Towards Scalable Querying of Large-Scale Models

Konstantinos Barmpis and Dimitrios S. Kolovos

Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK
{kb,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. Due to the aggregate size of indexed models, providing a reliable, usable, and fast mechanism for querying Hawk's index is essential. This paper presents the integration of Hawk with an existing model querying language, discusses the efficiency challenges faced, and presents an approach based on the use of derived features and indexes as a means of improving the performance of particular classes of queries. The paper also reports on the evaluation of a prototype that implements the proposed approach against the Grabats benchmark query, focusing on the observed efficiency benefits in terms of query execution time. It also compares the size and resource use of the model index against one created without using such optimizations.

Keywords: Scalability, model querying, model-driven engineering.

1 Introduction

The popularity and adoption of MDE in industry has increased substantially in the past decade as it provides several benefits compared to traditional software engineering practices, such as improved productivity and reuse [1], which allow for systems to be built faster and cheaper. However, certain limitations of supporting tools such as poor scalability which prevent wider use of MDE in industry [2,3] will need to be overcome. Scalability issues arise when large models (of the order of millions of model elements) are used in MDE processes.

When referring to scalability issues in MDE they can be split into the following categories [4]:

1. Model persistence: storage of large models; ability to access and update such models with low memory footprint and fast execution time.
2. Model querying and transformation: ability to perform intensive and complex queries and transformations on large models with fast execution time.
3. Collaborative work: multiple developers being able to query, modify and version control large-scale shared models in a non-invasive manner.

This paper contributes to the study of scalable techniques for large-scale model persistence and querying by presenting the use of derived attributes to substantially improve the efficiency of certain types of model queries, and reporting on

J. Cabot and J. Rubin (Eds.): ECMFA 2014, LNCS 8569, pp. 35–50, 2014.

the results obtained by exploring the integration of the Hawk [5] and Epsilon [6] frameworks that have been used to implement this. This paper builds upon [5] by discussing the implementation of the query layer the tool provides. In this work we assume that the reader is familiar with the organization of 3-level metamodeling architectures such as MOF/EMF.

The remainder of the paper is organized as follows. Section 2, introduces model version control, Hawk and model indexing. Section 3 presents Hawk's query layer and discusses how it can be optimized by use of derived attributes in the store. Section 4 presents the prototype implementation of the integration of Hawk with the Epsilon platform for providing a general-purpose query layer. In Section 5 this prototype integration is evaluated using variations of the Grabats benchmark, in order to test its performance. Finally, Section 6 discusses the application of these results and identifies interesting directions for further work in this area.

2 Background

This section briefly introduces version control in the context of MDE, provides an overview of Hawk and discusses querying, providing an overview of the various forms available today that have motivated the work presented here.

2.1 Model Version Control

To tackle the challenge of collaborative development and version control of large models, model-specific repositories and version control systems (such as CDO[1] and ModelCVS) have been proposed. The main advantages of such systems is that they provide support for synchronous collaboration, on-demand loading and locking of model fragments, and global server-side queries on models. On the downside, such repositories are typically proprietary, re-implement similar functionality (user management, model fragment locking/unlocking, check-in/out), and lack in features such as branching and tagging. Moreover, such repositories need to be administered (e.g. backed up) separately, and there is limited tool support for them outside the environment for which they were initially developed for (e.g. integration with other IDEs, continuous integration systems, and other 3rd-party model measurement and analysis tools). Finally, they arguably lack in robustness compared to file-based version control systems such as Subversion and Git.

As such, switching from a file-based to a model-specific version control system can require a significant leap of faith, which can become even more challenging if the models in question are of significant business value. On the other hand, in order to perform meaningful queries on models stored in a file-based version control system (e.g. to identify cross-references between model files or to search for model elements with particular properties across the entire repository), these models need to be first checked out in the developer's workspace and loaded into memory. This can be tedious, or even impossible, for large-scale models.

[1] `http://www.eclipse.org/emf/cdo`

2.2 Hawk

The limitations identified on both sides of the spectrum have motivated us to design and implement a framework (Hawk) that enables developers to perform queries on models stored in established file-based version control systems, without needing to maintain a complete copy of them in their local workspace. To achieve this, Hawk acts as a middle-man that creates and maintains indexes of models stored in remote file-based version control repositories; a *model index*[2] is a persisted form of a collection of (potentially interconnected) models, and its aim is to provide support for efficient querying of these models at a model element granularity. As discussed in [5], in our view, this provides an orthogonal approach for addressing the scalability concern that does not interfere with the current state of practice.

