
Article:
Paige, Richard Freeman orcid.org/0000-0002-1978-9852, Kolovos, Dimitrios orcid.org/0000-0002-1724-6563, Barmpis, Konstantinos et al. (2 more authors) (2019) Stress-Testing Remote Model Querying APIs for Relational and Graph-Based Stores. Software and Systems Modeling. pp. 1047-1075. ISSN: 1619-1366

https://doi.org/10.1007/s10270-017-0606-9

Noname manuscript No.
(will be inserted by the editor)

Stress-Testing Remote Model Querying APIs for Relational and Graph-Based Stores

Antonio Garcia-Dominguez · Konstantinos Barmpis · Dimitrios S. Kolovos · Ran Wei · Richard F. Paige

Received: date / Accepted: date

Abstract Recent research in scalable model-driven engineering now allows very large models to be stored and queried. Due to their size, rather than transferring such models over the network in their entirety, it is typically more efficient to access them remotely using networked services (e.g. model repositories, model indexes). Little attention has been paid so far to the nature of these services, and whether they remain responsive with an increasing number of concurrent clients. This paper extends a previous empirical study on the impact of certain key decisions on the scalability of concurrent model queries on two domains, using an Eclipse Connected Data Objects model repository, four configurations of the Hawk model index and a Neo4j-based configuration of the NeoEMF model store. The study evaluates the impact of the network protocol, the API design, the caching layer, the query language and the type of database, and analyses the reasons for their varying levels of performance. The design of the API was shown to make a bigger difference compared to the network protocol (HTTP/TCP) used. Where available, the query-specific indexed and derived attributes in Hawk outperformed the comprehensive generic caching in CDO. Finally, the results illustrate the still ongoing evolution of graph databases: two tools using different versions of the same backend had very different performance, with one slower than CDO and the other faster than it.

A. Garcia-Dominguez
School of Engineering and Applied Science, Aston University
Aston Triangle, Birmingham, B4 7ET
Tel.: +44 0121 204 4454
E-mail: a.garcia-dominguez@aston.ac.uk

K. Barmpis, D. S. Kolovos, R. Wei, R. F. Paige
Department of Computer Science, University of York
Deramore Lane, York, YO10 5GH
Tel.: +44 01904 325500
E-mail: {konstantinos.barmpis,dimitris.kolovos,ran.wei,richard.paige}@york.ac.uk

Keywords model persistence · remote model querying · NoSQL storage · relational databases · API design · stress testing · collaborative modelling

1 Introduction

Model-driven engineering (MDE) has received considerable attention due to its demonstrated benefits of improving productivity, quality and maintainability. However, industrial adoption has ran into various challenges regarding the maturity and scalability of MDE. Mohagheghi et al (2012) interviewed participants from four companies and noted concerns that the tools at the time did not scale to the large projects that would merit the use of MDE. Several ways in which MDE practice could learn from widely-used programming environments to handle large models better were pointed out by Kolovos et al (2008), with a strong focus on the need for modularity in modelling languages to improve scalability and simplify collaboration. Three categories of scalability issues in MDE were identified by Barmpis and Kolovos (2014a):

- Model persistence: storage of large models; ability to access and update such models with low memory footprint and fast execution time.
  The simplest solution (using one file per model) has not scaled well as models increase in size. One alternative approach is *fragmenting* the models into multiple smaller files. Another option is writing a *model persistence layer* that stores the model in a database of a certain type (relational, graph-oriented and so on).
- Model querying and transformation: ability to perform intensive and complex queries and transformations on large models with fast execution time.
  Efficient queries and transformations are closely related to the type of persistence used for the models. Fragmented models can be backed with an incrementally maintained *model index* (such as Hawk[1]) that can answer queries faster than going through the fragments. For database-backed models, the query must be transformed to an efficient use of the database, and the database must provide a high level of performance. This is the approach taken by Mogwaï[2], a query engine for models stored in the NeoEMF[3] layer that transforms OCL queries into Gremlin[4] API calls.
- Collaborative work: multiple developers being able to query, modify and version control large-scale shared models in a non-invasive manner.
  With fragmented models, existing version control systems can be reused. Database-backed systems need to implement their own version control approaches: this is the approach taken in *model repositories* such as CDO[5].

[1] `https://github.com/mondo-project/mondo-hawk`
[2] `https://github.com/atlanmod/Mogwai`
[3] `https://github.com/atlanmod/NeoEMF`
[4] `http://tinkerpop.apache.org/gremlin.html`
[5] `http://wiki.eclipse.org/CDO`

Regardless of how models are stored, high-performance querying is crucial when dealing with very large models. For instance, within the building industry it is common to use building information models (BIM) containing millions of elements and covering the logical and physical structure of entire buildings. These models need to be queried e.g. to compute *quantity takeoffs* which estimate the materials needed to complete construction (Bagnato et al, 2014). Reverse engineering source code into models (Bruneliére et al, 2014) also produces very large models, and these need to be queried to find design flaws or elements to be modernized, among other things. Complex graph pattern matching may further complicate things, as when validating railway models (Szárnyas et al, 2017).

Sharing models by sending files manually is inefficient (in effort and transmission time) and prone to mistakes (e.g. having someone use an outdated version). Instead, it is considered better to use model repositories such as CDO or file repositories such as Git, and to expose the models for querying/modification through networked services. As an example, in previous work (Garcia-Dominguez et al, 2016a), we demonstrated how Hawk enabled Constellation model repositories to offer dashboards with model metrics and advanced searching from a web interface. Within the MONDO project, one of the tools for collaborative modelling implemented an "online" approach where multiple concurrent users accessed the model over a web interface (Rath and Varró, 2016).

Exposing models through networked services introduces new layers of complexity, such as the design and implementation of the service, or the interactions between the layers as more and more clients try to access a model at the same time. Existing studies have not analysed these new factors, considering only local queries within the same machine or the "best case" scenario with only one remote user. It is important to stress-test these networked services, as solutions may exhibit various issues in high-load situations.

In this empirical study, we will evaluate the impact of several design decisions in the remote model querying services offered by multiple existing solutions (CDO, Hawk and Mogwai). While these tools have different goals in mind, they all offer this same functionality, and they all had to choose a particular network protocol, messaging style, caching/indexing style, query language and persistence mechanism. The results of this study aim to inform developers and end users of future remote model querying services on the tradeoffs between these choices.

This paper is an extended version of our prior conference work (Garcia-Dominguez et al, 2016b), which discussed a smaller study with fewer tools, queries and research questions. The new contributions of this paper are:

- An updated and extended discussion of the state of the art, with recent works on prefetching, partial loading, and non-relational model stores.
- A largely expanded experimental design, testing four additional tool configurations (Hawk with Neo4j/EPL, Hawk with OrientDB/EOL, Hawk with OrientDB/EPL and Mogwaï), new and revised queries for the GraBaTs'09

case study, and a new case study based on the queries from the Train Benchmark by Szárnyas et al (2017). The previous research question on the impact of the internals of the tools (RQ3) has been refined into multiple research questions.
- A revamped and expanded results section, with a stronger focus on statistical tests in order to manage the much larger volume of data in this work. Only the results from RQ2 have remained intact, since the APIs for CDO and Hawk have not changed.
- A revised set of conclusions, taking into account the more nuanced results produced by the Train Benchmark case study.

The rest of this work is structured as follows: Section 2 provides a discussion on existing work on model stores, Section 3 introduces the research questions and the design of the experiment, Section 4 discusses the obtained results and Section 5 presents the conclusions and future lines of work.

2 Background and Related Work

Persisting and managing large models has been extensively investigated over the past decade. This section presents the main state-of-the-art tools and technologies, with a focus on the tools used in this empirical study.

2.1 File-Based Model Persistence

One of the most common formats for storing models are files containing a serialized model representation. Tools like the Eclipse Modeling Framework (EMF) (Steinberg et al, 2008), ModelCVS (Kramler et al, 2006), Modelio[6] and MagicDraw[7] all use XML-based model serialisation. StarUML[8] stores models in JSON. To improve performance, many tools offer binary formats as well: this is the case for EMF, for instance.

Files are easy to deploy and use, and many tools (e.g. EMF) default to using a one-file-per-model approach. However, storing one model per file impacts scalability negatively as shown in (Barmpis and Kolovos, 2014a; Gómez et al, 2015). In this case, even a simple query or a small change requires loading the entire model in memory at once: this is impractical for large models. Recent work by Wei et al (2016) demonstrated a specialisation of the EMF XMI parser which can load only the subset required by the query to be run: while this reduced loading times and memory usage, changes in the partially loaded models cannot be saved back without losing information.

These limitations in scalability suggest that it could be beneficial to break up large models into smaller units (or "fragments") to allow for on-demand partial loading. Modelio does this by default in recent versions: for instance, each

[6] `https://www.modelio.org/`
[7] `http://www.nomagic.com/products/magicdraw.html#Collaboration`
[8] `http://staruml.io/`

UML class is stored in a separate file, and links between files are resolved through a purpose-built index. For EMF-based models, the EMF-Splitter framework by Garmendia et al (2014) can take a metamodel annotated with modularity information and produce editors that produce fragmented XMI-based models natively. Nevertheless, in a worst-case scenario, certain types of queries (e.g. a query that looked for all instances of a type) could still require loading the full set of fragments.

2.2 Database-Backed Model Persistence

In light of the scalability limitations resulting from storing models as text files, various database-backed model persistence formats have been proposed. Database persistence allows for partial loading of models as only accessed elements have to be loaded in each case. Furthermore, such technologies can leverage database indices and caches for improving element lookup performance as well as query execution time.

Most of these database-backed solutions store each object as its own database entity (e.g. row, document or graph node). This is the case for Teneo/Hibernate[9], one of the first Object-Relational Mappings (ORMs) for EMF models. More recent systems which store models in databases rely on NoSQL technologies to take advantage of their flexible schema-free storage and/or quick reference navigation, such as MongoEMF[10] (based on the MongoDB document store) or NeoEMF (Gómez et al, 2015). NeoEMF in particular implements a multi-backend solution: NeoEMF/Graph uses graph-based databases (Neo4j[11] in particular), NeoEMF/Map uses file-backed maps (as implemented by MapDB[12]), and NeoEMF/HBase uses HBase[13] distributed stores.

However, there are also approaches that operate at the fragment level: this is the case for EMF-Fragments by Scheidgen (2013). In this tool, the model is broken up along the EMF containment references that have been marked to be "fragmenting", and these fragments are addressable through a key-value store. The EMF-Fragments tool supports both MongoDB and HBase. Users can choose how to represent each inter-object reference in the metamodel: these can be kept as part of the source object (as usual in EMF XMI-based persistence) or separately from it (as usual in database-backed persistence).

For most of these database-backed solutions, querying is an orthogonal concern: existing query languages can be used, but the languages will not be able to leverage the underlying data structures to optimise certain common cases (e.g. OCL's "Type.allInstances()") or avoid constructing intermediate objects in memory. Mogwaï is a model query framework that tackles this issue for models stored in NeoEMF/Graph, translating OCL queries to Tinkerpop

[9] `http://wiki.eclipse.org/Teneo/Hibernate`
[10] `https://github.com/BryanHunt/mongo-emf/wiki`
[11] `https://neo4j.com/`
[12] `http://www.mapdb.org/`
[13] `http://hbase.apache.org/`

Gremlin through ATL and reporting reductions in execution up to a factor of 20 (Daniel et al, 2016).

2.3 Model Repositories

When collaborative modeling is involved, simply storing models in a scalable form such as inside a database stops being sufficient; in this case issues such as collaborative access and versioning need to also be considered. Examples of model repository tools are Morsa (Pagán et al, 2013), ModelCVS[14], Connected Data Objects (CDO), EMFStore (Koegel and Helming, 2010), Modelio, MagicDraw and MetaEdit+[15]. Model repositories allow multiple developers to manage models stored in a centralised repository by ensuring that models remain in a consistent state, while persisting them in a scalable form, such as in a database.

CDO in particular is one of the most mature solutions, having been developed since 2009 as an Eclipse project and being currently maintained by Obeo[16]. It implements a pluggable storage architecture that enables it to use various solutions such as relational databases (H2, MySQL) or document-oriented databases (MongoDB), among others. CDO includes Net4j, a messaging library that provides bidirectional communication over TCP, HTTP and in-memory connections, and uses it to provide an API that exposes remote models as EMF resources. In addition to storing models, CDO includes a CDOQuery API that makes it possible to run queries remotely on the server, reducing the necessary bandwidth.

2.4 Heterogeneous Model Indexing

An alternative to using model repositories for storing models used in a collaborative environment is to store them as file-based models in a classical version control system, ideally in a fragmented form. As discussed by Barmpis et al (2015), this approach leverages the benefits of widely-used file-based version control systems such as SVN and Git, but retains the issues file-based models face (Section 2.1). To address this issue a model indexer can be introduced that monitors the models and mirrors them in a scalable model index. The model index is synchronised with the latest version of the models in the repository and can be used to perform efficient queries on them, without having to check them out locally or load them into memory.

One example of such a technology is Hawk[17]. Hawk can maintain a graph database which mirrors the contents of the models stored in one or more version control repositories and perform very efficient queries on them. Hawk

[14] `http://www.modelcvs.org/versioning/index.html`

[15] `http://www.metacase.com/`

[16] As stated in `http://projects.eclipse.org/projects/modeling.emf.cdo`

[17] `https://github.com/mondo-project/mondo-hawk`

can be used as a Java library, as a set of plugins for the Eclipse IDE, or as a network service through an Apache Thrift[18]-based API.

Hawk can be extended to add support for various file formats, storage backends and query languages. As part of the integration efforts with the Softeam Modelio and Constellation products (Garcia-Dominguez et al, 2016a), two new components were added: a model parser for Modelio EXML/RAMC files, and a storage backend based on OrientDB. OrientDB is an open source multi-paradigm database engine which can operate as a key-value store, as a document database or as a graph database. While studies from 2014 showed that OrientDB had lower performance than Neo4j for model querying (Barmpis and Kolovos, 2014a), its relative performance with regards to Neo4j has improved since then, and its more permissive license makes it more appealing to industrial users (ASL2.0 instead of Neo4j's GPLv3).

3 Experiment Design

As mentioned in the introduction, once we have scalable modelling and scalable querying, the next problem to solve is how to share those huge models across the organisation. Exposing them through a model querying service over the network is convenient, as they can provide answers without waiting for the model itself to be transferred. However, the design and implementation of the service is not trivial, and the underlying implementation may not react well to serving multiple concurrent clients.

This section presents the design of an empirical study that evaluates the impact of several factors in the performance of the remote model querying services of multiple tools: a model repository (CDO), several configurations of a model index (Hawk with Neo4j/OrientDB backends and EOL/EPL queries), and a database-backed model storage layer (NeoEMF). By studying the performance of these queries, we will be evaluating the responsiveness of the underlying tools with increasing levels of demand and how their different layers interact with each other.

