a node itself, it can be directly referenced and updated if necessary; if the derived attribute was located inside its parent *Element* node, that node would have to be referenced instead and hence all derived attributes in it would have to be updated, as there would not be a way to distinguish which ones need updating and which ones do not.

In the example above, for the derived attribute *numberOfBooks* of node *a1*, the access would read as follows: The derived attribute *numberOfBooks* needs to access node *a1* for its feature *books*. Hence anytime the feature *books* changes for node *a1* (ie: if a member of this reference is added or removed), the derived attribute *numberOfBooks* will have to be recomputed (and only then). As demonstrated by [6], this approach works for expressions of arbitrary complexity as long as they are deterministic (they do not introduce any randomness using random number generators, hash-maps or other genuinely unordered collections). As EOL defaults to using Sequences for collections and does not inherently use random number generators, as long as the expressions provided do not specifically introduce non-determinism, this approach is sound [6].

```
let nodesToBeUpdated be the set containing the derived attribute nodes which
  will have to be updated – initially empty
foreach change in the collection of changes do
    if the change is a model element change then
        add any derived attribute which accesses this element (or any of its
          structural features) to nodesToBeUpdated
    else if the change is a structural feature change then
        add any derived attribute which accesses this structural feature to
          nodesToBeUpdated
    end
end
foreach node in nodesToBeUpdated do
    re-compute the value of the (derived attribute) node
    update the accesses to the new elements/features this node now requires
end
```

Algorithm 4: Derived attribute incremental update algorithm

4 Evaluation

In this section, an existing large-scale benchmark is used to conduct performance tests for updating a Hawk model index. The sample models are mutated in order to simulate changes that are picked up by Hawk.

4.1 The GraBaTs 2009 Case Study

For evaluating query execution performance in Hawk we use large-scale models extracted by reverse engineering existing Java code. The updated version of the JDTAST metamodel used in the *SharenGo Java Legacy Reverse-Engineering* MoDisco use case[7], presented in the GraBaTs 2009 contest [7] described below, as well as the five models provided in the contest, are used for this purpose. In JDTAST *TypeDeclaration*s are used to define Java classes and interfaces, *MethodDeclaration*s are used to define methods and *Modifier*s are used to define modifiers (e.g static, synchronized) for Java classes and methods. Figures of the relevant subset of the JDTAST metamodel are found in works like [2, 12].

The GraBaTs 2009 contest provided five models, Set0–Set4 (of progressively larger models, with 70,447, 198,466, 2,082,841, 4,852,855 and 4,961,779 model elements, respectively), conforming to the JDTAST metamodel. These models are injected into Hawk and then mutated using various heuristics in order to test and evaluate its update procedure. In the following sections we use this case study as a running example to illustrate the implementation and to evaluate it.

4.2 Execution Environment

Performance figures that have been measured on a PC with Intel(R) Core(TM) i5-4670K CPU @ 3.40GHz, with 32GB of physical memory, a Solid State Drive (SSD) hard disk, and running the Windows 7 (64 bits) operating system are presented. The Java Virtual Machine (JVM) version 1.8.0_20-b26 has been restarted for each measure as well as for each repetition of each measure. In each case, 20GB of RAM has been allocated to the JVM (which includes any virtual memory used by the embedded Neo4J database server running the tests).

4.3 Model Manipulation

In order to perform model manipulation operations, we used Epsilon's EOL language [10]. EOL is an imperative OCL dialect which supports model modification. We decided to perform five model mutations (changes), which are representative of modifications performed in Java code. These mutations are performed by five EOL operations (available online[8]). By using these operations in an EOL script we can change the model it is run on in a realistic[9] yet sufficiently random manner[10].

[7] `http://www.eclipse.org/gmt/MoDisco/useCases/JavaLegacyRE/`

[8] `https://github.com/kb634/mondo-hawk/blob/master/model_manipulations.eol`

[9] Operations often used in manipulation of Java code, such as deleting a Java class

[10] For example, by randomizing which Java class is deleted each time

4.4 Model Update Execution Time

Table 1 shows the average time taken to complete an update for the models produced by performing the model mutations presented in Section 4.3 on the original GraBaTs models. M(INS) represents the initial insert of the original GraBaTs models into an empty Hawk index (using the naive insert process) and M(0%)–M(50%) represent the update time (from the original model) to one containing 0% to 50% content mutations. These mutations contain an equal degree of each mutation operation found in Section 4.3 so that the total change to the model ends up being N% of the original model contents. As such, each of the five mutation operations performs changes equal to $\frac{N}{5}$% of the original model elements; since some changes are addition/removal operations on model elements, the size of the resulting model is not the same as that of the original.