This section briefly describes the architecture, design, and prototype implementation of Hawk to provide context for how it is used for indexing large models and consequently to efficiently query such model indexes.

System Architecture and Design. Hawk aims at delivering a system capable of working 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 query it. It needs to be scalable so that it can accommodate large sets of models, and non-invasive (the VCS repositories should not need to be modified or configured).

Hawk comprises components which monitor a set of version control systems, parse and index relevant models stored in them. For details on supported version control systems, model formats, index persistence back-ends as well as additional components of Hawk readers can refer to [5].

Overview of a Hawk Model Index. Based on results obtained through extensive benchmarking [4], we have decided to use a NoSQL graph database (Neo4J[3]) for persisting model indexes. An example of such an index, containing a simple library metamodel and a model that conforms to it, is illustrated in Figure 1. In general, a model index typically contains the following entities:

– **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 or *Metamodel*s they contain.
– **Metamodel Nodes.** These represent metamodels and contain their names and their unique namespace URIs (in EMF, these would be *EPackages*[4]). They are linked with relationships to the (metamodel) *Type*s they contain.

[2] This should not to be confused with a *database index* provided by many SQL and NoSQL databases.

[3] `http://neo4j.org/`

[4] We choose to draw parallels with concepts from EMF as they are well-understood and unambiguous.

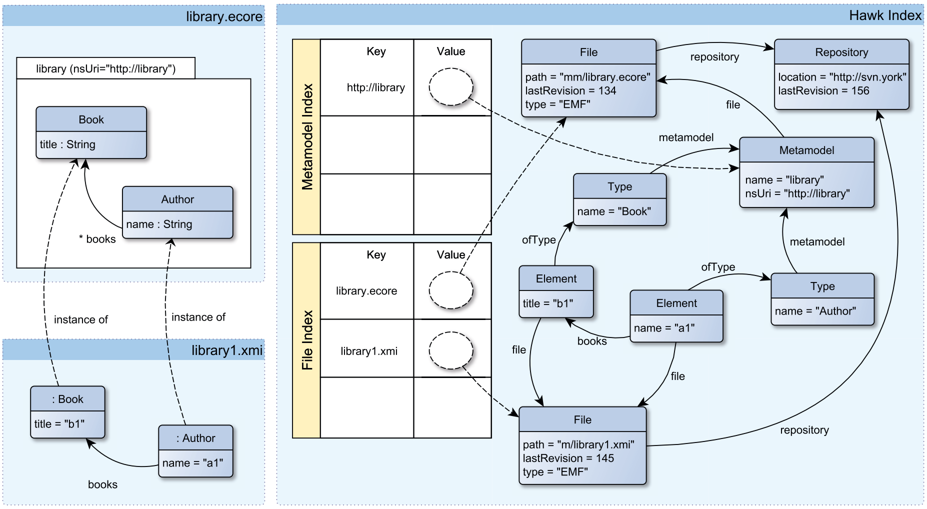

Fig. 1. High-level overview of the contents of a library model index (persisted in a NoSQL graph database)

- **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 can contain their attributes (as properties) and their references (to other model elements) as relationships to them.
- **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 fully 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 Querying of Model Indexes

To be of practical value, a model indexing framework such as Hawk needs to be able to provide correct and efficient responses to queries made on its indexes. There are two principal ways of querying a model index:

Native Querying. The most straightforward, and often the most performant, way of querying an index is using the native API of its persistence back-end. In a model stored in a database the API provided by the tool providing the driver

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

used to persist said model would be used with a relevant query language (such as SQL statements if a relational database is used or Cypher if a Neo4J NoSQL database is used), or using direct API calls in a programing language such as Java. Nevertheless, it also demonstrates certain shortcomings which should be considered:

- *Query Conciseness.* Native queries can be particularly verbose and, consequently, difficult to write, understand and maintain. An example of this can be found in Section 6.1 of [4].
- *Query Abstraction Level.* Native queries are bound to the specific technology used; they have to be engineered for that technology and cannot be used for a different back-end without substantial alteration in most cases.