3.1 Research Questions

RQ1. *What is the impact of the network protocol on remote query times and throughputs?*

In order to connect to a remote server, two of the most popular options are using raw TCP connections (for the sake of performance and flexibility) or sending HTTP messages (for compatibility with web browsers and interoperability with proxies and firewalls). Both Hawk and CDO support TCP and HTTP. Since NeoEMF did not officially have a remote querying API at

[18] `http://thrift.apache.org/`

the time of writing this paper, it was extended by the authors with TCP and HTTP-based APIs implemented in the same way as Hawk's.

Properly configured HTTP servers and clients can reuse the underlying TCP connections with HTTP 1.1 pipelining and avoid repeated handshakes, but the additional overhead imposed by the HTTP fields may still impact the raw performance of the tool.

RQ2. *What is the impact of the design of the remote query API on remote query times and throughputs?*

Application protocols for network-based services can be stateful or stateless. Stateful protocols require that the server keeps track of part of the state of the client, while stateless protocols do not have this requirement. In addition, the protocol may be used mostly for transporting opaque blocks of bytes between server and client, or it might have a well-defined set of operations and messages.

While a stateful protocol may be able to take advantage of the shared state between the client and server, a stateless protocol is generally simpler to implement and use. Service-oriented protocols need to also take into account the granularity of each operation: "fine" operations that do only one thing may be easier to recombine, but they will require more invocations than "coarse" operations that perform a task from start to finish. One example of a fine operation could be fetching a single model element by ID. A coarse operation would be running an entire query in the server and retrieving the results.

CDO implements a stateful protocol on top of the Net4j library, which essentially consists of sending and receiving buffers of bytes across the network. On the other hand, Hawk and our extended version of NeoEMF implement a stateless service-oriented API on top of the Apache Thrift library, exposing a set of specific operations (e.g. "query", "send object" or "register metamodel"). The Hawk API supports both fine- and coarse-grained operations (fetching single elements or running queries), whereas the Mogwai API only supports running entire queries. Invoking a query for Hawk and Mogwai only requires one pair of HTTP request/response messages.

While the stateful CDO clients and servers may cooperate better with each other, the simpler and coarser APIs in Hawk and Mogwaï may reduce the total network roundtrip for a query by exchanging fewer messages.

RQ3. *What is the impact of the internal caching and indexing mechanisms on remote query times and throughputs?*

Database-backed systems generally implement various caching strategies to keep the most frequently accessed data in memory, away from slow disk I/O. At the very least, the DBMS itself will generally keep its own cache, but the system might use additional memory to cache especially important subsets or to keep them in a form closer to how it is consumed.

Another common strategy is to prepare indices in advance, speeding up particular types of queries. DBMSs already provide indices for common concepts such as primary keys and unique values, but these systems may add

their own application-specific indices that precompute parts of the queries to be run.

RQ4. *What is the impact of the mapping from the queries to the backend on remote query times?*

Remote query APIs are usually bound to certain model querying languages: CDO embeds an OCL interpreter, Hawk has the Epsilon languages and NeoEMF translates a subset of OCL to Gremlin through Mogwaï. Once the query is written, it has to be run by a query engine against the chosen backend.

The interactions between the query language, the engine and the underlying backend need to be analysed. Declarative query languages delegate more work into the query engine, whereas imperative query languages rely on the user to fine-tune accesses. Query engines have to map the query into an efficient use of the backend. In some cases, there may be useful features in a backend that are not made available to users, whether due to a limitation in the mapping of the query engine, or to the lack of a matching concept in the query.

RQ5. *Do graph-based tools scale better against demand than tools that store models in relational databases?*

Various authors (including the authors of this paper) have previously reported considerable performance gains when running single queries on graph-based solutions when compared to solutions backed by databases or flat files. It may seem that graph databases are always the better choice, but they have been around for less time than relational approaches and usually require more fine tuning to achieve the ideal performance. This question will focus on whether this advantage is common across graph-based tools and if it extends to situations with very high levels of demand.

3.2 Experiment Setup

In order to provide answers for the above research questions, a networked environment was set up to emulate increasing numbers of clients interacting with a model repository (CDO 4.4.1.v20150914-0747), a model index (Hawk 1.0.0. 201609151838) or a graph-based model persistence layer (NeoEMF on commit `375e077` combined with Mogwaï on commit `543fec9`) and collect query response times. The environment is outlined in Figure 1, and consists of the following:

– **One "Controller" machine** that supervises the other machines through SSH connections managed with the Fabric Python library[19]. It is responsible for starting and stopping the client and server processes, monitoring their execution, and collecting the measured values. It does not run any queries itself, so it has no impact on the obtained results.

[19] `http://www.fabfile.org/`

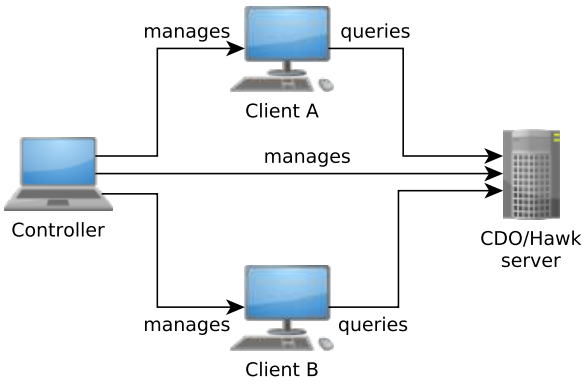

Fig. 1: Network diagram for the experimental setup

– **Two "Client" machines** that invoke the queries on the server, fetch the results and measure query response times. The two client machines were running Ubuntu Linux 14.04.3, Linux 3.19.0-47-generic and Oracle Java 8u60 on an Intel Core i5 650 CPU, 8GiB of RAM and a 500GB SATA3 hard disk.
  The client machines had three client programs installed: one for CDO, one for Hawk and one for Mogwaï/NeoEMF. Only one of these programs ran at a time. Each of these programs received the address of the server to connect to, the size of the Java fixed thread pool to be used, the number of queries to be distributed across these threads and the query to be run. The clients sent their queries to the server and simply waited to receive the response from the server: they did not fetch model elements directly[20].
– **One "Server" machine** that hosts the CDO model repository, the Hawk model index and the NeoEMF model store, and provides TCP and HTTP ports exposing the standard CDO and Hawk APIs for remote querying and a small proof of concept API for NeoEMF/Mogwaï. The server machine had the same configuration as the client machines. The server waits to receive a query and runs it locally through an embedded database, and then replies back with the identifiers of the matching model elements.
  The server machine also had three server programs installed: one for CDO, one for Hawk and one for Mogwaï/NeoEMF. Again, only one of these programs ran at a time. All server programs were Eclipse products based on Eclipse Mars and used the same embedded HTTP server (Eclipse Jetty

[20] Early experiments where the Hawk clients did access the models over the network to run the queries showed unsatisfactory performance, with query times an order of magnitude slower than sending the query to be run by the server. This led us to discard this alternative.

9.2.13). All systems were configured to use up to 4096MB of memory (`-Xmx4096m -Xms2048m`)[21].
In particular, the CDO server was based on the standard CDO server product, with the addition of the experimental HTTP Net4j connector. No other changes were made to the CDO configuration. The CDO *DB Store* storage component was used in combination with the default H2 database adapter. *DB Store* was the most mature and feature-complete option at the time of writing[22].
- **One 100Mbps network switch** that connected all machines together in an isolated local area network.

As the study was intended to measure query performance results with increasing numbers of concurrent users, the client programs were designed to first warm up the servers into a *steady state*. Query time was measured as the time required to connect to the server, run the query on the server and retrieve the model element identifiers of the results over the network. Queries would be run 1000 times in all configurations, to reduce the impact of variations due to external factors (CPU and I/O scheduling, Java just-in-time recompilation, disk caches, virtual memory and so on).

Several workloads were defined. The lightest workload used only 1 client machine with 1 thread sending 1000 queries to the server in sequence. The other workloads used 2 client machines generating load at the same time using a producer/consumer design where the producer thread would queue 500 query invocations, and $t \in \{2, 4, 8, 16, 32\}$ consumer threads (*client threads*) would execute them as quickly as possible. For instance, with 2 client threads in a machine, each thread would be expected to execute approximately 250 invocations: the exact number might slightly vary due to differences in execution time across invocations. These workloads could therefore simulate between 1 (1 machine with 1 client thread) and 64 (2 machines at the same time, with 32 client threads each) concurrent clients.

3.3 Queries Under Study

After defining the research questions and preparing the environment, the next step was to populate CDO, Hawk and Mogwaï/NeoEMF with the contents to be queried, and to write equivalent queries in their supported query languages. Two use cases were considered, each with their own sets of queries: one related to reverse engineering existing Java code, and one related to pattern matching in railway models.

[21] The Neo4j performance guide suggests this amount for a system with up to 100M nodes and 8GiB RAM, to allow the OS to keep the graph database in its disk cache.

[22] According to the online help in the June 2016 release: `http://download.eclipse.org/modeling/emf/cdo/drops/R20160607-1209/help/org.eclipse.emf.cdo.doc/html/reference/StoreFeatures.html`.

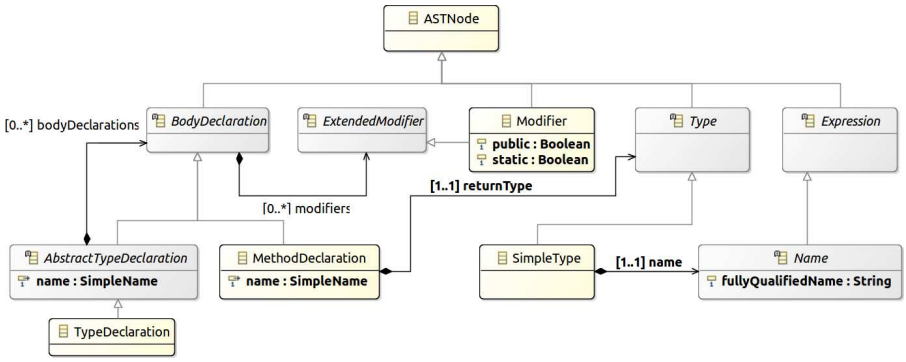

Fig. 2: Relevant excerpt of the JDTAST metamodel for the GraBaTs'09 queries.

3.3.1 Singletons in Java models: GraBaTs'2009 queries

The first use case, *SharenGo Java Legacy Reverse-Engineering*[23], was based on MoDisco and was originally presented at the GraBaTs 2009 tool contest (GraBaTs, 2009). It has been widely used for research in scalable modelling (Barmpis and Kolovos, 2014a; Pagán et al, 2013; Benelallam et al, 2014; Carlos et al, 2015), as it provides a set of models reverse-engineered from increasingly large open source Java codebases. The largest codebase in the case study was selected, covering all the `org.eclipse.jdt` projects and producing over 4.9 million model elements. CDO required 1.4GB to store the model, Hawk required 2.0GB with Neo4j and 3.7GB with OrientDB, and NeoEMF required 6.0GB.

These model elements conformed to the Java Development Tools AST (JDTAST) metamodel. Some of the types within the JDTAST metamodel include the TypeDeclarations that represent Java classes and interfaces, the MethodDeclarations that represent Java methods, and the Modifiers that represent Java modifiers on the methods (such as `static` or `public`). The relevant excerpt of the metamodel is shown in Figure 2.

Based on these types, task 1 in the GraBaTs 2009 contest required defining a query (from now on referred to as the GraBaTs query) that would locate all possible applications of the Singleton design pattern in Java (Sottet and Jouault, 2009). In other words, it would have to find all the TypeDeclarations that had at least one MethodDeclaration with `public` and `static` modifiers that returned an instance of the same TypeDeclaration.

To evaluate CDO, the GraBaTs query was written in OCL as shown in Listing 1. The query (named OQ after "OCL query") filters the TypeDeclarations by iterating through their MethodDeclarations and their respective Modifiers.

[23] http://www.eclipse.org/gmt/modisco/useCases/JavaLegacyRE/

Listing 1: GraBaTs query written in OCL (OQ) for evaluating CDO.

```
DOM::TypeDeclaration.allInstances()→select(td |
    td.bodyDeclarations→selectByKind(DOM::MethodDeclaration)
      →exists(md : DOM::MethodDeclaration |
        md.modifiers→selectByKind(DOM::Modifier)
          →exists(mod : DOM::Modifier | mod.public)
        and md.modifiers→selectByKind(DOM::Modifier)
          →exists(mod : DOM::Modifier | mod._static)
        and md.returnType.oclIsTypeOf(DOM::SimpleType)
        and md.returnType.oclAsType(DOM::SimpleType).name.fullyQualifiedName
          = td.name.fullyQualifiedName))
```

Listing 2: GraBaTs query written in EOL (HQ1) for evaluating Hawk.

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

Listing 3: GraBaTs query written in EOL (HQ2) using derived attributes on the MethodDeclarations for evaluating Hawk, without indexed lookups.

```
return TypeDeclaration.all.select(td |
  td.bodyDeclarations.exists(md:MethodDeclaration |
    md.isPublic = true and md.isStatic = true and md.isSameReturnType = true));
```

Listing 4: GraBaTs query written in EOL (HQ3) using derived attributes on the MethodDeclarations for evaluating Hawk, with indexed lookups.

```
return MethodDeclaration.all.select(md |
  md.isPublic = true and md.isStatic = true and md.isSameReturnType = true
).collect( td | td.eContainer ).asSet;
```

To evaluate Hawk, we used the three EOL implementations of the GraBaTs query of our previous work (Barmpis and Kolovos, 2014b). The first version of the query ("Hawk query 1" or HQ1, shown in Listing 2) is a translation of OQ to EOL, and follows the same approach.

The second version (HQ2), shown in Listing 3, makes use of thee derived attributes on MethodDeclarations: *isStatic* (the method has a `static` modifier), *isPublic* (the method has a `public` modifier), and *isSameReturnType* (the method returns an instance of its TypeDeclaration). A detailed discussion about how derived attributes are declared in Hawk and how they

Listing 5: GraBaTs query written in EOL (HQ4) using derived attributes on the TypeDeclarations for evaluating Hawk.

```
return TypeDeclaration.all.select(td|td.isSingleton = true);
```

Listing 6: GraBaTs query written in OCL for Mogwaï (MQ)

```
import DOM : 'platform:/resource/jdtast.neoemf/model/JDTAST.ecore'

package DOM
  context TypeDeclaration
    def: singletons : Set(TypeDeclaration) =
      TypeDeclaration.allInstances()→select(td |
        td.bodyDeclarations→exists(md | md.oclIsTypeOf(MethodDeclaration)
          and md.modifiers→exists(mod |
            mod.oclIsKindOf(Modifier) and mod.oclAsType(Modifier).public = true)
          and md.modifiers→exists(mod |
            mod.oclIsKindOf(Modifier) and mod.oclAsType(Modifier)._static = true)
          and md.oclAsType(MethodDeclaration).returnType.oclIsTypeOf(SimpleType)
          and md.oclAsType(MethodDeclaration).returnType.oclAsType(SimpleType)
            .name.fullyQualifiedName = td.name.fullyQualifiedName))
endpackage
```

are incrementally re-computed upon model changes is available in our previous works (Barmpis et al, 2015; Barmpis and Kolovos, 2014b).