Table 1. Update Execution Time Results

Mutation	**Execution Time (in seconds)**									
	Set0		**Set1**		**Set2**		**Set3**		**Set4**	
	Naive	Inc.	Naive	Inc.	Naive	Inc.	Naive	Inc.	Naive	Inc.
M(INS)	9.96	n/a	18.69	n/a	118.19	n/a	291.06	n/a	346.46	n/a
M(0%)	16.61	2.70	45.72	6.07	-	63.96	-	162.52	-	224.85
M(10%)	16.82	3.94	47.71	10.45	-	94.59	-	247.94	-	292.86
M(20%)	17.76	4.71	48.22	11.53	-	115.86	-	364.94	-	417.50
M(30%)	18.93	5.66	50.60	15.04	-	145.56	-	440.78	-	622.51
M(40%)	21.84	7.04	54.73	18.79	-	165.48	-	781.35	-	-
M(50%)	22.09	7.97	60.21	20.92	-	193.41	-	-	-	-

For each case both the incremental and naive updates were tested and compared with one another. The naive update follows the process described in the prequel for naive insertion, after having had the currently indexed elements deleted from the index. As the naive update process failed to terminate for the larger sets (Set2–Set4), figures for these models are not presented for the naive update process. The reason for this failure is that the Neo4J back-end runs out of memory when trying to delete the entire contents of the model index. This is an unforeseen limitation in the Neo4J database, as we require of it to perform a single transaction to delete the entire contents (as it does not support nested transactions but only flattened nested transactions, which only commit when the top-level transaction is closed) in order to maintain consistency between model versions. We also note that the incremental update fails to complete for 50% of Set3, 40% of Set4 and 50% of Set4. This is due to the fact that the magnitude of the change is so large that not enough memory is available for Neo4J to be able to fit this change in a transaction. The aim is to test the limits of Hawk, as such a system typically aims at collecting a large amount of fragmented models and not large monolithic ones; in the former case memory would not be an issue as

it can be flushed after each file is updated. Furthermore, a 40% or 50% change on a model with millions of elements is not an expected use-case and again is presented to test the limits of the system.

These results suggest that the incremental update process is substantially faster than the naive approach, while also not compromising availability of the index[11]. This can be largely attributed to there being no support for "mass deletes" in the index, which ends up taking the majority of time needed for a naive update. The actual time taken for the incremental updates is promising as it scales linearly with the magnitude of the change in the model, giving us improvements of up to 78.10% decrease in execution time for a 10% model change and up to 65.25% for a 50% model change, averaging a 70.7% decrease in execution time over all of the comparable results[12].

4.5 Derived Attribute Update Execution Time

Results for the execution time of altering derived attributes are not presented as they would have to be compared to a baseline. Such a baseline would have been to use a naive approach whereby all derived attributes in the model index would have to be updated any time any model element or feature gets updated. As this approach would have been inefficient compared to the incremental one, it was never implemented so a meaningful comparison cannot be made.

4.6 Threats to Validity

There are five observed threats to the validity of this approach:

- The model mutations performed may have influenced the results. We tried to limit this by performing multiple mutations in each case, all of which contain a random factor in them.
- The percentage change of each model may not be indicative of real model change. We tried to limit this by exploring a large variety of changes ranging from zero to fifty percent of the original model.
- The model sizes used for empirical evaluation may not be indicative. Hawk aims at handling large fragmented models, thus we anticipate that the size of each fragment will not be orders of magnitude greater than the test models.
- Using an integer representation for the signatures has a chance for collisions; this chance tends to 1 in 4.29 billion for non-trivial Strings. In all of the empirical tests performed no clashes have been observed, which gives us some confidence that the approach should be used for performance reasons.
- The last one is regarding the correctness of the incremental algorithm. While this is not formally proven, empirical tests comparing the index state after an incremental update with that of the original naive update, previously used in Hawk (for the same changes), provided the same results for all of the mutated models where both the incremental and naive updates completed.

[11] As it does not block any incoming queries which may need to be performed

[12] The 10 results from set0 and set1 that both naive and incremental approaches completed, disregarding the 0% change values as they are presented as a baseline.

5 Related Work

Aiming at tackling versioned collaborative development of models, proprietary model repositories such as MagicDraw's TeamServer[13] have been developed; they allow for model-element-level versioning, comparison and querying and support multiple concurrent users. Nevertheless, such systems are highly-coupled with the respective vendors' modelling tools and hence have limited flexibility as they bind the user to a specific technology.