Back-End Independent Navigation and Querying. An alternative way to access and query models is through higher-level query languages that are independent of the persistence mechanism. Examples of such languages include the Object Constraint Language (OCL), the Epsilon Object Language (EOL) [7] (from the Epsilon [8] platform) and the Atlas Transformation Language (ATL), which abstract over concrete model representation and persistence technologies using intermediate layers such as the *OCL pivot metamodel* [9] and *Epsilon Model Connectivity* [6] layer.

In terms of execution, queries expressed in such high-level languages can be executed on an in-memory representation of the model, or translated into queries expressed in persistence-level query languages such as SQL and XQuery[6], at compile-time or at run-time. Full translation is only feasible in cases where the high-level and the lower-level query languages are isomorphic in terms of capabilities. This is not always the case: for example, EOL supports dynamic dispatch which is not supported in SQL. Even when full compile-time translation is not feasible, partial translation at run-time has been shown to deliver significant performance improvements as seen in [10].

3 Scalable Model Index Querying

This section will use the library example seen in Figure 1 as a running example and will discuss how derived attributes can be used to improve the performance of queries made on Hawk model indexes. The principal aim of this work is to present how using such derived attributes can greatly improve performance of relevant queries made on such model indexes and to provide incentive for building a complete framework for supporting them in Hawk.

3.1 Querying a Model Index

Regardless of the use of native or back-end independent querying, in order to respond to a query (from now on referred to as the *library query*) requesting the authors that have more than N books in the example index, the following steps would have to occur:

[6] http://journal.ub.tu-berlin.de/eceasst/article/viewFile/108/103

1. The starting point of the query would have to be found. In this case, the collection of all instances of *Author* in the model would have to be retrieved.
2. For each author node, the number of the "books" relationships of the node identified in step 2 would need to be counted and compared against N.

Step 1 is easy to perform in Hawk as an index of *Metamodels* is kept which can be used to rapidly provide a starting point for a query which requires elements of a specific type (such as *Author* instances for example). If a query uses the whole model index as a starting point then there is no optimization to be performed as the entire model index would have to be traversed in order to find the Node representing the *Author* type. Step 2 where we can begin optimizing to improve the execution time of queries which have to iterate (possibly on multiple levels) to find a result.

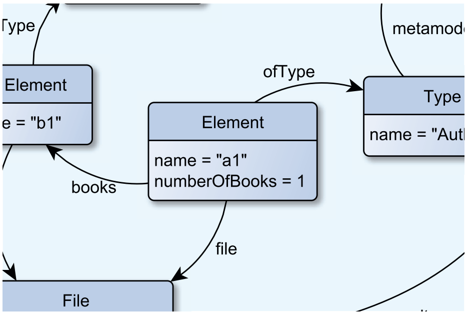

Fig. 2. Pre-computing the number of books of each author

An effective way to increase query efficiency is to pre-compute and cache – at indexing time – information that can be used to speed up particular queries of interest. Using the library example, we can store the total number of books of each author under a new, derived, 'numberOfBooks' attribute attached to each author, as shown in Figure 2. By pre-computing and caching this information, the query above can be rewritten so that it does not have to iterate though all the books of each author, but instead it can directly compare N against the value of its (derived) 'numberOfBooks' property.

3.2 Adding Derived Attributes

Our aim in this work is to explore the impact that such derived attributes can have on the performance of queries on large model indexes. As such, we have opted for a minimal approach for defining derived attributes and their derivation logic. In our current prototype, we need to create a derived attribute on the relevant *EClass* (i.e. a derived integer attribute 'numberOfBooks' on the *EClass Author*) and annotate it as 'HawkDerivedFeature'. As illustrated in Figure 3, the derivation logic is specified using an OCL-like (EOL in our prototype) expression in the details of the annotation. Such attributes are currently created manually