The third version (HQ3), shown in Listing 4, uses the same derived attributes but starts off from the MethodDeclarations so Hawk can take advantage of the fact that derived attributes can also be indexed, replacing iterations by lookups and noticeably speeding up execution.

The fourth version (HQ4), shown in Listing 5, assumed instead that Hawk extended TypeDeclarations with the *isSingleton* derived attribute, setting it to true when the TypeDeclaration has a `static` and `public` MethodDeclaration returning an instance of itself. This derived attribute eliminates one more level of iteration, so the query only goes through the TypeDeclarations.

The query for Mogwaï (MQ) is shown in Listing 6. Ideally we would have used the same OCL query in CDO and in Mogwaï, but unfortunately CDO OCLQuery only accepts raw expressions and Mogwaï only accepts constraints within packages and contexts. Additionally, there are limitations in Mogwaï's implementation (particularly, the ATL transformation from OCL to the Gremlin API) that require making small changes in the queries. For instance, the OCL translator in Mogwaï does not support the Eclipse OCL-specific "selectByKind" operation, and additional type conversions are needed.

The GraBaTs query has been translated to one OCL query for CDO (OQ), 1 OCL query for Mogwaï (MQ) and four possible EOL queries for Hawk (HQ1 to HQ4). It must be noted that since CDO and Mogwaï/NeoEMF do not support derived attributes like Hawk, it was not possible to rewrite OQ or

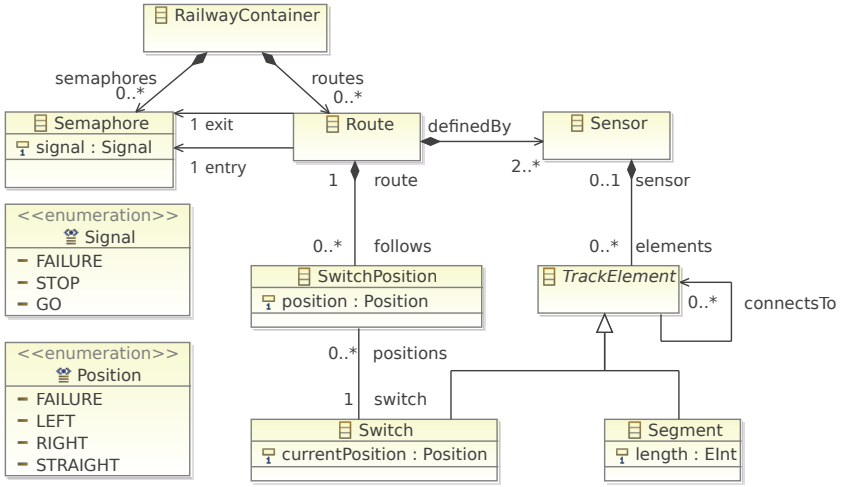

Fig. 3: Containment hierarchy and references of the metamodel of the Train Benchmark (Szárnyas et al, 2017).

MQ in the same way as HQ1. Since the same query would be repeatedly run in the experiments, the authors inspected the code of CDO, Mogwaï and Hawk to ensure that neither tool cached the results of the queries themselves: this was verified by re-running the queries while adding unique trivially true conditions, and comparing execution times.

3.3.2 Railway model validation: Train Benchmark queries

To improve the external validity of the answers for the research questions in Section 3.1, a second case study with a wider assortment of queries was needed. For this purpose, it was decided to use some of the queries and models from the Train Benchmark by Szárnyas et al (2017).

The Train Benchmark (TB) was originally developed within the MONDO EU FP7 project on scalability in model-driven engineering, in order to compare the querying performance of various technical solutions with regards to model validation. The original benchmark divided execution into four stages (*read*, *check*, *edit* and *re-check*), and tested two scenarios: a *batch* scenario with only *read* and *check*, and an *incremental* scenario with all four stages. Since the focus of the present study is on scalability to user demand rather than reacting to changes, only the batch scenario will be adopted.

The queries on the TB operate on domain-specific models of railway systems: the containment hierarchy and references of the underlying metamodel are shown on Figure 3. The RailwayContainer acts as the root of the model, which contains Routes and the Semaphores between them. A Route is formed of two or more Sensors which monitor Switches and Segments. The Switches have a particular SwitchPosition for each Route.

Based on this metamodel, the TB includes automatic model generators that can produce synthetic models of arbitrary size by producing a random number of small fragments and reconnecting them in a random manner. After this, a small portion of the elements (<1%) is modified to produce the validation errors that will be detected by the later queries. The benchmark includes generators for both the *repair* case (the *edit* stage corrects validation errors) and the *inject* case (*edit* introduces validation errors).

For the present study, the *repair* generator was used. Multiple models were generated during initial experiments (with between 1,418 and 3,355,152 elements) by varying the *size* parameter of the generator between 1 and 2048. However, some of the queries were too slow on CDO and Mogwaï for stress testing, and a medium-sized model had to be selected (*size* = 32, with 49 334 elements). In general, the simplicity of the TB metamodel ensures that queries access larger portions of the model than the GraBaTs queries in Section 3.3.1, and some of the queries perform more complex pattern matching as well.

The present study used the OCL versions of the TB queries for CDO and Mogwaï, with some adjustments in the case of Mogwaï. For Hawk, the OCL queries were translated to the Epsilon Object Language (EOL), optimized for Hawk features and then further translated to the Epsilon Pattern Language (EPL). EPL (Kolovos et al, 2016) is a specialisation of EOL, providing a more declarative and readable syntax for graph pattern matching in models. The queries look for violations of various well-formedness constraints:

- *ConnectedSegments* (CS): each Sensor must have 5 or fewer Segments connected to them. The queries in Figure 4 find Sensors that are monitoring a sequence of 6 or more Segments.
  The Mogwaï version is similar to the original OCL one, but it can only return the sixth TrackElement that produces the violation. The original OCL query packed all the participants in each match into a list of tuples, but Mogwaï queries can only return a flat list of individual model elements. Matching a sequence of six consecutive elements is rather awkward, requiring many nested repetitions of *select* and *collect*. EOL has a similar readability issue, but EPL has a much cleaner syntax for this sort of graph matching problem.
- *PosLength* (PL): Segments must have positive length. The queries on Figure 5 find Segments that have zero or negative length.
  In this case, Hawk can be told to index the "length" attribute of Segment in advance to jump directly to the relevant elements.
- *RouteSensor* (RS): Sensors associated to a Switch that belongs to a Route must also be associated with the same Route. The queries on Figure 6 find Sensors that are connected to a Switch, but the Sensor and the Switch are not connected to the same Route.
  The EOL and EPL versions filter the Sensors by taking advantage of a derived attribute, "nMonitoredSegments", defined through the EOL expression "self.monitors.size" (where "self" takes each of the Sensors as a

```
railway::Sensor.allInstances()→collect(
 sensor | sensor.monitors→select(oclIsKindOf(railway::Segment))→
  collect(segment1 | segment1.connectsTo→select(oclIsKindOf(railway::Segment))→
   select(segment2 | segment2.monitoredBy→includes(sensor))→
   collect(segment2 | segment2.connectsTo→select(oclIsKindOf(railway::Segment))→
    select(segment3 | segment3.monitoredBy→includes(sensor))→
    collect(segment3 | segment3.connectsTo→select(oclIsKindOf(railway::Segment))→
     select(segment4 | segment4.monitoredBy→includes(sensor))→
     collect(segment4 | segment4.connectsTo→select(oclIsKindOf(railway::Segment))→
      select(segment5 | segment5.monitoredBy→includes(sensor))→
      collect(segment5 | segment5.connectsTo→select(oclIsKindOf(railway::Segment))→
      select(segment6 | segment6.monitoredBy→includes(sensor))→collect(
       segment6 | Tuple{sensor = sensor, segment1 = segment1,
        segment2 = segment2, segment3 = segment3, segment4 = segment4,
        segment5 = segment5, segment6 = segment6}))))))
```

(a) Original OCL

```
import railway : 'platform:/resource/railway.neoemf/model/railway.ecore'

package railway
 context Sensor
 def: connectedSegments : Bag(TrackElement) = Sensor.allInstances()→collect(sensor |
  sensor.monitors→select(oclIsTypeOf(Segment))→collect(segment1 |
   segment1.connectsTo→select(oclIsTypeOf(Segment))→select(segment2 |
    segment2.monitoredBy→includes(sensor))→collect(segment2 |
    segment2.connectsTo→select(oclIsTypeOf(Segment))→select(segment3 |
     segment3.monitoredBy→includes(sensor))→collect(segment3 |
     segment3.connectsTo→select(oclIsTypeOf(Segment))→select(segment4 |
      segment4.monitoredBy→includes(sensor))→collect(segment4 |
      segment4.connectsTo→select(oclIsTypeOf(Segment))→select(segment5 |
       segment5.monitoredBy→includes(sensor))→collect(segment5 |
       segment5.connectsTo→select(oclIsTypeOf(Segment))→select(segment6 |
        segment6.monitoredBy→includes(sensor))))))))
endpackage
```

(b) OCL query for Mogwaï

```
return Sensor.all.select(sensor | sensor.nMonitoredSegments > 5).collect(
 sensor | sensor.monitors.
  collect(segment1: Segment | segment1.connectsTo.
   select(segment2: Segment | segment2.monitoredBy.includes(sensor)).
   collect(segment2 | segment2.connectsTo.
    select(segment3: Segment | segment3.monitoredBy.includes(sensor)).
    collect(segment3 | segment3.connectsTo.
     select(segment4: Segment | segment4.monitoredBy.includes(sensor)).
     collect(segment4 | segment4.connectsTo.
      select(segment5: Segment | segment5.monitoredBy.includes(sensor)).
      collect(segment5 | segment5.connectsTo.
      select(segment6: Segment | segment6.monitoredBy.includes(sensor)).collect(
       segment6 | Map{"sensor" = sensor, "segment1" = segment1,
        "segment2" = segment2, "segment3" = segment3, "segment4" = segment4,
        "segment5" = segment5, "segment6" = segment6}
      ))))))).flatten.asSequence;
```

(c) EOL

```
pattern ConnectedSegments
 sensor : Sensor
 in: Sensor.all.select(s|s.nMonitoredSegments > 5),
 segment1: Segment from: sensor.monitors,
 segment2: Segment from: segment1.connectsTo.select(s|s.monitoredBy.includes(sensor)),
 segment3: Segment from: segment2.connectsTo.select(s|s.monitoredBy.includes(sensor)),
 segment4: Segment from: segment3.connectsTo.select(s|s.monitoredBy.includes(sensor)),
 segment5: Segment from: segment4.connectsTo.select(s|s.monitoredBy.includes(sensor)),
 segment6: Segment from: segment5.connectsTo.select(s|s.monitoredBy.includes(sensor)) {}
```

(d) EPL

Fig. 4: Train Benchmark *ConnectedSegments* query.

```
railway::Segment.allInstances()→select(segment | segment.length <= 0)
  →collect(segment | Tuple{segment = segment})
```

(a) Original OCL

```
import railway : 'platform:/resource/railway.neoemf/model/railway.ecore'

package railway
 context Segment
  def: negLength : Set(Segment) =
   Segment.allInstances()→select(segment | segment.length <= 0)
endpackage
```

(b) OCL for Mogwaï

```
return Segment.all.select(segment |
  segment.length <= 0
);
```

(c) EOL, indexed "length"

```
pattern PosLength
  segment : Segment
    in: Segment.all.select(s|s.length <= 0) {}
```

(d) EPL, indexed "length"

Fig. 5: Train Benchmark *PosLength* query.

value). This reduces the problem to a lookup of the relevant SENSORs and a quick pattern matching to find the offending sixth SEGMENT.

- *SemaphoreNeighbor* (SN): the exit SEMAPHORE of a ROUTE must be the entry SEMAPHORE of the ROUTE that it connects to. The queries on Figure 7 find ROUTEs that are reachable from another ROUTE but do not have their SEMAPHOREs as entry point.
  There is an important difference between the original OCL and the EOL/EPL versions: Hawk can traverse a reference "x" in reverse by using "revRefNav_x", since Neo4j and OrientDB edges are navigable in both directions. This allows the query to be written more without the inefficient nested "Route.allInstances" that was required by the OCL version.
- *SwitchMonitored* (SM): every SWITCH must be monitored by a SENSOR. The queries on Figure 8 find SWITCHes that are not being monitored.
  The EOL/EPL variants use a derived attribute "isMonitored" on every SWITCH, defined as "not self.monitoredBy.isEmpty()".
- *SwitchSet* (SS): the entry SEMAPHORE of a ROUTE can only show "GO" if all SWITCHes along the ROUTE are in the same position. The queries on Figure 9 find SWITCHes that do not have the right position.
  In this case, there is only a minor change due to the fact that in Hawk, enumerated values are stored as simple strings.

```
railway::Route.allInstances()→collect(
 route | route.follows→collect(swP | swP.target→collect(
   sw | sw.monitoredBy→select( sensor | route.gathers→excludes(sensor))→collect(
    sensor | Tuple{route = route, sensor = sensor, swP = swP, sw = sw}
   ))))
```

(a) Original OCL

```
import railway : 'platform:/resource/railway.neoemf/model/railway.ecore'

package railway
 context Route
 def: routeSensor : Bag(Sensor) = Route.allInstances()→collect(route |
  route.follows→collect(swP | swP.target→collect(
    sw | sw.monitoredBy→select(sensor | route.gathers→excludes(sensor))
  )))
endpackage
```

(b) OCL for Mogwaï

```
return Route.all.collect(
 route | route.follows.collect(swP | swP.target.collect(
   sw | sw.monitoredBy.select(sensor | route.gathers.excludes(sensor)).collect(
    sensor | Map{"route" = route, "sensor" = sensor, "swP" = swP, "sw" = sw}
   )))).flatten;
```

(c) EOL

```
pattern RouteSensor
  route: Route, swP: SwitchPosition from: route.follows, sw: Switch from: swP.target,
  sensor: Sensor from: sw.monitoredBy.select(s|route.gathers.excludes(s)) {}
```

(d) EPL

Fig. 6: Train Benchmark *RouteSensor* query.

4 Results and Discussion

The previous section described the research questions to be answered, the environment that was set up for the experiment and the queries to be run. This section will present the obtained results, answer the research questions (with the help of additional data in some cases) and discuss potential threats to the validity of the work. The raw data and all related source code supporting these results are available from the Aston Data Explorer repository[24].