Similarly, open-source model repositories such as CDO[14] and EMFStore [8] have arguably gained little traction while commonly supporting a wide variety of back-end technologies. In our view, there are several valid reasons for this. From a practitioner's point of view, choosing a model-specific version control system supported by a small open-source community over a robust and widely-used and supported file-based version control system for storing business-critical models is not a straightforward decision. Also, using two version control systems in parallel (e.g. Git for code and CDO for models) can introduce fragmentation as models and code changed in the context of the same conceptual *commit*, will need to be manually distributed over two unconnected version control systems.

Various model persistence mechanisms have been developed in the past few years as a scalable alternative to the XMI file-based model persistence used in popular modeling technologies such as EMF. Many of these, such as NeoEMF [4], Morsa [12], MongoEMF[15] and EMF fragments [13] use NoSQL databases like Neo4J or MongoDB as a back-end and deliver promising results with respect to model traversal and querying. On the other hand such systems do not handle version control of models stored in them.

6 Conclusions and Further Work

In this work we presented an incremental approach to updating model indexes, using lightweight model element signatures. From the empirical data collected we can conclude that incremental updates seem to outperform a naive approach to achieving synchronization in model indexes. As availability can be important in this context, the fact that the solution which does not compromise availability is the most performant is noteworthy. We also discussed an incremental approach for updating derived attributes which uses model changes and accesses to only update derived attributes affected by a model change.

Obtaining these results motivates us to further this work by investigating the use of derived references in model indexes, by providing better support for metamodel evolution, and by providing support for scoping queries to limit results to elements found in specific files in an efficient manner.

[13] `http://www.nomagic.com/products/teamwork-server.html`
[14] `http://www.eclipse.org/cdo/documentation/index.php`
[15] `https://github.com/BryanHunt/mongo-emf/`

Acknowledgments This research was part supported by the EPSRC, through the Large-Scale Complex IT Systems project (EP/F001096/1) and by the EU, through the MONDO FP7 STREP project (#611125).

References

1. Barmpis, K., Kolovos, D.S.: Hawk: towards a scalable model indexing architecture. In: Proceedings of the Workshop on Scalability in Model Driven Engineering. pp. 6:1–6:9. BigMDE '13, ACM, New York, NY, USA (June 2013)
2. Barmpis, K., Kolovos, D.S.: Evaluation of contemporary graph databases for efficient persistence of large-scale models. Journal of Object Technology 13-3, 3:1–26 (July 2014)
3. Barmpis, K., Kolovos, D.S.: Towards scalable querying of large-scale models. In: Proceedings of the 10th European Conference on Modelling Foundations and Applications. ECMFA'14 (July 2014)
4. Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., Launay, D.: Neo4emf, a scalable persistence layer for emf models. In: Modelling Foundations and Applications, pp. 230–241. Springer (2014)
5. Egyed, A.: Instant consistency checking for the uml. In: Proc. of the 28th International Conference on Software Engineering. pp. 381–390. ICSE '06, ACM (2006)
6. Egyed, A.: Automatically detecting and tracking inconsistencies in software design models. Software Engineering, IEEE Transactions on 37(2), 188–204 (2011)
7. GraBaTs: 5th Int. Workshop on Graph-Based Tools (2009), `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/`
8. Koegel, M., Helming, J.: Emfstore: a model repository for emf models. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. pp. 307–308. ACM (2010)
9. Kolovos, D.S., Paige, R.F., Polack, F.A.: Scalability: The Holy Grail of Model Driven Engineering. In: Proc. Workshop on Challenges in MDE, collocated with MoDELS '08, Toulouse, France (2008)
10. Kolovos, D.S., Paige, R.F. and Polack, F.A.: The Epsilon Object Language. In: Proc. European Conference in Model Driven Architecture (EC-MDA) 2006. LNCS, vol. 4066, pp. 128–142. Bilbao, Spain (July 2006)
11. Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform Random Generation of Huge Metamodel Instances. In: Proceedings of ECMDA-FA '09. pp. 130–145. Springer-Verlag, Berlin, Heidelberg (2009)
12. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: A repository for scalable model management. Software & Systems Modeling pp. 1–21 (2013)
13. Scheidgen, M., Zubow, A.: Map/reduce on emf models. In: Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and Cloud Computing. pp. 7:1–7:5. MDHPCL '12 (2012)
14. Shah, S.M., Wei, R., Kolovos, D.S., Rose, L.M., Paige, R.F., Barmpis, K.: A framework to benchmark nosql data stores for large-scale model persistence. In: Proc. 15th Conf. on Model-Driven Engineering Lang. and Systems, Models'14 (2014)