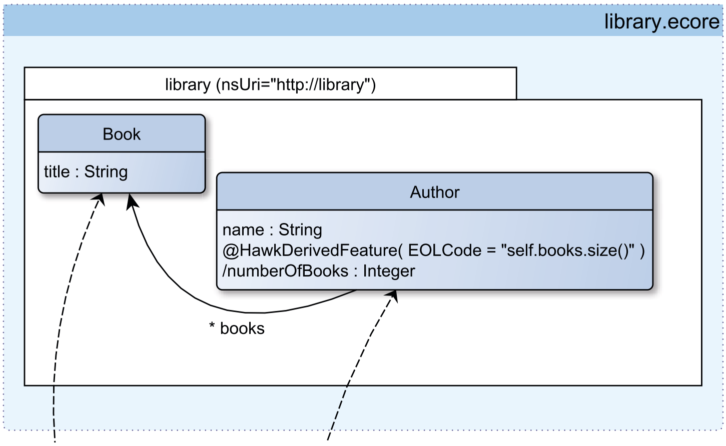
library.ecore
library (nsUri="http://library")
Book
title : String
Author
name : String
@HawkDerivedFeature(EOLCode = "self.books.size()")
/numberOfBooks : Integer
* books

Fig. 3. Defining the *numberOfBooks* derived attribute

by the user and if they cannot be resolved a simple error value is produced in the index.

Since our focus is only on evaluating the performance improvements delivered, several interesting engineering problems that would have to be addressed by a usable system have been intentionally ignored:

- How to enable the declaration of derived attributes when using an immutable metamodel (e.g. UML);
- How to efficiently handle non-parsable expressions (on the expression language level) or expressions failing on a model element basis (but parsable);
- How to allow parsers from other expression languages to be easily integrated with the framework;
- How to efficiently deal with metamodel evolution, specifically how to handle types of changes such as only evolving the annotations, evolving some of the metamodel elements themselves but retaining the same annotations, evolving both the metamodel elements and the annotations at the same time.

The following section discusses how we evaluate the derived attribute value computation expressions and how we then use the computed values to enhance the performance of queries in our prototype.

4 Implementation

Before discussing the derived attribute computation and caching process, this section introduces Epsilon and its Model Connectivity Layer (EMC). It then discusses implementation details of Hawk's query layer integration with Epsilon.

4.1 Epsilon

The Epsilon platform [8] is an extensible family of languages for common model management tasks and includes tailored languages for tasks such as model-to-text transformation (EGL), model-to-model transformation (ETL), model refactoring (EWL), comparison (ECL), validation (EVL), migration (Flock), merging (EML) and pattern matching (EPL). All task-specific languages in Epsilon build on top of a core expression language – the Epsilon Object Language (EOL) – to eliminate duplication and enhance consistency.

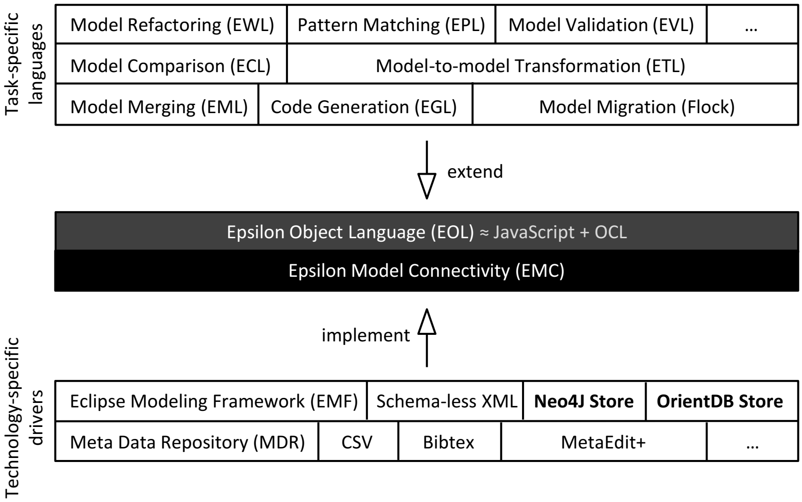

Fig. 4. The Epsilon Model Connectivity Layer

As seen in Figure 4, EOL – and as such all languages that build on top of it – is not bound to a particular metamodeling architecture or model persistence technology. Instead, an intermediate layer – the Epsilon Model Connectivity layer – was introduced to allow for seamless integration of any modeling back-end.