[24] http://dx.doi.org/10.17036/44783FFA-DA36-424D-9B78-5C3BBAE4AAA1

```
railway::Route.allInstances()→collect(
 route1 | route1.exit→collect(semaphore | route1.gathers→collect(
  sensor1 | sensor1.monitors→collect(te1 | te1.connectsTo→collect(
   te2 | te2.monitoredBy→collect(sensor2 | railway::Route.allInstances()
    →select(route2 | route2.gathers→includes(sensor2)
     and route2.entry→excludes(semaphore)
     and route1 <> route2)→collect(route2 | Tuple{
      semaphore = semaphore, route1 = route1, route2 = route2,
      sensor1 = sensor1, sensor2 = sensor2, te1 = te1, te2 = te2}))))))))
```

(a) Original OCL

```
import railway : 'platform:/resource/railway.neoemf/model/railway.ecore'

package railway
 context Sensor
  def: semNeighbor: Bag(Route) = Route.allInstances()→collect(
   route1 | route1.exit→collect(semaphore | route1.gathers→collect(
    sensor1 | sensor1.monitors→collect(te1 | te1.connectsTo→collect(
     te2 | te2.monitoredBy→collect(sensor2 | Route.allInstances()
      →select(route2 | route2.gathers→includes(sensor2)
       and route2.entry→excludes(semaphore)
       and route1 <> route2)))))))
endpackage
```

(b) OCL for Mogwaï

```
return Route.all.collect(
  route1 | route1.exit.collect(semaphore | route1.gathers.collect(
      sensor1 | sensor1.monitors.collect(te1 | te1.connectsTo.collect(
          te2 | te2.monitoredBy.collect(sensor2 | sensor2.revRefNav_gathers
            .select(r | r <> route1 and r.entry <> semaphore).collect(route2 | Map{
              "semaphore"=semaphore, "route1"=route1, "route2"=route2,
              "sensor1"=sensor1, "sensor2"=sensor2, "te1"=te1, "te2"=te2}
          ))))))).flatten.asSequence;
```

(c) EOL, uses reverse reference navigation ("revRefNav_")

```
pattern SemaphoreNeighbor
  route1: Route, semaphore: Semaphore from: route1.exit,
  sensor1: Sensor from: route1.gathers, te1: TrackElement from: sensor1.monitors,
  te2: TrackElement from: te1.connectsTo, sensor2: Sensor from: te2.monitoredBy,
  route2: Route from: sensor2.revRefNav_gathers.select(
   r | r <> route1 and r.entry <> semaphore
  ) {}
```

(d) EPL, uses reverse reference navigation ("revRefNav_")

Fig. 7: Train Benchmark *SemaphoreNeighbor* query.

```
railway::Switch.allInstances()
  →select(sw | sw.monitoredBy→isEmpty())
  →collect(sw | Tuple{sw = sw})
```

(a) Original OCL

```
import railway : 'platform:/resource/railway.neoemf/model/railway.ecore'

package railway
 context Switch
  def: notMonitored : Set(Switch) =
   Switch.allInstances()→select(sw | sw.monitoredBy→isEmpty())
endpackage
```

(b) OCL for Mogwaï

```
return Switch.all.select(
  sw | sw.isMonitored = false
);
```

(c) EOL, with derived "isMonitored"

```
pattern SwitchMonitored
 sw : Switch from: Switch.all
   .select(sw|sw.isMonitored = false) {}
```

(d) EPL, with derived "isMonitored"

Fig. 8: Train Benchmark *SwitchMonitored* query.

4.1 Measurements Obtained

The median execution times (in milliseconds) and coefficients of dispersion over 1000 executions of the GraBaTs'09 queries from Section 3.3.1 are shown on Table 1. Likewise, the results for the Train Benchmark queries from Section 3.3.2 are shown on Tables 2 to 4. To save space, Hawk with the Neo4j backend is abbreviated to "Hawk/N" and "H/N". Likewise, Hawk with the OrientDB backend is shortened to "Hawk/O" and "H/O". These abbreviations will be used throughout the rest of the paper as well.

Medians were picked as a measure of centrality due to their robustness against the occasional outliers that a heavily stressed system can produce. Coefficients of dispersion are dimensionless measures of dispersion that can be used to compare data sets with different means: they are defined as τ/η, where τ is the mean absolute deviation from the median η. Coefficients of dispersion are robust to non-normal distributions, unlike the better known coefficients of variation (Bonett and Seier, 2006). The tables allow for quick comparison of performance levels across tools, queries and number of client threads. Nevertheless, more specific visualisations and statistical analyses will be derived for some of the following research questions.

One important detail is that *SemaphoreNeighbor* was not fully run through CDO and Mogwaï, as it runs too slowly to allow for stress testing. More specifically, with only 1 client thread over TCP, the median time for the first

Query	Tool/Lang	Proto	1 thread		2 threads		4 threads		8 threads		16 threads		32 threads		64 threads	
			M	CD	M	CD	M	CD	M	CD	M	CD	M	CD	M	CD
HQ1	Hawk/N	HTTP	2 025	0.01	2 121	0.06	3 154	0.01	6 347	0.02	12 818	0.01	26 052	0.02	53 753	0.03
		TCP	1 631	0.01	1 712	0.08	2 598	0.00	5 327	0.03	10 882	0.02	22 092	0.02	45 646	0.04
	Hawk/O	HTTP	3 979	0.01	4 432	0.05	9 369	0.09	20 257	0.12	40 626	0.20	79 638	0.26	159 645	0.33
		TCP	3 491	0.01	3 991	0.06	8 623	0.09	17 692	0.16	34 296	0.30	66 547	0.39	136 348	0.42
HQ2	Hawk/N	HTTP	549	0.01	601	0.08	919	0.01	1 890	0.06	3 915	0.03	8 046	0.04	16 265	0.06
		TCP	582	0.01	631	0.08	964	0.02	1 931	0.06	3 902	0.03	8 075	0.04	16 331	0.05
	Hawk/O	HTTP	1 804	0.02	2 210	0.07	4 973	0.20	11 078	0.13	21 184	0.24	41 923	0.31	84 298	0.39
		TCP	1 715	0.01	2 102	0.07	4 745	0.17	10 253	0.19	19 103	0.28	37 723	0.35	74 815	0.39
HQ3	Hawk/N	HTTP	223	0.40	485	0.19	976	0.16	2 409	0.10	5 003	0.17	11 258	0.18	26 243	0.20
		TCP	215	0.40	485	0.19	955	0.16	2 440	0.12	5 033	0.14	11 204	0.18	26 041	0.22
	Hawk/O	HTTP	186	0.48	402	0.25	735	0.30	1 624	0.45	3 963	0.24	8 681	0.20	18 096	0.22
		TCP	181	0.54	400	0.28	710	0.31	1 664	0.41	3 838	0.27	8 454	0.19	17 640	0.21
HQ4	Hawk/N	HTTP	16	0.16	17	0.13	20	0.15	32	0.35	61	0.44	130	0.48	259	0.49
		TCP	14	0.17	15	0.12	19	0.10	32	0.33	65	0.36	132	0.38	273	0.42
	Hawk/O	HTTP	25	0.11	26	0.14	57	0.70	65	0.58	139	0.37	276	0.46	544	0.53
		TCP	24	0.06	25	0.14	56	0.86	67	0.40	140	0.34	267	0.49	523	0.55
OQ	CDO	HTTP	8 004	0.01	8 004	0.01	8 010	0.07	8 207	0.06	13 115	0.14	21 229	0.07	39 328	0.17
		TCP	1 088	0.07	1 328	0.10	2 233	0.03	5 364	0.05	10 670	0.06	21 066	0.06	38 522	0.14
MQ	Mogwaï	HTTP	5 500	0.01	5 998	0.05	8 756	0.01	21 494	0.03	64 340	0.02	108 768	0.03	215 731	0.05
		TCP	5 673	0.01	6 248	0.05	9 014	0.01	26 762	0.03	67 540	0.03	110 789	0.02	219 019	0.05

Table 1: Median execution times in milliseconds and coefficients of dispersion over 1000 executions of the GraBaTs'09 queries, by tool, language, protocol and client threads.

Query	Tool/Lang	Proto	1 thread		2 threads		4 threads		8 threads		16 threads		32 threads		64 threads	
			M	CD	M	CD	M	CD	M	CD	M	CD	M	CD	M	CD
CS	CDO	HTTP	8009	0.01	8011	0.01	8010	0.05	8122	0.06	8630	0.08	17196	0.16	29094	0.15
		TCP	1076	0.04	1197	0.07	1801	0.02	3608	0.03	7169	0.04	14408	0.05	27837	0.12
	H/N/EOL	HTTP	206	0.04	221	0.12	336	0.04	669	0.11	1378	0.06	2837	0.06	5697	0.09
		TCP	209	0.02	223	0.12	337	0.02	669	0.11	1335	0.06	2737	0.05	5647	0.08
	H/N/EPL	HTTP	246	0.02	257	0.09	389	0.02	780	0.10	1570	0.05	3187	0.04	6480	0.07
		TCP	246	0.02	257	0.08	385	0.02	761	0.10	1533	0.05	3098	0.05	6328	0.07
	H/O/EOL	HTTP	435	0.02	494	0.09	780	0.03	1614	0.10	3036	0.20	6120	0.27	11807	0.36
		TCP	420	0.02	466	0.10	761	0.03	1619	0.09	3167	0.18	6098	0.27	12000	0.33
	H/O/EPL	HTTP	486	0.02	517	0.11	828	0.03	1687	0.10	3264	0.17	6563	0.22	13305	0.27
		TCP	490	0.02	553	0.14	902	0.03	1806	0.10	3440	0.16	6627	0.24	13276	0.26
	Mogwaï	HTTP	8133	0.00	8794	0.04	12767	0.00	33478	0.03	94119	0.03	160948	0.02	319803	0.04
		TCP	7896	0.01	8844	0.04	12803	0.00	30694	0.02	85720	0.02	157047	0.02	319346	0.03
PL	CDO	HTTP	2129	0.42	2900	0.49	3392	0.74	2878	0.43	3376	0.34	8114	0.27	13058	0.17
		TCP	224	0.03	383	0.03	585	0.08	1186	0.10	2419	0.11	5024	0.12	9987	0.12
	H/N/EOL	HTTP	106	0.04	111	0.11	151	0.06	297	0.18	603	0.13	3062	0.31	7905	0.17
		TCP	110	0.04	119	0.08	157	0.06	295	0.16	602	0.12	3451	0.16	7821	0.24
	H/N/EPL	HTTP	157	0.03	166	0.09	236	0.04	478	0.15	982	0.09	4621	0.24	11081	0.19
		TCP	168	0.03	178	0.08	236	0.05	453	0.13	932	0.08	4761	0.16	10851	0.17
	H/O/EOL	HTTP	197	0.03	213	0.12	336	0.24	794	0.34	1363	0.18	3764	0.25	9126	0.29
		TCP	218	0.02	242	0.11	389	0.29	822	0.30	1437	0.16	3905	0.24	9250	0.28
	H/O/EPL	HTTP	286	0.03	303	0.10	570	0.30	1213	0.28	1989	0.15	6428	0.19	14191	0.22
		TCP	311	0.02	336	0.11	548	0.24	1221	0.25	2084	0.13	6709	0.19	14378	0.22
	Mogwaï	HTTP	285	0.03	305	0.08	449	0.04	990	0.09	2611	0.16	5128	0.17	10336	0.20
		TCP	265	0.02	286	0.07	427	0.04	949	0.09	2262	0.11	4449	0.12	8852	0.13

Table 2: Median execution times in milliseconds and coefficients of dispersion over 1000 executions of the Train Benchmark queries *ConnectedSegments* and *PosLength*, by tool, language, protocol and client threads.

Query	Tool/Lang	Proto	1 thread		2 threads		4 threads		8 threads		16 threads		32 threads		64 threads	
			M	CD	M	CD	M	CD	M	CD	M	CD	M	CD	M	CD
RS	CDO	HTTP	2893	0.17	2899	0.17	2695	0.28	2599	0.27	2652	0.12	2668	0.14	3940	0.40
		TCP	177	0.03	184	0.11	237	0.07	471	0.08	938	0.10	1872	0.12	3667	0.14
	H/N/EOL	HTTP	208	0.02	203	0.13	325	0.03	654	0.11	1321	0.06	2671	0.05	5454	0.08
		TCP	208	0.03	216	0.07	344	0.02	685	0.11	1370	0.06	2788	0.05	5697	0.08
	H/N/EPL	HTTP	235	0.02	250	0.12	382	0.02	765	0.10	1550	0.05	3121	0.05	6286	0.07
		TCP	240	0.02	251	0.08	386	0.02	760	0.10	1523	0.05	3091	0.05	6308	0.07
	H/O/EOL	HTTP	633	0.02	688	0.11	1139	0.03	2351	0.11	4618	0.17	9170	0.21	19280	0.25
		TCP	707	0.01	779	0.11	1310	0.03	2703	0.11	5328	0.15	10829	0.20	21925	0.25
	H/O/EPL	HTTP	676	0.02	744	0.11	1238	0.04	2596	0.12	5092	0.17	10494	0.21	21289	0.25
		TCP	767	0.01	859	0.10	1424	0.04	2833	0.11	5727	0.17	11191	0.20	23283	0.25
	Mogwaï	HTTP	338	0.03	371	0.10	579	0.03	1209	0.09	2575	0.05	5112	0.05	10324	0.07
		TCP	308	0.01	346	0.13	536	0.02	1114	0.08	2456	0.06	4858	0.08	9744	0.08
SM	CDO	HTTP	2103	0.20	1908	0.22	2493	0.21	2435	0.14	2443	0.10	2557	0.07	2961	0.15
		TCP	83	0.02	86	0.03	81	0.26	153	0.21	320	0.20	655	0.21	1269	0.20
	H/N/EOL	HTTP	8	0.14	8	0.16	8	0.26	10	0.36	18	0.55	31	0.80	49	0.76
		TCP	5	0.07	6	0.17	6	0.21	8	0.33	15	0.61	31	0.76	46	0.83
	H/N/EPL	HTTP	10	0.20	9	0.18	9	0.28	15	0.45	23	0.49	48	0.58	90	0.74
		TCP	7	0.25	7	0.16	8	0.35	13	0.47	19	0.71	41	0.78	74	0.85
	H/O/EOL	HTTP	11	0.18	11	0.15	15	0.51	24	0.65	44	0.46	78	0.68	143	0.88
		TCP	9	0.08	9	0.14	12	0.53	23	0.63	46	0.45	86	0.62	148	0.85
	H/O/EPL	HTTP	13	0.19	13	0.16	19	0.56	33	0.59	54	0.42	111	0.62	213	0.74
		TCP	9	0.12	11	0.17	18	0.65	29	0.63	59	0.41	118	0.56	168	0.86
	Mogwaï	HTTP	33	0.09	34	0.10	48	0.08	117	0.25	259	0.19	480	0.15	947	0.17
		TCP	33	0.05	35	0.11	47	0.06	106	0.26	240	0.27	468	0.27	939	0.53

Table 3: Median execution times in milliseconds and coefficients of dispersion over 1000 executions of the Train Benchmark queries *RouteSet* and *SwitchMonitored*, by tool, language, protocol and client threads.