The Epsilon Model Connectivity Layer (EMC). This layer of Epsilon uses a driver-based approach where integration with a particular modeling technology is achieved by implementing a *driver* that conforms to a Java interface (*IModel*) provided by EMC. For a more detailed discussion on EMC and the *IModel* interface, the reader can refer to Chapter 3 of [6].

4.2 Querying a Hawk Model Index Using the Epsilon Object Language

Below, we summarize the implementation of the important methods needed by an EMC driver to enable integration with Epsilon, as well as that of the derived attributes used by Hawk's driver to improve its query performance.

Table 1. Interesting methods in the IModel interface

Method	Return Type	Description
allContents()	Collection<?>	Returns a collection containing all of the nodes contained in the index in the form of *NeoIdWrapper*s
hasType(String type)	boolean	Returns whether the type *type* exists in the index by trying to find it through the *Metamodel* index of the store.
getAllOfType(String type)	Collection<?>	Returns a collection containing all of the objects of type *type* in the index by first invoking hasType(type) and, if successful, finding the type using the *Metamodel* index and then creating a collection of *NeoIdWrapper*s containing every element which has an *ofType* relationship to *type*.
getTypeOf(Object instance)	Object	Returns the type node of the element *instance* in the index by directly accessing the node *instance* (as this method is always passed a *NeoIdWrapper* as the *instance*) and navigating its *ofType* relationship to get the type node. The returned object is a *NeoIdWrapper*.
isOfType(Object instance, String type)	boolean	Returns whether the node *instance* in this model is of type *type* by first invoking hasType(type) and, if successful, invoking getTypeOf(instance) and performing a String comparison on the resulting names.
knowsAboutProperty(Object instance, String property)	boolean	Returns whether the element *instance* in this index can have the structural feature *property* by first invoking getTypeOf(instance) and then invoking the EMF method *getEStructuralFeature(type, property)*.

IModel Interface Method Implementations. In order to use Epsilon's EOL to query model indexes stored in Hawk, an implementation of the IModel interface is required. In Table 1 we present a description of various methods of interest in the IModel interface and a summary of their implementation details in Hawk. Note that any model element loaded into memory is of Java class *NeoIdWrapper*. This is a lightweight object which contains only the location of the relevant model element in the store (its 'id' value for example in a Neo4J NoSQL Graph database) as well as a reference to the Epsilon model it is part of; this object can be used to load the element's attributes and relationships on demand.

Derived Attribute Value Computation. As discussed above, in the current prototype, we use EOL expressions to describe the derived attributes to be computed. For example, to derive the feature 'numberOfBooks' on an *Author* node, we use the 'self.books.size()' expression, as shown in Figure 3. The keyword 'self' denotes the element itself and since in this case the element is an *Author* instance (as the code was in an *EAnnotation* placed on the *EClass Author*) it will successfully evaluate the expression, returning the value *1* in this case. Such EOL expressions are actually executed, after the model insertion has been completed, using Hawk's EMC driver to query the database. Empirical data on the impact this has on total insertion time can be found in Section 5.

Reverse Reference Navigation. In the spirit of EMF's eContainer() method which allows an *EObject* to get access to its container object, Hawk provides a mechanism for reverse-navigating a containment reference in order to access the container. This feature is embedded into the parser by means of prefixing the relevant reference with "revRefNav_". For example, say one has an object 'A' with a containment reference called 'contain' to an object 'B'. Then, by typing "B.revRefNav_contain" in EOL, we get as a result object A.

5 Evaluation

In this section, the Grabats metamodel and models are used to perform various performance tests on the query layer of Hawk.

5.1 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 [11] described below, as well as the five models provided in the contest, are used for this purpose.

In this metamodel, there are *TypeDeclaration*s that are used to define Java classes and interfaces, *MethodDeclaration*s that are used to define Java methods (in classes or interfaces, for example) and *Modifier*s that are used to define Java modifiers (like static or synchronized) for Java classes or Java methods. Figures of the relevant subset of the JDTAST metamodel are found in works like [4,12].

The Grabats 2009 contest comprised several tasks, including the case study used in this paper for benchmarking different model querying and pattern detection technologies. More specifically, task 1 of this case study is performed, using all of the case studies' models, set0 – set4 (which represent progressively larger models, from one with 70447 model elements (set0) to one with 4961779 model elements (set4)), all of which conform to the JDTAST metamodel.

These models are injected into Hawk for the insertion benchmark and then queried using the Grabats 2009 task 1 query (from now on referred to as the

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

Grabats query) [13]. This query requests all instances of *TypeDeclaration* elements which declare at least one *MethodDeclaration* that has static and public modifiers and has the declared type being its returning type.

In the following sections we use this case study as a running example to illustrate the Hawk implementation and evaluate the results of using this JDTAST metamodel (and models).

5.2 Execution Environment

Performance figures that have been measured on a PC with Intel(R) Core(TM) i5-2300 CPU @ 2.80GHz, with 8GB of physical memory, and running the Windows 7 (64 bits) operating system are presented. The Java Virtual Machine (JVM) version 1.7.0_45-b18 has been restarted for each measure as well as for each repetition of each measure. In each case, 6GB of RAM has been allocated to the JVM (which includes any virtual memory used by the embedded Neo4J database server running the tests). **Results are in seconds and Megabytes, where appropriate**.

5.3 Model Insertion

Tables 2 and 3 show the results for the insertion of the various Grabats XMI models into Neo4J using three variants of the metamodel (derivation strategies):

Table 2. Model Insertion, Size Results

Model	**Size (in Mb)**		
	Original	DerivedMethodDeclaration	DerivedTypeDeclaration
Set0	20.474	20.542	20.533
Set1	61.193	61.388	61.226
Set2	534.448	547.339	535.156
Set3	1184.09	1219.15	1186.28
Set4	1279.42	1317.68	1281.88

- **Original.** This is the unaltered version of the JDTAST metamodel provided by the Grabats contest.
- **DerivedMethodDeclaration.** This version of the JDTAST metamodel includes three *EAnnotation* attributes (named *isPublic*, *isStatic* and *isSameReturnType*) in the *MethodDeclaration* class which contain the EOL code to derive (as a boolean) whether:
  – The current instance of this *MethodDeclaration* (self) has as return type the *TypeDeclaration* it is contained in. The EOL code reads as follows:

```
self.returnType.isTypeOf(SimpleType) and self.
    revRefNav_bodyDeclarations.isTypeOf(TypeDeclaration) and
    self.returnType.name.fullyQualifiedName == self.
    revRefNav_bodyDeclarations.name.fullyQualifiedName
```

– The current instance of this *MethodDeclaration* (self) is public. The EOL code reads as follows:

```
self.modifiers.exists(mod:Modifier|mod.public=="true"))
```

– The current instance of this *MethodDeclaration* (self) is static. The EOL code reads as follows:

```
self.modifiers.exists(mod:Modifier|mod.static=="true"))
```

Where the attribute *revRefNav_bodyDeclarations* allows reverse-navigation of the containment reference (*bodyDeclarations*) and retrieves the instance of the containing class (this is necessary as the JDTAST metamodel does not specify an opposite reference to the containment *bodyDeclarations*).

- **DerivedTypeDeclaration.** This version of the JDTAST metamodel includes a single *EAnnotation* attribute (named *isGrabats*) in the *TypeDeclaration* class which contains the EOL code to derive (as a boolean) whether the this instance (self) fulfills the Grabats query requirements. The EOL code reads as follows:

```
self.bodyDeclarations.exists(md:MethodDeclaration|md.modifiers
    .exists(mod:Modifier|mod.public=="true") and md.modifiers.
    exists(mod:Modifier|mod.static=="true") and md.returnType.
    isTypeOf(SimpleType) and md.returnType.name.
    fullyQualifiedName == self.name.fullyQualifiedName)
```

From table 2 we note that the increase in size when deriving attributes is very small (0.288% – 2.99%) so the only performance concern would be the increase in insertion time. In table 3 the numbers in brackets represent the time taken for the derivation of the attributes to be computed (which happens after the full model insertion). From table 3 we calculate the insertion time increases (using: $\frac{derivationtime}{totaltime-derivationtime} \times 100\%$) and present them in table 4. Table 4 demonstrates how there is a substantial (but reasonable) increase in insertion time for both derivation strategies presented. What is interesting is that even though DerivedTypeDeclaration computes a much heavier expression, due to the fact that it is performed sparsely, it requires comparable (and even slightly lower) insertion time to the DerivedMethodDeclaration strategy.