Query	Tool/Lang	Proto	1 thread		2 threads		4 threads		8 threads		16 threads		32 threads		64 threads	
			M	CD	M	CD	M	CD	M	CD	M	CD	M	CD	M	CD
SN	H/N/EOL	HTTP	365	0.01	359	0.11	553	0.02	1135	0.08	2413	0.04	5810	0.10	13419	0.10
		TCP	352	0.01	370	0.07	552	0.02	1119	0.09	2285	0.05	5564	0.09	12546	0.09
	H/N/EPL	HTTP	871	0.01	917	0.10	1370	0.01	2755	0.04	5561	0.02	11216	0.03	22746	0.04
		TCP	839	0.01	878	0.11	1303	0.01	2634	0.04	5284	0.02	10619	0.03	22152	0.05
	H/O/EOL	HTTP	1054	0.01	1177	0.10	1999	0.04	3970	0.13	7990	0.22	15993	0.30	28698	0.42
		TCP	1203	0.01	1321	0.07	2128	0.04	4137	0.12	7996	0.22	16009	0.31	33188	0.39
	H/O/EPL	HTTP	2582	0.02	2834	0.06	4424	0.02	8233	0.11	17094	0.20	33895	0.29	65888	0.39
		TCP	2784	0.01	2958	0.09	4701	0.03	8747	0.12	18124	0.19	35411	0.28	67154	0.41
SS	CDO	HTTP	2103	0.21	1901	0.18	2618	0.22	2378	0.16	2243	0.11	2252	0.10	2178	0.13
		TCP	69	0.04	69	0.04	67	0.57	77	0.27	162	0.22	314	0.24	619	0.23
	H/N/EOL	HTTP	28	0.11	27	0.14	35	0.11	58	0.28	125	0.30	249	0.26	503	0.20
		TCP	28	0.08	27	0.10	34	0.07	56	0.24	116	0.23	234	0.18	463	0.20
	H/N/EPL	HTTP	36	0.06	36	0.11	48	0.06	83	0.26	176	0.26	355	0.18	718	0.17
		TCP	36	0.06	36	0.09	46	0.05	83	0.23	170	0.18	341	0.14	673	0.19
	H/O/EOL	HTTP	53	0.06	54	0.11	78	0.20	162	0.31	320	0.32	579	0.46	1173	0.53
		TCP	53	0.04	55	0.10	80	0.16	165	0.27	324	0.29	579	0.47	1149	0.55
	H/O/EPL	HTTP	72	0.04	74	0.11	106	0.09	218	0.24	423	0.27	809	0.42	1559	0.47
		TCP	71	0.05	76	0.07	106	0.07	223	0.21	429	0.25	828	0.41	1559	0.54
	Mogwaï	HTTP	55	0.08	57	0.13	85	0.15	195	0.28	464	0.16	854	0.15	1729	0.15
		TCP	53	0.04	56	0.14	81	0.06	182	0.22	436	0.21	816	0.30	1764	0.32

Table 4: Median execution times in milliseconds and coefficients of dispersion over 1000 executions of the Train Benchmark queries *SemaphoreNeighbor*, and *SwitchSet*, by tool, language, protocol and client threads.

```
railway::Route.allInstances()→collect(
 route | route.entry→select(signal = railway::Signal::GO)→collect(
  semaphore | route.follows→collect(
   swP | swP.target→select(currentPosition <> swP.position)→collect(
    sw | Tuple{route = route, semaphore = semaphore, swP = swP, sw = sw}
   ))))
```

(a) Original OCL

```
import railway : 'platform:/resource/railway.neoemf/model/railway.ecore'

package railway
 context Switch
  def: switchSet: Bag(Switch) = Route.allInstances()→collect(
   route | route.entry→select(signal = 'GO')→collect(semaphore | route.follows→collect(
    swP | swP.target→select(currentPosition <> swP.position))))
endpackage
```

(b) OCL for Mogwaï

```
return Route.all.collect(
 route | route.entry.select(e | e.isDefined() and e.signal = 'GO').collect(
  semaphore | route.follows.collect(
   swP | swP.target.select(t | t.currentPosition <> swP.position).collect(
    sw | Map{"route" = route, "semaphore" = semaphore, "swP" = swP, "sw" = sw}
   ))))).flatten.asSequence;
```

(c) EOL

```
pattern SemaphoreNeighbor
  route: Route,
  semaphore: Semaphore from: route.entry.select(e | e.isDefined() and e.signal = 'GO'),
  swP: SwitchPosition from: route.follows,
  sw: Switch from: swP.target.select(t | t.currentPosition <> swP.position) {}
```

(d) EPL

Fig. 9: Train Benchmark *SwitchSet* query.

10 runs of SN was 88.44 seconds for CDO and 305.19 seconds for Mogwaï. For this reason, only Hawk was fully evaluated regarding SN.

The next step was to check if the execution times belonged to a normal distribution for the sake of analysis. Shapiro-Wilk tests[25] rejected the null hypothesis ("the sample comes from a normal distribution") with p-values < 0.01 for almost all of the combinations of query, tool, language, protocol and thread count: only 11 out of 516 tested combinations reported p-values ≥ 0.01. In order to visualize how they deviated from a normal distribution, further manual

[25] Monte Carlo simulations have shown that Shapiro-Wilk tests have better statistical power than other normality tests (Razali and Wah, 2011).

inspections with quartile-quartile plots were conducted. These confirmed that most distributions tended to be either heavy-tailed, bimodal, multimodal, or curved.

This is somewhat surprising, as the natural intuition is that execution times should follow a normal distribution: 90% of the Java benchmarks conducted by Georges et al (2007) with single-core processors did follow a Gaussian distribution according to Kolmogorov-Smirnov tests. At the same time, 10% of those benchmarks were *not* normally distributed (being reportedly skewed or bimodal), and modern machines with multi-core processors have only grown more non-deterministic since then. More recently, Chen et al (2015) concluded that execution times for multithreaded loads in modern multicore machines do not follow neither normal nor log-normal distributions, and that more robust nonparametric methods are needed for performance comparison. Our study in particular involves 3 machines communicating over Ethernet and doing heavy disk-based I/O: even with 1 thread per client machine, the server will experience a non-deterministic multithreaded load as the one studied by Chen at al. For these reasons, the rest of this paper will assume that the query execution times are not normally distributed, and will use non-parametric tests.

Query	Tool	Proto	1t	2t	4t	8t	16t	32t	64t
OQ	CDO	HTTP						1	1
CS	CDO/H2	HTTP					1		
CS	Hawk/O/EOL	HTTP	1						
CS	Hawk/O/EPL	HTTP	2						
PL	CDO/H2	HTTP				1			
RS	Hawk/O/EOL	HTTP				1			
RS	CDO/H2	TCP							1
SN	Hawk/O/EPL	HTTP	1						
SS	Hawk/O/EPL	HTTP	1						
SS	CDO/H2	TCP							1
SS	Hawk/O/EPL	TCP					2		1

Table 5: Failed executions (timeout / server error), by query, tool, protocol and client threads ("1t": 1 thread). Only combinations with failed executions are shown.

Some of the configurations had intermittent issues when running queries. This was another goal of our stress testing: finding if the different tools would fail with increased demand and if they could recover from these errors (which they did by themselves). Table 5 shows the configurations that produced server errors, and Table 6 shows the configurations that reported the wrong number of results. The "correct" number of results is computed by running each query across all tools in local installations, without the risk of the network or the stress test influencing the result, and ensuring they all report equivalent re-

Query	Tool	Proto	1t	2t	4t	8t	16t	32t	64t
OQ	CDO	HTTP			3	8	22	17	1
CS	CDO	HTTP	2	2	1		2	3	4
CS	CDO	TCP				1	3	3	6
RS	CDO	HTTP	2	3	2	6	11	12	9
RS	CDO	TCP	6	3	3	10	15	28	32
SS	CDO	TCP				1			

Table 6: Executions with incorrect number of results, by query, tool, protocol and client threads ("1t": 1 thread). Only combinations with incorrect executions are shown.

sults. In our previous conference paper (Garcia-Dominguez et al, 2016b), incorrect executions only happened for the CDO HTTP API, but a similar issue exists even in the TCP API for some of the Train Benchmark queries. Hawk over Neo4j and Mogwaï were the only combinations of tool and backend that did not report failed or incorrect executions.

Regarding the failed and incorrect executions of CDO, at this early stage of the study we could only treat it as a black box, as we were merely users of this tool and not their developers. However, our analysis of RQ2 in Section 4.3 suggest that this is due to the stateful buffer-based design of the CDO API. As for the failed executions of Hawk with OrientDB, we attribute these problems to concurrency issues in the OrientDB backend, since Hawk with Neo4j does not report any issues and has otherwise the exact same code.

4.2 RQ1: Impact of Protocol

Tool	Query	1t	2t	4t	8t	16t	32t	64t
CDO	OQ	1.00	1.00	1.00	0.98	0.39	0.12	0.07
Mogwaï	MQ	-0.98	-0.51	-0.96	-0.99	-0.80	-0.31	-0.20
Hawk/N	HQ1	1.00	0.83	1.00	0.99	0.98	0.98	0.92
	HQ2	-0.96	-0.45	-0.91	-0.18			
	HQ3	0.25						
	HQ4	0.55	0.54	0.32				
Hawk/O	HQ1	1.00	0.73	0.40	0.51	0.34	0.26	0.19
	HQ2	0.96	0.40	0.14	0.26	0.23	0.13	0.12
	HQ3			0.09			0.08	
	HQ4	0.22	0.13					

Table 7: Cliff deltas for GraBaTs'09 query execution (-1: HTTP is faster for all pairs, 1: HTTP is slower for all pairs), for configurations where Mann-Whitney U test reports significance (p-value < 0.01), by tool, query and number of client threads ("1t": 1 thread).

Tool	Query	1t	2t	4t	8t	16t	32t	64t
CDO	CS	1.00	1.00	1.00	1.00	0.98	0.42	0.25
	PL	1.00	1.00	1.00	1.00	0.81	0.38	0.51
	RS	1.00	1.00	1.00	1.00	1.00	0.94	0.17
	SM	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	SS	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Mogwaï	CS	0.76	0.12	-0.39	0.95	0.95	0.52	0.07
	PL	0.93	0.61	0.57	0.23	0.41	0.34	0.30
	RS	0.98	0.42	0.91	0.42	0.42	0.41	0.42
	SM		-0.13		0.16	0.17		
	SS	0.37	0.12	0.29	0.14	0.14	0.12	
H/N/EOL	CS	-0.30	-0.22	-0.09		0.23	0.29	
	PL	-0.58	-0.40	-0.44			-0.30	
	RS		-0.50	-0.78	-0.19	-0.33	-0.36	-0.27
	SM	0.96	0.85	0.72	0.37	0.16	0.11	
	SN	0.89	-0.36			0.52	0.21	0.31
	SS		0.14	0.14		0.07	0.13	0.17
H/N/EPL	CS			0.22	0.09	0.24	0.28	0.15
	PL	-0.83	-0.42		0.16	0.27	-0.12	
	RS	-0.48	-0.09	-0.29		0.16	0.09	
	SM	0.65	0.66	0.33	0.19	0.16	0.09	
	SN	0.97	0.47	0.99	0.44	0.86	0.78	0.36
	SS	0.08	0.07	0.21			0.12	0.18
H/O/EOL	CS	0.74	0.48	0.41		-0.07		
	PL	-0.94	-0.56	-0.53	-0.07	-0.14		
	RS	-0.98	-0.55	-0.98	-0.55	-0.37	-0.34	-0.22
	SM	0.80	0.58	0.20				
	SN	-1.00	-0.51	-0.71	-0.16			-0.15
	SS		-0.08	-0.13				
H/O/EPL	CS	-0.24	-0.58	-0.91	-0.34	-0.10		
	PL	-0.94	-0.49	-0.11		-0.15	-0.08	
	RS	-1.00	-0.58	-0.94	-0.36	-0.31	-0.16	-0.17
	SM	0.77	0.55	0.10	0.10	-0.07		0.12
	SN	-1.00	-0.51	-0.86	-0.25	-0.11		
	SS	0.10	-0.22					

Table 8: Cliff deltas for Train Benchmark query execution (-1: HTTP is faster for all pairs, 1: HTTP is slower for all pairs), for configurations where Mann-Whitney U test reports significance (p-value < 0.01), by tool, query and number of client threads ("1t": 1 thread).

A quick glance at the results on Tables 1–4 shows that there are notable differences in some cases between HTTP and TCP, but not always: in fact, sometimes HTTP appears to be faster.

To clarify these differences, pairwise Mann-Whitney U tests (Mann and Whitney, 1947) were conducted between the HTTP and TCP results of every configuration. p-values < 0.01 were required to reject the null hypothesis that there was the same chance of HTTP and TCP being slower than the other for that particular configuration. Where the null hypothesis was rejected, Cliff deltas were computed to measure effect size (Hess and Kromrey, 2004). Cliff delta values range between +1 (for all pairs of execution times, HTTP was always slower) and -1 (HTTP was always faster). Cohen d effect sizes (Cohen, 1988) were not considered since execution times were not normally distributed. The results are summarised on Tables 7 and 8. 99% confidence intervals of

the difference between HTTP and TCP execution times were also computed during the Mann-Whitney U tests, but due to space constraints they were not included in those tables. Some of those confidence intervals will be mentioned in the following paragraphs.

CDO is the simplest case here: all tested configurations have significant differences and report positive effect sizes, meaning that HTTP was consistently slower than TCP. Cliff deltas become much weaker (closer to 0) with increasing number of threads, except for the SM and SS queries. This is also confirmed through the confidence intervals: for OQ with 1 thread, it is [+6929ms, +6944ms], while with 64 threads it is only [+78ms, +1406ms]. By comparing the medians, it can be seen that HTTP can be over 3000% slower than TCP in extreme cases, such as SS with 4 client threads.

Mogwaï has conflicting results across the two case studies. For the OQ GraBaTs'09 query, HTTP is quite often faster, though HTTP and TCP become rather similar for 32 and 64 threads with absolute values below 0.35. For the Train Benchmark queries, TCP is more often faster than HTTP, though this difference again drops off as the number of client threads increases. For SM and SS, the two faster running queries for Mogwaï in the TB case study, the difference is again very small. This suggests that for Mogwaï, in addition to the protocol used, the way concurrency is handled by the server and how it interacts with the query might have an impact as well. In particular, the Jetty HTTP server and the TCP server use different types of network I/O: non-blocking for Jetty (which decouples network I/O from request processing) and blocking for Thrift (which simplifies the Thrift message format).

The results from Hawk are the most complex to analyse. Regarding the GraBaTs'09 queries, HQ1 is consistently slower on HTTP for Neo4j and OrientDB, with strong effect sizes for all numbers of client threads. HQ2 is only slower on HTTP for OrientDB, especially with few client threads: with Neo4j, HTTP is faster for 1–8 threads. HQ3 and HQ4 sometimes run slower on HTTP, but effect sizes are weaker overall and in most cases there is not a significant difference. It appears that once queries are optimized through derived and indexed attributes, there is not that much difference between HTTP and TCP.

As for the Train Benchmark queries under Hawk, a first step is studying the Cliff deltas for each query:

- CS does not show a consistent pattern neither by backend nor by query language: effect sizes are only moderate with Neo4j (with absolute values below 0.30), and with OrientDB effect sizes are positive when using EOL and negative when using EPL.
- PL and RS are usually faster on HTTP, especially with OrientDB.
- SM on the other hand is consistently slower on HTTP. This time, the strongest effect sizes are produced when using Neo4j.
- SN is consistently slower on HTTP with Neo4j, and consistently faster on HTTP with OrientDB.
- SS effect sizes are usually positive with Neo4j and negative with OrientDB, but they are weak, with absolute values below 0.22.