These results demonstrate that even though it is reasonable to add derived attributes even for quite complex derivations, careful consideration is needed so only important attributes are derived, otherwise it can result in unacceptable insertion times.

Table 3. Model Insertion, Execution time Results

Model	**Execution Time (in seconds)**		
	Original	DerivedMethodDeclaration	DerivedTypeDeclaration
Set0	16	16 (0.12)	16 (0.10)
Set1	34	36 (1.46)	37 (1.16)
Set2	553	658 (73)	625 (19)
Set3	2287	2650 (404)	2486 (347)
Set4	2502	2947 (493)	2893 (477)

Table 4. Model Insertion, Execution time Increase Percentage

Model	**Execution Time Increase (in %)**	
	DerivedMethodDeclaration	DerivedTypeDeclaration
Set0	0.756	0.629
Set1	4.23	3.237
Set2	12.48	3.135
Set3	17.99	16.22
Set4	20.09	19.74

5.4 Query Execution Time

Table 5 shows the results for performing the first Grabats 2009 [11,13] query on the various persisted models. As previously mentioned, the Grabats query finds all occurrences of *TypeDeclaration* elements that declare at least one public static method with the declared type as its returning type. For these tests three queries have been created in EOL (Q1 – Q3):

- *Q1* reads:

```
TypeDeclaration.all.collect(
  td|td.bodyDeclarations.select(
    md:MethodDeclaration|md.modifiers.exists(mod:Modifier|mod.
        public=="true")
    and md.modifiers.exists(mod:Modifier|mod.static=="true")
    and md.returnType.isTypeOf(SimpleType)
    and md.returnType.name.fullyQualifiedName == td.name.
        fullyQualifiedName ) )
```

This query (Q1) is the basic Grabats query using the original metamodel to insert the relevant models into Hawk. As such it only uses attributes found in the unaltered JDTAST metamodel.

- *Q2* reads:

  ```
  TypeDeclaration.all.collect(
    td|td.bodyDeclarations.select(
      md:MethodDeclaration|md.isPublic == "true"
      and md.isStatic == "true"
      and md.isSameReturnType == "true" ) )
  ```

  This query (Q2) contains the annotations described above for the DerivedMethodDeclaration insertion. As it uses attributes found in the unaltered JDTAST metamodel as well as the derived attributes 'isPublic', 'isStatic' and 'isSameReturnType'.
- *Q3* reads:

  ```
  TypeDeclaration.all.select( td|td.isGrabats == "true" )
  ```

  This query (Q3) contains the annotations described above for the DerivedTypeDeclaration insertion. As it uses attributes found in the unaltered JDTAST metamodel as well as the derived attribute 'isGrabats'.

Table 5. Grabats Query Execution Time Results

	Execution Time (in seconds)				
Model	Original	DerivedMethodDeclaration		DerivedTypeDeclaration	
	Q1	Q1	Q2	Q1	Q3
Set0	0.391	0.391	0.281	0.391	0.172
Set1	0.797	0.794	0.651	0.750	0.516
Set2	5.398	5.583	3.893	5.521	1.890
Set3	11.358	14.979	8.427	13.916	3.543
Set4	13.333	15.962	9.198	15.363	3.776

Query Q1 is run on all three types of inserted models as it does not contain any new constructs. Q2 is run on models which have used the DerivedMethodDeclaration annotations as it contains constructs using features derived by virtue of that annotated metamodel. Q3 is similarly run on models which have used the DerivedTypeDeclaration annotations.

It should be noted that the querying of the original models (using the original query – Q1) in Epsilon, which was presented in [4] has slightly worst execution times as it uses an older version of the EMC driver implemented for Hawk (and also ran Java 1.6).

The first interesting thing to note here is that running Q1 on the models with derived attributes is slightly less performant on the larger models (set2 – set4) than running it on the unaltered model. This is to be expected as the driver has to navigate through a larger database in these cases (as it is augmented with the derived attributes). As such, any operation which requires iteration on attributes of an object will possibly be slower than originally. Running Q1 on DerivedMethodDeclaration is slightly slower than Q1 on DerivedTypeDeclaration as DerivedTypeDeclaration only introduces one new attribute (isGrabats) for each *TypeDeclaration* while DerivedMethodDeclaration introduces three new attributes for each *MethodDeclaration* (and there are more *MethodDeclaration* instances than *TypeDeclaration* ones).

Looking at Q2, we see that it offers a significant performance increase to the original tests with mean improvement of 26.23% and maximum improvement of 31.01% (on the largest model, set4). Similarly for Q3 we note an even larger improvement in performance with mean 59.35% and maximum 71.68% (again, on the largest model, set4).

These results support the idea that for both small and large model sizes the targeted use of derived attributes can greatly benefit the resulting queries. What's more, these results seem to indicate that the larger the model size the more effective using derived attributes is in improving performance. Taking the larger models (set2 – set4) we note that the improvement percentage stays roughly the same or even tends to increase with the size of the model.

6 Conclusions and Further Work

From the empirical data collected we can conclude that using derived attributes in Hawk greatly improves the performance of queries performed that make use of them. There seems to be a steady (almost entirely positively correlated) relationship between the percentage increase in the performance gain (in terms of execution time) on queries performed and model size. Nevertheless there are two compromises to be made when considering the use of such attributes. The first is that the insertion time of models containing derived attributes is slower than the original models due to the overhead of deriving them. The second is that using several broad derived attributes, while less performant than using one single targeted derived attribute (while taking the same if not more time to insert), enables their possible use for different queries on the model while the targeted attribute can only be used in a much narrower scope. Finally, we note that general queries performed on models using derived attributes seem to be slightly less performant than ones using the original model; as such, derived attributes should only be used when there is reasonable confidence that they will be required (for example when needing to perform a known heavyweight transformation or query on the model).

Obtaining these encouraging results motivates us for implementing a fully engineered solution of using derived attributes in Hawk, while taking into account the concerns mentioned at the end of Section 3.1, in the future. Firstly, restricting the types of expressions allowed for derived attributes to be computed, so that model evolution can be performed in reasonable time, is planned. Next, a way to persist the expressions for derived attributes outside the metamodel, when the metamodel is immutable for example, will be looked at. Finally, use of embedded indexes found in Graph NoSQL databases in order to index specific attributes of interest, with the goal of further increasing query performance when such attributes are required for much of the computation of the query, will be investigated.

Acknowledgements. 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. Mohagheghi, P., Fernandez, M.A., Martell, J.A., Fritzsche, M., Gilani, W.: MDE Adoption in Industry: Challenges and Success Criteria. In: Chaudron, M.R.V. (ed.) MODELS 2008 Workshops. LNCS, vol. 5421, pp. 54–59. Springer, Heidelberg (2009)
2. 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 2008, Toulouse, France (2008)
3. Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform Random Generation of Huge Metamodel Instances. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 130–145. Springer, Heidelberg (2009)
4. Barmpis, K., Kolovos, D.: Evaluation of contemporary graph databases for efficient persistence of large-scale models. Journal of Object Technology (to appear, 2014)
5. Barmpis, K., Kolovos, D.: Hawk: Towards a scalable model indexing architecture. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, BigMDE 2013, pp. 6:1–6:9. ACM, New York (2013)
6. Kolovos, D.S., Rose, L., Garcia, A.D., Paige, R.F.: The Epsilon Book (2008), `http://www.eclipse.org/epsilon/doc/book/`
7. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Object Language. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
8. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.: The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In: Proc. 14th IEEE International Conf. on Engineering of Complex Computer Systems, Potsdam, Germany (2009)
9. Willink, E.: Aligning OCL with UML. In: Proceedings of the Workshop on OCL and Textual Modelling. Electronic Communications of the EASST (2011)
10. Kolovos, D.S., Wei, R., Barmpis, K.: An approach for efficient querying of large relational datasets with ocl-based languages. In: XM 2013–Extreme Modeling Workshop, p. 48 (2013)
11. Grabats2009: 5th Int. Workshop on Graph-Based Tools (2012), `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/`
12. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: A repository for scalable model management. Software & Systems Modeling, 1–21 (2013)
13. Sottet, J.S., Jouault, F.: Program comprehension. In: Proc. 5th Int. Workshop on Graph-Based Tools (2009)