From these results, it appears that the largest factor on HTTP slowdown patterns for Hawk is the chosen backend, suggesting that the interaction between the concurrency and I/O patterns of the Jetty HTTP server, the Thrift TCP server, and the Hawk backend may be relevant. While the Hawk Neo4j backend took advantage of the thread-safety built into Neo4j, the OrientDB backend has only recently implemented its own thread pooling to preserve database caches across queries. The query language was only important for CS, showing there may be an interaction but it could be relevant only for certain queries.

As for the coefficients of dispersion, the general trend is that they increase as more client threads are used. This is to be expected from the increasingly non-deterministic multithreaded load, but the exact pattern changes depending on the tool and the protocol. For most configurations, Hawk shows very similar CDs with HTTP and TCP, and so does Mogwaï (except for some rare cases such as SM and SS over 32 threads): this is likely due to the fact that the message exchanges are the same across both solutions (a single request/response pair). However, CDO shows consistently different CDs over HTTP and over TCP, suggesting that they may be fundamentally different in design from each other. This will be the focus of the next section.

4.3 RQ2: Impact of API Design

One striking observation from RQ1 was that CDO over HTTP had much higher overhead than Hawk and Mogwaï over HTTP. Comparing the medians of OQ and HQ1 with 1 client thread, CDO+HTTP took 635.66% longer than CDO+TCP, while Hawk+HTTP only took 24.16% longer than Hawk+TCP. This contrast showed that CDO used HTTP to implement their APIs very differently from the other tools.

To clarify this issue, the Wireshark packet sniffer was used to capture the communications between the server and the client for one invocation of OQ and HQ1 (with Hawk over Neo4j). These captures showed quite different approaches for an HTTP-based API:

- **CDO** involved exchanging 58 packets (10203 bytes), performing 11 different HTTP requests. Many of these requests were very small and consisted of exchanges of byte buffers between the server and the client, opaque to the HTTP servlet itself.
  Most of these requests were either within the first second of the query execution time or within the last second. There was a gap of approximately 6 seconds between the first group of requests and the last group. Interestingly, the last request before the gap contained the OCL query and the response was an acknowledgement from CDO. On the first request after the gap, the client sent its session ID and received back the results from the query.
  The capture indicates that these CDO queries are asynchronous in nature: the client sends the query and eventually gets back the results. While the

default Net4j TCP connector allows the CDO server to talk back to the client directly through the connection, the experimental HTTP connector relies on polling for this task. This has introduced unwanted delays in the execution of the queries. The result suggests that an alternative solution for this bidirectional communication would be advisable, such as WebSockets.
- **Hawk** involved exchanging 14 packets (2804 bytes), performing 1 HTTP request and receiving the results of the query in the same response. Since its API is stateless, there was no need to establish a session or keep a bidirectional server–client channel: the results were available as soon as possible.
  While this synchronous and stateless approach is much simpler to implement and use, it does have the disadvantage of making the client block until all the results have been produced. Future versions of Hawk could also implement asynchronous querying as suggested for CDO.
  One side note is that Hawk required using much less bandwidth than CDO: this was due to a combination of using fewer requests, using `gzip` compression on the responses and taking advantage of the most efficient binary encoding available in Apache Thrift (*Tuple*).

In summary, CDO and Hawk use HTTP in very different ways. The CDO API is stateful and consists of exchanging pending buffers between server and client: queries are asynchronous. This is not a problem when using TCP, since messages can be exchanged both ways. However, HTTP by itself does not allow the server to initiate a connection to the client to send back the results when they are available: to emulate this, polling is used. This could be solvable with technologies such as WebSockets, which is essentially a negotiation process to upgrade an HTTP connection to a full-duplex connection.

This stateful and buffer-based communication explains some of the intermittent communication issues that were shown for CDO in Tables 5 and Tables 6. In a heavily congested multithreaded environment, concurrency issues (race conditions or thread-unsafe code) may result in buffers being sent out of order or mangled together. If the state of the connection becomes inconsistent, it may either fail to produce a result or may miss to collect some of the results that were sent across the connection.

In comparison, the Hawk API is stateless and synchronous: query results are sent back in a single message. Since there are no multiple messages that need to be correlated to each other, this problem is avoided entirely.

These results suggest that while systems may benefit from supporting both synchronous querying (for small or time-sensitive queries) and asynchronous querying (for large or long-running queries), asynchronous querying can be complex to implement in a robust manner. Proper full-duplex channels are required to avoid delays (either raw TCP or WebSockets over HTTP) and adequate care must be given to thread safety and message ordering.

4.4 RQ3: Impact of Caching and Indexing

This section will focus on the results from the TCP variants, since they were faster or equivalent to the HTTP variants in the previous tests. It will also focus on the times in the ideal situation where there is only 1 client thread: later questions will focus on the scenarios with higher numbers of client threads.

4.4.1 GraBaTs'09 queries

A Kruskal-Wallis test reported there were significant differences in TCP execution times across tool/query combinations with 1 client thread (p-value below 0.01). A post-hoc Dunn test (Dunn, 1961) was then used to compute p-values for pairwise comparisons, using the Bonferroni correction. There was only one pairwise comparison with p-value higher than 0.01: HQ3 with Hawk/Neo4j against HQ3 with Hawk/Orient (p-value = 0.054). These two configurations will be considered to be similar in performance. All other comparisons will be based on the medians shown in Table 1.

Looking at the OQ and HQ1 times for CDO, Hawk and Mogwaï, CDO is the fastest, with a median of 1088ms compared to 5673ms from Mogwaï, 1631ms from Hawk/Neo4j and 3491ms from Hawk/OrientDB. This is interesting, as normally one would assume that the join-free adjacency of the graph databases used in Hawk and Mogwaï would give them an edge over the default H2 relational backend in CDO.

Enabling the SQL trace log in CDO showed that after the first execution of OQ, later executions only performed one SQL query to verify if there were any new instances of TypeDeclaration. Previous tests had already rejected the possibility that CDO was caching the query results. Instead, an inspection of the CDO code revealed a collection of generic caches. Among others, CDO keeps a CDOExtentMap from EClasses to all their EObject instances, and also keeps a CDORevisionCache with the various versions of each EObject. CDO keeps a cache of prepared SQL queries as well.

In comparison, Hawk and Mogwaï do not use object-level caching by default, relying mostly on the graph database caches instead. Neo4j caches are shared across all threads, whereas in OrientDB they are specific to each thread, requiring more memory. OrientDB caches can be configured to free up memory as soon as possible (weak Java references) or use up as much memory as possible (soft Java references): for this study, the second mode was used, but the authors identified issues with this particular mode. The issues were initially notified and resolved[26], but the lack of an LRU policy in the OrientDB in-house cache prompted the authors to have Hawk replace it with a standard Guava cache.

Beyond object-level caching, Hawk caches type nodes and Mogwaï caches the compiled version of the ATL script that transforms OCL to Gremlin. The ATL caching in Mogwaï was in fact added during this study, as a result of

[26] `https://github.com/orientechnologies/orientdb/issues/6686`

communication with the Mogwaï developers that produced several iterations tackling limitations in the OCL transformer, reducing query latency and resolving concurrency issues.

The above results indicate that a strong caching layer can have an impact large enough to trump a more efficient persistent representation in some situations. Nevertheless, the results of HQ2, HQ3 and HQ4 confirm the findings of our previous work in scalable querying (Barmpis et al, 2015; Barmpis and Kolovos, 2014b): adding derived attributes to reduce the levels of iteration required in a query speeds up running times by orders of magnitude, while adding minimal overhead due to the use of incremental updating. These derived attributes can be seen as application-specific caches that precompute parts of a query, unlike the application-agnostic caches present in CDO:

- HQ2 replaces the innermost loop in HQ1 with the use of pre-computed derived attributes (*isStatic*, *isPublic* and *isSameReturnType*) of a generic nature. These derived attributes produce a 2.80x speedup on Hawk/Neo4j and 2.04x speedup on Hawk/OrientDB. OrientDB receives less of a boost as following edges in general appears to be less efficient than in Neo4j.
- HQ3 uses the same attributes but rearranges the query to have them appear in the outermost "select", so Hawk can transform the iteration transparently into a lookup. Compared to HQ2, HQ3 is 2.71x faster on Neo4j and 9.47x faster on OrientDB. From the previous Dunn post-hoc test, it appears that indexing in the Hawk and OrientDB backends is similarly performant in this case.
- HQ4 uses a much more specific derived attribute (*isSingleton*) that eliminates one more level of iteration, turning the query into a simple lookup. HQ4 is one order of magnitude faster than HQ3 both on Neo4j and OrientDB, but here Hawk/Neo4j is somewhat faster. This suggests that a single index lookup is faster on Neo4j, whereas multiple index lookups are faster on OrientDB. This may be due to the way OrientDB caches index pages internally, compared to Neo4j.

4.4.2 Train Benchmark

The Train Benchmark results span over 6 queries of very different nature: some are very lightweight, while others require a more intensive traversal of the underlying graph. For each query, a Kruskal-Wallis test confirmed that there were significant differences in TCP execution times across configurations (p-value $<$ 0.01). A post-hoc Dunn test confirmed that most pairwise combinations of configurations had significant differences as well (p-value < 0.01 with Bonferroni correction), except for SwitchSet between Hawk/Orient/EOL and Mogwaï. Having established most differences in times are significant, this section will use the medians on Tables 2 and 4 to compare the tools.

To simplify the comparison, rather than using the tables directly, this section will use the more intuitive radar plots in Figure 10 to guide the discussion. Comparing the relative area of each different tool gives a general impression

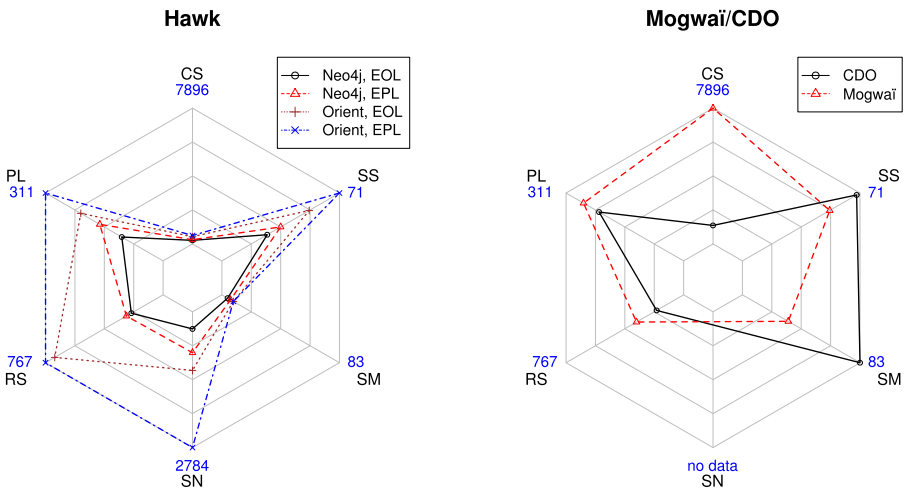

Fig. 10: Radar plot for median Train Benchmark TCP query execution times in milliseconds over 1000 executions, with 1 client thread.

of their standing: tools with smaller areas are faster in general. The Hawk side and the CDO/Mogwaï side use the same scales, to allow for comparisons across plots. CDO and Mogwaï do not have any data points for SN, since they were too slow for a full run (Section 4.1).

The Hawk side compares the relative performance of the four tested configurations (two backends, two query languages). It can be seen that the OrientDB backend is close to the Neo4j backend in some queries (CS and SM), twice as slow in most queries, and noticeably slower in RS. Examining these results suggests that while derived/indexed attributes are effective on both backends, range queries in OrientDB do not deal well with high-cardinality attributes:

- The two queries that ran in similar times (CS and SM) use custom Hawk indices: CS performs an indexed range query on a derived attribute ($nMonitoredSegments > 5$), and SM performs an indexed lookup ($isMonitored =$ false).
- However, PL is still slow even though it uses an indexed range query ($length \leq 0$), which apparently contradicts the results obtained with CS. One important difference between the queries is that there are many more distinct values of *length* (978) than of *nMonitoredSegments* (2): the indexed range query in PL will need to read many more SB-Tree nodes than in CS.

Looking at the CDO/Mogwaï side, it appears that the generic caching in CDO helped obtain good performance in PL, RS (where it slightly outperformed even Hawk with Neo4j) and CS, but it was not that useful for SM. In SM, Mogwaï can follow the *monitoredBy* reference faster than CDO, and Hawk can use an indexed lookup to fetch directly the 35 unmonitored SWITCHes instead of going through all 1501 of them. In general, it appears that CDO deals

quite well with queries that involve few types, in addition to queries with few nested reference traversals.

While Mogwaï does not support indexed attributes, its use of Neo4j through NeoEMF should have given it similar performance to that of Hawk with Neo4j through the default Neo4j caching. Instead, it is always slower than Hawk with Neo4j and EOL, and it is only faster than Hawk with OrientDB and EOL on RS. After a discussion with the Mogwaï/NeoEMF developers, it seems that this difference may be due to the use of Neo4j 1.9.6 in NeoEMF (Hawk uses 2.0.5, after testing various 2.x releases), and to inefficiencies in the bundled implementation of Gremlin.

4.5 RQ4: Impact of Mapping from Query to Backend

In a database-backed model querying solution, the query language is the interface shown to the user for accessing the stored models, and a query engine is the component that maps the query into an efficient use of the backend. Good solutions are those whose queries are easy to read and write and are mapped to the best possible use of the backend.

Since the query language, the query engine and the backend are all interrelated, it is hard to separate their individual contributions. CDO and Mogwaï use the same query language, but run it in very different ways. Likewise, Mogwaï and Hawk share a backend (Neo4j), but they store models differently and use different APIs to access it. For this reason, it is not possible to talk about what is the "best" query language in isolation of the other factors, or make other similar general statements. Instead, the answer to RQ4 will start from each source language and draw comparisons on how their queries were mapped to the capabilities of the backends, for the different tools that supported them:

- OCL is reasonably straightforward to use for queries with simple pattern matching, like OQ/MQ from GraBaTs'09 or the Train Benchmark PL and SM queries. However, it quickly becomes unwieldy with queries that have more complex pattern matching, requiring many nested select/collect invocations in cases such as SN (Figure 4 on page 17).
  CDO and Mogwaï map OCL in very different ways. CDO parses the OCL query into a standard Eclipse OCL abstract syntax tree of Java objects and evaluates the tree, providing a CDO environment that integrates caching and reads from the database as needed with multiple SQL queries. This allows it to start running the query very quickly, but it also implies that OCL queries need to switch back and forth between the H2 database layer and the model query layer, reducing performance. This may have been one of the main reasons for CDO's inclusion of an object-level cache.
  Mogwaï, on the other hand, parses the OCL query as a model, transforms it into Gremlin, compiles the Gremlin script into bytecode, executes the query entirely within Gremlin and deserialises the results back into EMF objects. This process increases query latency over an interpreted approach,

but queries could potentially run faster thanks to less back-and-forth between layers. However, as mentioned for RQ3, the use of an old release of Neo4j (1.9.6) in the current version of Mogwaï has made it run quite slow, negating this advantage over CDO and Hawk.

– EOL is inspired by OCL, and while the examples show that it is slightly more concise, it still suffers from the same nested collect/select problem when performing complex graph pattern matching. The execution approach is also similar: the EOL query is turned into an abstract syntax tree, which is visited in a post-order manner to produce the final value.
  However, the EOL-Hawk bridge (Barmpis et al, 2015) takes advantage of several features in the underlying graph database: custom indices (already discussed for RQ3) and the bidirectional navigability of the edges. It also allows for following references in reverse (from target to source), and certain queries can be written much more efficiently. This was the reason why the median time for SN was 352ms with Hawk/Neo4j/EOL and over 300s with Mogwaï. It is a missed opportunity for Mogwaï, which could have exposed this capability as well through OCL.
– EPL is a refined version of EOL which is specialized towards pattern matching. Looking at SN again, the EPL version is much easier to understand, with no explicit nesting: these nested loops are implicit in EPL's execution. Like EOL, EPL is also interpreted instead of compiled, reducing latency for some queries.
  As shown in Tables 3 and Figure 10, EPL appears to be consistently slower than EOL, even though queries are very similar. The overhead is especially notable for SN, where EPL is twice as slow as EOL. To clarify this issue, a profiler was used to follow 5 executions of the EOL and EPL versions of SN. It revealed that the additional type checking done implicitly by EPL on every match candidate was the main reason for the heavy slowdown. While this check is painless on traditional in-memory models, on the graph databases built by Hawk this check requires following one more edge and potentially performing disk I/O. Disabling this type check by referring to the "Any" root supertype in Epsilon instead returned execution times to values similar to those of EOL.

In closing, these experiences show that while query compilation may have a higher potential for performance, it may be more important to focus on selecting a stronger database technology and fully expose the strengths of this technology through the query language and the query engine. Developers wishing to repurpose existing "declarative" query languages need to test if any language features interact negatively with the chosen technology, as the cost of certain common operations may have changed dramatically.

4.6 RQ5: Scalability with Demand

The next question was concerned about how well relational and graph-based approaches scale as demand increases: one approach could do well with few

Tool	Query	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads	64 threads
Hawk/N	HQ1	0.66 - 0.67	0.70 - 0.73	0.86 - 0.86	0.20 - 0.20	0.16 - 0.16	0.95 - 0.96	0.84 - 0.86
	HQ2	1.85 - 1.87	1.91 - 2.00	2.31 - 2.33	2.76 - 2.80	2.71 - 2.75	2.60 - 2.63	2.36 - 2.42
	HQ3	4.72 - 4.81	2.70 - 2.81	2.29 - 2.36	2.19 - 2.24	2.11 - 2.16	1.84 - 1.90	1.49 - 1.57
	HQ4	76.79 - 77.45	83.21 - 84.94	115.63 - 117.08	163.65 - 172.77	156.89 - 167.44	152.44 - 162.54	138.61 - 151.56
Hawk/O	HQ1	0.31 - 0.31	0.32 - 0.33	0.25 - 0.26	0.30 - 0.31	0.30 - 0.32	0.30 - 0.33	0.29 - 0.32
	HQ2	0.63 - 0.63	0.60 - 0.62	0.46 - 0.48	0.51 - 0.53	0.54 - 0.57	0.54 - 0.57	0.52 - 0.56
	HQ3	5.64 - 5.83	3.25 - 3.39	3.07 - 3.22	3.14 - 3.26	2.72 - 2.82	2.42 - 2.52	2.20 - 2.31
	HQ4	45.51 - 45.92	48.58 - 49.78	36.84 - 41.67	78.10 - 82.31	72.66 - 77.72	73.43 - 81.44	71.80 - 80.36
Mogwaï	MQ	0.19 - 0.19	0.20 - 0.21	0.25 - 0.25	0.20 - 0.20	0.16 - 0.16	0.19 - 0.19	0.18 - 0.18

Table 9: Bounds of the 99% confidence interval for median execution time ratios between CDO and other tools (GraBaTs'09). Values greater than 1 indicate that CDO is slower, while values less than 1 indicate that the other tool is slower.

Tool	Query	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads	64 threads
H/N/EOL	CS	5.08 - 5.13	5.23 - 5.30	5.32 - 5.35	5.32 - 5.45	5.33 - 5.40	5.23 - 5.29	4.90 - 5.02
	PL	2.04 - 2.05	3.18 - 3.21	3.65 - 3.72	3.91 - 4.07	3.91 - 4.03	1.44 - 1.50	1.24 - 1.30
	RS	0.85 - 0.85	0.84 - 0.85	0.69 - 0.70	0.68 - 0.70	0.68 - 0.69	0.66 - 0.68	0.64 - 0.66
	SM	16.60 - 16.60	14.50 - 15.00	12.00 - 12.80	17.50 - 18.82	19.48 - 21.49	19.51 - 23.03	24.31 - 29.08
	SS	2.46 - 2.52	2.54 - 2.58	1.29 - 1.97	1.30 - 1.39	1.33 - 1.42	1.31 - 1.39	1.29 - 1.37
H/N/EPL	CS	4.33 - 4.37	4.55 - 4.60	4.66 - 4.69	4.68 - 4.78	4.65 - 4.71	4.62 - 4.68	4.36 - 4.46
	PL	1.33 - 1.34	2.11 - 2.14	2.42 - 2.46	2.57 - 2.65	2.57 - 2.64	1.04 - 1.08	0.91 - 0.94
	RS	0.74 - 0.74	0.73 - 0.73	0.61 - 0.62	0.61 - 0.63	0.61 - 0.62	0.60 - 0.61	0.58 - 0.59
	SM	11.71 - 11.71	11.71 - 11.86	8.50 - 9.20	11.24 - 12.36	15.30 - 17.50	14.37 - 16.70	14.84 - 17.59
	SS	1.92 - 1.95	1.92 - 1.94	0.95 - 1.47	1.0 - 1.0	1.0 - 1.0	0.90 - 0.95	0.90 - 0.95
H/O/EOL	CS	2.53 - 2.55	2.52 - 2.55	2.36 - 2.37	2.20 - 2.24	2.22 - 2.30	2.27 - 2.40	2.25 - 2.42
	PL	1.02 - 1.03	1.54 - 1.56	1.39 - 1.45	1.37 - 1.44	1.64 - 1.71	1.24 - 1.31	1.04 - 1.10
	RS	0.25 - 0.25	0.23 - 0.24	0.18 - 0.18	0.17 - 0.18	0.17 - 0.18	0.17 - 0.18	0.16 - 0.17
	SM	9.56 - 9.56	9.11 - 9.22	5.00 - 5.51	5.95 - 6.76	6.69 - 7.37	7.01 - 7.89	7.59 - 9.08
	SS	1.29 - 1.30	1.24 - 1.26	0.54 - 0.69	0.44 - 0.47	0.48 - 0.51	0.51 - 0.56	0.50 - 0.56
H/O/EPL	CS	2.17 - 2.19	2.09 - 2.12	1.99 - 2.01	1.97 - 2.00	2.05 - 2.12	2.11 - 2.21	2.05 - 2.17
	PL	0.72 - 0.72	1.11 - 1.13	1.0 - 1.0	1.0 - 1.0	1.14 - 1.17	0.72 - 0.76	0.67 - 0.70
	RS	0.23 - 0.23	0.21 - 0.22	0.17 - 0.17	0.16 - 0.17	0.16 - 0.16	0.16 - 0.17	0.15 - 0.16
	SM	9.11 - 9.22	7.64 - 7.91	3.67 - 4.08	4.90 - 5.57	5.24 - 5.76	5.09 - 5.71	6.64 - 7.90
	SS	0.97 - 0.97	0.91 - 0.92	0.41 - 0.63	0.33 - 0.35	0.36 - 0.38	0.36 - 0.39	0.38 - 0.42
Mogwaï	CS	0.13 - 0.14	0.13 - 0.14	0.14 - 0.14	0.12 - 0.12	0.08 - 0.08	0.09 - 0.09	0.09 - 0.09
	PL	0.84 - 0.84	1.32 - 1.33	1.35 - 1.37	1.23 - 1.26	1.06 - 1.09	1.10 - 1.13	1.11 - 1.15
	RS	0.57 - 0.57	0.53 - 0.53	0.44 - 0.45	0.42 - 0.43	0.38 - 0.39	0.38 - 0.39	0.37 - 0.38
	SM	2.52 - 2.53	2.41 - 2.44	1.56 - 1.68	1.40 - 1.49	1.30 - 1.38	1.36 - 1.44	1.29 - 1.39
	SS	1.30 - 1.31	1.20 - 1.22	0.53 - 0.83	0.41 - 0.44	0.36 - 0.38	0.38 - 0.40	0.34 - 0.36

Table 10: Bounds of the 99% confidence interval for median execution time ratios between CDO and other tools (Train Benchmark). Values greater than 1 indicate that CDO is slower, while values less than 1 indicate that the other tool is slower.

clients, but then quickly drop in performance with more clients. Ideally, we would simply swap relational backends with graph-based backends in each tool and do separate comparisons. Unfortunately, CDO does not include a graph-based backend, and Hawk and Mogwaï do not support relational backends. Instead, we will perform the comparison across tools, assuming that each tool was specially tailored to their backend and that therefore they are good representatives for their type of approach. These results could be revisited if new backends were developed, but they should serve as a good snapshot of their standing at the time of writing this paper.

In this section, the relational approaches will be represented by CDO (based on the embedded H2 database), and the graph-based approaches will be represented by Hawk (combined with Neo4j 2.0.5 or OrientDB 2.2.8) and Mogwaï (backed by NeoEMF, which uses Neo4j 1.9.6). CDO is one of the most mature model persistence layers and has considerable industrial adoption, so it can be considered a good representative for the relational approaches.

First, Kruskal-Wallis tests confirmed (with p-values < 0.01) that for each combination of query and client threads, TCP execution times had significant differences across the tested combinations of tool, backend and query language. Post-hoc Dunn tests were used to evaluate the null hypotheses that CDO execution times were similar to each of the non-CDO configurations (p-values < 0.01). In most cases, the null hypothesis was rejected, but there were some exceptions (2 out of 63 for the GraBaTs'09 queries, and 4 out of 175 for the Train Benchmark queries).

After confirming significant differences for most CDO vs. non-CDO pairs, the next step was quantifying how those pairs scaled relative to each other. Cliff deltas would have been able to express if a certain configuration started being faster more often than the other at a certain point, but they could not show if the gap between CDO and the non-CDO configuration increased, stayed the same or decreased together with the client threads. Instead, it was decided to use the median of the t_c/t_o ratios between random pairings of the t_c CDO TCP execution times and the t_o non-CDO TCP execution times: values larger than 1 would imply that CDO was slower, and values smaller than 1 would mean that CDO was faster. To increase the level of confidence of the results, bootstrapping over 10 000 rounds was used to estimate a 99% confidence interval of this "median of ratios" metric. The confidence intervals produced for the GraBaTs'09 and Train Benchmark queries are shown on Tables 9 and 10, respectively. Cells with "1.0–1.0" represent cases where CDO and the tool did not report significantly different times according to the Dunn tests.

In absolute terms, in most cases if a query runs faster or slower on a certain tool than on CDO, it will remain that way for all client threads. However, there are some exceptions:

- Mogwaï becomes slightly faster than CDO for the PL query with 2 or more threads, and slower than CDO for SS with 4+ threads. In fact, all non-CDO solutions experience a noticeable drop in performance for SS with 4

threads: it is just that Mogwaï did not have enough leeway to stay ahead of CDO. It appears that when running queries with no specific optimisations (e.g. indexed attributes), there may be less thread contention on CDO than on the other tools, closing the gap that originally existed in some cases.
- Hawk with Neo4j/EPL and Hawk with OrientDB/EOL start with better performance than CDO for SS, but quickly drop to similar or slightly inferior performance when using 4 or more client threads. In the first case, the additional type checks performed by EPL are weighing Hawk down. In the second case, the lower performance of the OrientDB backend gives Hawk less margin to handle the CPU saturation at 4 threads – with OrientDB/EPL, Hawk is already slightly slower than CDO with 1 thread.

One interesting observation is that depending on the combination between the query and the tool, some queries maintain a consistent ratio with CDO (e.g. OQ on Mogwaï), others raise then fall (PL for Mogwaï and Hawk), and others simply fall (RS and SS on all tools). This further supports the idea that thread contention profiles among the different tools vary notably for the same query. While further studies would be necessary to find out the specific reasons for most of these cases, there are some configurations for which it is easier to explain. The reason behind HQ4 having consistently increasing ratios for Hawk/Neo4j and Hawk/OrientDB is that it reduces multiply nested loops with a single lookup, changing the underlying order of the computation: the heavier the load, the larger the contrast created by this change.

As a general conclusion, graph databases by themselves are not a silver bullet — Mogwaï for instance did not outspeed CDO in many queries. It is important to use recent releases and take advantage of every feature at their disposal in order to achieve a solid advantage over mature relational technologies.

4.7 Threats to Validity

This section discusses the threats to the internal and external validity of the results, as well as the steps we have taken to mitigate them. Starting with the internal validity of the results, these are the threats we have identified:

- There is a possibility that CDO, Hawk or Mogwaï could have been configured or used in a more optimal way. Since the authors developed Hawk, this may have allowed them to fine-tune Hawk better than CDO or Mogwaï. However, the servers did not show any undesirable virtual memory usage, excessive garbage collection or unexpected disk I/O. The H2 backend was chosen for CDO due to its maturity in comparison to the other backends, and the Neo4j backend has consistently produced the best results for Hawk according to previous work. Mogwaï is only available for the Neo4j backend of NeoEMF, so using an alternative configuration was out of the question. The authors contacted the CDO developers regarding how to compress responses and limit results by resource, to make it more comparable with

Hawk, and were informed that these were not supported yet[27]. The authors also collaborated with the Mogwaï developers to improve performance as much as possible during the writing of the paper, contributing bugfixes and suggesting various improvements that reduced query latency.
- The queries for CDO/Mogwaï and Hawk were written in different languages, so part of the differences in their performance may be due to the languages and not the systems themselves. The aim in this study was to use the most optimized language for each system, since Hawk does not support OCL and Mogwaï and CDO do not support EOL.
  Analytically, we do not anticipate that this is likely to have a strong impact on the obtained results for CDO and Hawk as both languages are very similar in nature and are executed via mature Java-based interpreters. It may only be an issue with Mogwaï, whose OCL-to-Gremlin transformation is still a work in progress and may change when Mogwaï transitions to Neo4j 2.x.

As for whether the results can be generalised beyond this study, there are a few threats that must be acknowledged:

- This study has not considered running several different queries concurrently. While multiple configurations for Hawk have been considered (all 4 combinations of Neo4j/Orient and EOL/EPL), only one configuration was studied for CDO and for Mogwaï. The tested configurations would be quite typical in most organisations, but it would be interesting to perform studies that mix different queries running in different models concurrently, and configure Hawk and CDO with different backends, memory limits, and model sizes.
- The experiment has compared a specific set of tools: one for model repositories (CDO), one for graph-based model indexing (Hawk) and one for querying models persisted as graphs (Mogwaï on top of NeoEMF/Graph). This raises the question of whether the results could be extended to other tools of the same types.
  The first part of our answer is that this categorization was not relevant for this study: any tool could have been used as long as it provided a high-level remote querying API and relied on a database for persisting the models. CDO, Mogwaï (in combination with NeoEMF/Graph) and Hawk are three instances of these same requirements, and therefore any generalisations are backed by not one, but the three tools.
  The next part is that while some of the detailed results are specific to certain tools (e.g. comparisons between Neo4j releases), there are higher-level results which reaffirm knowledge from other areas in software engineering. For instance, RQ1 showed that HTTP's overhead was roughly constant if the message patterns were similar, and RQ2 confirmed just how much of an impact a different message pattern could have. RQ3 compared generic against application-specific caching, RQ4 discussed readability and query

[27] https://www.eclipse.org/forums/index.php?t=rview&goto=1722258
https://www.eclipse.org/forums/index.php?t=rview&goto=1722096

implementation quality, and RQ5 confirmed using a graph backend may not always bring better performance by itself. The high-level observations collected during these studies can be extended to any database-backed remote model querying solution in the future: indeed, part of our intention with this paper was to make future developers aware of these aspects.
– The results are based on two specific case studies: it could be argued that different case studies could have yielded different results. To avoid introducing bias, the authors refrained from defining custom benchmarks and instead adopted benchmarks from the existing literature. These benchmarks were picked as they covered different application areas (software engineering versus critical systems modelling), different metamodels (highly hierarchical software metamodels versus "flat" railway metamodels), and different workloads (localized pattern matching in GraBaTs'09 versus a combination of complex pattern matching and simple "all X with attribute Y meeting Z" queries in TB).
For these reasons, we argue that the 7 queries across the 2 case studies are representative of pattern matching queries on models, where we want to find elements whose state and relationships meet certain conditions. We do not expect other model querying case studies to change the results significantly. However, our case studies do not cover other model management tasks, such as code generation or model transformation: those would require their own case studies. Incidentally, Hawk did significantly speed up code generation in our previous work (Garcia-Dominguez et al, 2016a).

5 Conclusions and Further Work

This study was a largely extended version of our prior conference paper, going from 2 configurations to 6 (CDO, Mogwaï, and all 4 combinations of Hawk with Neo4j/Orient and with EOL/EPL), and adding 6 new queries written in 3 languages (OCL, EOL and EPL). This wider study confirmed some prior results, while giving a more nuanced outlook on others.

It was confirmed once more that the network protocol used had very different impact depending on how it was used: CDO once more had dramatic overheads of 600%, while Hawk and our simple HTTP server for Mogwaï had at most a 20% overhead. In fact, statistical tests showed that for the more efficient GraBats'09 queries, there was no significant difference beyond a certain number of client threads. For the Train Benchmark queries, some queries even ran faster on HTTP thanks to the more fine tuned default thread management on the Jetty HTTP server. One worrying result is that for some Train Benchmark queries, CDO showed incorrect and failed queries even over TCP — this could point to underlying thread safety or race condition issues in the framework or the networking library.

Comparing CDO/Hawk packet captures confirmed that the problem with CDO over HTTP was the naïve way in which server-to-client communications

had been implemented, which used simple polling instead of state-of-the-art approaches such as WebSockets.

Regarding caching and indexing, CDO's application-agnostic caching performed quite well in both the GraBaTs'09 and Train Benchmark queries. However, Hawk was able to outspeed CDO easily when derived and indexed attributes (a form of application-specific caching) were used, as it happened for the HQ3, HQ4, CS, PL and SM queries. The Hawk OrientDB backend did show some performance degradation when performing ranged queries on attributes with high cardinalities, however. The current version of Mogwaï did not perform as well in this regard, as it had no support for indexed attributes and does not implement a caching layer of its own: the only caching is for the compiled ATL script that transforms OCL queries into Gremlin programs. We suggest that Mogwaï should adopt one in the future.

As for the impact of the query language, it was found that Mogwaï's full recompilation of OCL into native Gremlin queries did not give it a definitive advantage over CDO's on-the-fly SQL query generation: in fact, it seemed to perform the worst among all tools, though this may have been due to the use of an older Neo4j release. The interpreted nature of EOL and EPL did not result in performance issues, but it was found that without taking the appropriate precautions, EPL would perform additional work that would result in a severe drop of performance for queries with many nested loops. Beyond the implementation approach of the language, we found that Mogwaï missed the opportunity to integrate Neo4j's ability to traverse edges in both directions into its OCL dialect: if it had done so, it would have readily outsped CDO on the SN query, as Hawk did (median was 300ms for Hawk/Neo4j/EOL compared to 100s for CDO).

Finally, 99% confidence intervals for the execution time ratios of CDO against the other configurations were computed. For the most part, tools retained their relative performance as the number of client threads increased. There were some exceptions, however: some configurations that started faster than CDO using the Mogwaï tool, the Hawk OrientDB backend or the EPL query language would become slower than CDO as the number of threads increased – the only configuration that did not show this was Hawk with Neo4j and EOL. However, even this optimal configuration could somewhat lose its performance edge against CDO in some queries: a future study comparing levels of thread contention across tools could be useful to shed light on the reasons.

In closing, this study showed that achieving high-performance and scalable remote model querying is not only a matter of choosing the right backend and using it efficiently: every other part of the system must be carefully engineered. Our ideal system would meet these requirements:

– The API should support both synchronous and asynchronous querying. Synchronous querying is more robust against high loads (as seen with Hawk and Mogwaï), since it does not require maintaining a correlation between multiple responses. Asynchronous querying, where the results are trickled

back to the client, can handle larger result sets but is hard to protect against stressful situations (as seen with CDO).
- Any server-to-client communication needed for asynchronous querying should be conducted over a real full-duplex channel rather than through polling, to avoid introducing unnecessary delays.
- To reduce roundtrip times, APIs should support running entire queries in the server rather than simply fetching individual elements to be filtered on the client. In other words, the API should include two levels of granularity: one at the query level, and one at the model element level.
- The query engine must include a caching layer, and ideally should be able to precompute the results of common subqueries.
- The query language must allow users to take advantage of important features on the backend, while not imposing unexpected work on it.
- Using a graph database can noticeably improve performance in queries that require following many references, but it is not a silver bullet: graph databases are young in comparison to relational databases, and presently their use requires more fine tuning and benchmarking.

For future work, we would like to examine scalability within a real collaborative modelling environment instead of producing synthetic loads, where a mix of queries is run concurrently according to the needs of the users over time. Another direction for future work is analysing the queries to split the work in a query efficiently between the client and the server, using the server for model retrieval and the client to transform the retrieved values. This will require balancing the reduced workload on the server with the increased network latency and transmission costs.

One more possible line of work is studying how to scale systems such as Hawk and CDO horizontally over multiple servers, either by *sharding* or splitting the data according a domain-specific criteria (e.g. Java projects in the GraBaTs'09 dataset, or subsets of the rail network in the Train Benchmark datset), or by *replicating* all the data. Sharding could be less expensive per server, but it would require breaking down queries into smaller parts and integrating the results: this could be done in the client, or in an intermediate "broker" node. Effectively, this would increase the number of requests done through the network, and it may not be worth it except for very large queries. Querying with replication would be simpler, only requiring the addition of a load balancer in front of the servers. In fact, this particular approach would be easier to study in the short term, as Hawk already has an experimental integration with the *multi-master* replication mode of OrientDB. So far it has only been used for increased availability, but increased performance could be achieved as well by developing a load balancer node that exposed the same API as current Hawk servers.

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). We would also like to thank Gwendal Daniel for his support

on the use of Mogwaï and NeoEMF. The research data supporting this publication are available on `http://dx.doi.org/10.17036/44783FFA-DA36-424D-9B78-5C3BBAE4AAA1`.

References

Bagnato A, Brosse E, Sadovykh A, Maló P, Trujillo S, Mendialdua X, De Carlos X (2014) Flexible and scalable modelling in the MONDO project: Industrial case studies. In: Proceedings of the 3rd Extreme Modeling Workshop, Valencia, Spain, pp 42–51

Barmpis K, Kolovos DS (2014a) Evaluation of contemporary graph databases for efficient persistence of large-scale models. Journal of Object Technology 13-3:3:1–26, DOI 10.5381/jot.2014.13.3.a3

Barmpis K, Kolovos DS (2014b) Towards scalable querying of large-scale models. In: Proceedings of the 10th European Conference on Modelling Foundations and Applications, pp 35–50, DOI 10.1007/978-3-319-09195-2_3

Barmpis K, Shah S, Kolovos DS (2015) Towards incremental updates in large-scale model indexes. In: Proceedings of the 11th European Conference on Modelling Foundations and Applications, DOI 10.1007/978-3-319-09195-2_3

Benelallam A, Gómez A, Sunyé G, Tisi M, Launay D (2014) Neo4EMF, a scalable persistence layer for EMF models. In: Proceedings of the 10th European Conference on Modelling Foundations and Applications, Springer, pp 230–241

Bonett DG, Seier E (2006) Confidence Interval for a Coefficient of Dispersion in Nonnormal Distributions. Biometrical Journal 48(1):144–148, DOI 10.1002/bimj.200410148

Bruneliére H, Cabot J, Dupé G, Madiot F (2014) MoDisco: A model driven reverse engineering framework. Information and Software Technology 56(8):1012–1032, DOI 10.1016/j.infsof.2014.04.007

Carlos XD, Sagardui G, Murguzur A, Trujillo S, Mendialdua X (2015) Runtime translation of model-level queries to persistence-level. In: Model-Driven Engineering and Software Development, Springer, Cham, pp 97–111, DOI 10.1007/978-3-319-27869-8_6

Chen T, Guo Q, Temam O, Wu Y, Bao Y, Xu Z, Chen Y (2015) Statistical Performance Comparisons of Computers. IEEE Transactions on Computers 64(5):1442–1455, DOI 10.1109/TC.2014.2315614

Cohen J (1988) Statistical Power Analysis for the Behavioral Sciences, 2nd edn. Routledge, ISBN 0-8058-0283-5

Daniel G, Sunye G, Cabot J (2016) Mogwaï: A framework to handle complex queries on large models. In: Proceedings of the IEEE 10th International Conference on Research Challenges in Information Science, IEEE, Grenoble, France, pp 1–12, DOI 10.1109/RCIS.2016.7549343

Dunn OJ (1961) Multiple Comparisons Among Means. Journal of the American Statistical Association 56(293):52, DOI 10.2307/2282330

Garcia-Dominguez A, Barmpis K, Kolovos DS, da Silva MAA, Abherve A, Bagnato A (2016a) Integration of a graph-based model indexer in commercial modelling tools. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, ACM Press, Saint Malo, France, pp 340–350, DOI 10.1145/2976767.2976809

Garcia-Dominguez A, Barmpis K, Kolovos DS, Wei R, Paige RF (2016b) Stress-Testing Centralised Model Stores. In: Proceedings of the 12th European Conference on Modelling Foundations and Applications, Springer, Vienna, Austria, pp 48–63, DOI 10.1007/978-3-319-42061-5_4

Garmendia A, Guerra E, Kolovos DS, de Lara J (2014) EMF Splitter: A Structured Approach to EMF Modularity. In: Proceedings of the 3rd Extreme Workshop at the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems, Valencia, Spain, vol 1239, pp 22–31, URL http://ceur-ws.org/Vol-1239/xm14_submission_3.pdf

Georges A, Buytaert D, Eeckhout L (2007) Statistically rigorous Java performance evaluation. ACM SIGPLAN Notices 42(10):57–76

Gómez A, Tisi M, Sunyé G, Cabot J (2015) Map-based transparent persistence for very large models. In: Egyed A, Schaefer I (eds) Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol 9033, Springer Berlin Heidelberg, pp 19–34, DOI 10.1007/978-3-662-46675-9_2

GraBaTs (2009) 5th International Workshop on Graph-Based Tools. URL http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/, last checked: November 14th, 2016.

Hess MR, Kromrey JD (2004) Robust confidence intervals for effect sizes: A comparative study of Cohen's d and Cliff's delta under non-normality and heterogeneous variances. In: Annual Meeting of the American Educational Research Association, pp 1–30, URL http://www.coedu.usf.edu/main/departments/me/documents/cohen.pdf

Koegel M, Helming J (2010) EMFStore: a model repository for EMF models. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ACM, vol 2, pp 307–308, DOI 10.1145/1810295.1810364

Kolovos DS, Paige RF, Polack FA (2008) Scalability: The holy grail of model driven engineering. In: Proceedings of the 1st Int. Workshop on Challenges in Model-Driven Software Engineering, Toulouse, France, pp 10–14, URL http://ssel.vub.ac.be/ChaMDE08/_media/chamde2008_proceedingsd121.pdf, last checked: November 14th, 2016.

Kolovos DS, Rose L, Garcia-Dominguez A, Paige R (2016) The Epsilon book, chap The Epsilon Pattern Language (EPL), pp 153–164. URL http://www.eclipse.org/epsilon/doc/book/, last checked: November 14th, 2016.

Kramler G, Kappel G, Reiter T, Kapsammer E, Retschitzegger W, Schwinger W (2006) Towards a semantic infrastructure supporting model-based tool integration. In: Proceedings of the 2006 International Workshop on Global Integrated Model Management, ACM, New York, NY, USA, GaMMa '06, pp 43–46, DOI 10.1145/1138304.1138314

Mann HB, Whitney DR (1947) On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. The Annals of Mathe-

matical Statistics 18(1):50–60, DOI 10.1214/aoms/1177730491

Mohagheghi P, Gilani W, Stefanescu A, Fernandez MA (2012) An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. Empirical Software Engineering 18(1):89–116, DOI 10.1007/s10664-012-9196-x

Pagán JE, Cuadrado JS, Molina JG (2013) A repository for scalable model management. Software & Systems Modeling 14(1):219–239, DOI 10.1007/s10270-013-0326-8

Rath I, Varró D (2016) Prototype tool for collaboration. Project deliverable D4.4, Budapest University of Technology, Budapest, Hungary, URL `http://www.mondo-project.org/`

Razali NM, Wah YB (2011) Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. Journal of statistical modeling and analytics 2(1):21–33, iSBN 978-967-363-157-5

Scheidgen M (2013) Reference representation techniques for large models. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, ACM, Budapest, Hungary, DOI 10.1145/2487766.2487769

Sottet JS, Jouault F (2009) Program comprehension. In: Proceedings of the 5th International Workshop on Graph-Based Tools, URL `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2009reverseengineering.pdf`, last checked: November 11th, 2016.

Steinberg D, Budinsky F, Paternostro M, Merks E (2008) EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional, ISBN 978-0321331885

Szárnyas G, Izsó B, Ráth I, Varró D (2017) The Train Benchmark: cross-technology performance evaluation of continuous model queries. Software & Systems Modeling DOI 10.1007/s10270-016-0571-8

Wei R, Kolovos DS, Garcia-Dominguez A, Barmpis K, Paige RF (2016) Partial loading of XMI models. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, ACM Press, Saint Malo, France, pp 329–339, DOI 10.1145/2976767.2976787