Hawk solutions to the TTC 2018 Social Media Case

Antonio García-Domínguez
SARI, SEA Research Group
Birmingham, B4 7ET
a.garcia-dominguez@aston.ac.uk

Aston University

Abstract

The TTC 2018 Social Media Case required answering queries about social networks, where people write posts, comment on them, and friend or unfriend each other. NoSQL databases have been popular in the analysis of large social networks, and the Hawk heterogeneous model indexer can turn the models in the case to Neo4j NoSQL databases. This paper presents three solutions that were developed on top of each other, reducing on each step the amount of work required to update the results of the query.

1 Introduction

The TTC 2018 Social Media Case focuses on the challenge of keeping query results up to date as the underlying models change. The specific models are based on data from the 2016 DEBS Grand Challenge, adapted from the LDBC Social Network Benchmark. The case required answering two model queries, finding the top 3 "controversial" posts in the first one and finding the top 3 "influential" comments in the second one.

The case provided a benchmarking framework, with models of various sizes in XMI format. For each model size, there was an initial model instance file and a list of change sequence files. The benchmark required loading the initial model instance, creating an initial result ("view"), and then for each change sequence file, applying its changes to update the results. Solutions were evaluated based on correctness and speed.

This paper presents three solutions based on the Hawk heterogeneous model indexing framework, which is designed to speed up queries on large repositories of interconnected file-based models. Hawk can monitor a set of storage locations (e.g. local folders, SVN/Git repositories or web addresses) and mirror their models into a single graph database, which can be queried more efficiently than simply loading the various fragments. More details about Hawk have been discussed in our prior works [GDBK$^+$17]: this paper will only focus on the key details needed to understand the solution.

The initial solution is a straightforward use of the existing capabilities in Hawk for XMI-based models. The second solution uses the explicit deltas between model versions to speed up the updating of the graph. The third solution uses graph change listeners to speed up the recomputation of the results.

The rest of the paper is organized as follows. Section 2 presents the high-level architecture of Hawk, Section 3 describes the three implemented solutions, Section 4 provides metrics for the various results and Section 5 offers some conclusions and lines of future work for Hawk that were highlighted by this case study.

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

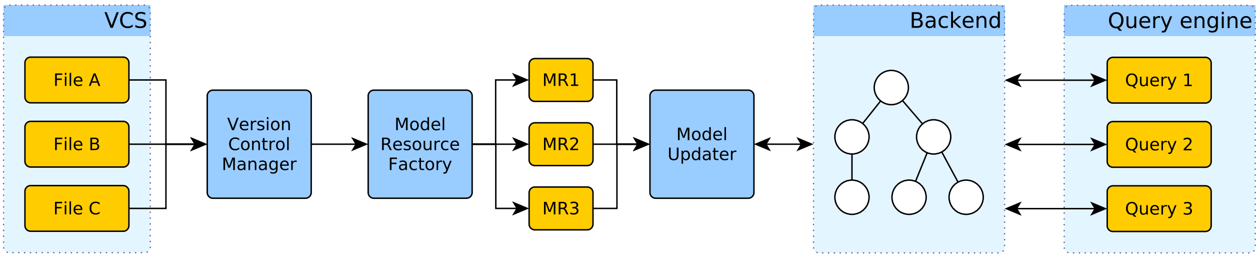


Figure 1: Hawk indexing workflow, by components

2 Hawk

Hawk implements a component-based workflow in order to support multiple modelling technologies, database backends, and query languages. The architecture is shown on Figure 1:

1. A *version control manager* or "connector" component detects whether any files were changed in each location.
2. The *model resource factory* loads the changed files as Hawk model resources.
3. The *model resources* are given to the *model updater*, which manipulates the *backend* so it reflects the latest state of the model resources. This is done incrementally, as discussed in prior work [BSK15].
4. The backend (Neo4j in our solutions) is manipulated through the Hawk graph database abstraction layer.
5. Users can ask questions through a *query engine*, by default a dialect of the Epsilon Object Language (EOL) [KPP06].

3 Implemented solutions

Three solutions were implemented in sequence, building off from each other: we will discuss their details.

3.1 Naive update and query (NU)

Listing 1: Full query for controversial posts (Q1)

```
var scored = Post.all.collect(p|Sequence {p.id, p.score(), p.timestamp}).asSequence;
Native('java.util.Collections').sort(scored, new Native('org.hawk.ttc2018.queries.ResultComparator'));
return scored.subList(0, scored.size.min(3));

operation Submission score() : Integer {
  return self.closure(p|p.comments).flatten.collect(p | 10 + p.likedBy.size).sum();
}
```

The first solution was a relatively direct application of Hawk. Hawk was told about the social network and change sequence metamodels, then told to watch the specific file with the initial model instance (which was reset to its original contents between runs) and mirror the file into a Neo4j graph.

The queries for controversial posts and influential comments were first developed in their full form as in Listings 1 and 2. A custom Java-based comparator had to be written (*ResultComparator*) so EOL would sort the results as expected by the Social Media case expected (highest scores first, then most recent).

Each full query was then split into two parts: score computations would be registered as derived "score" attributes of posts and comments (pre-computed and revised incrementally [BSK15]), and rankings would be regular queries that took advantage of the pre-computed scores. Listing 3 defines the derived *Post* "score" attribute, while Listing 4 uses the derived attribute to rank the posts. Influential posts were handled similarly.

Change sequences were interpreted by a separately implemented EOL script, which would update the initial model instance file in place. Hawk was told to check the initial model instance file for changes and update the graph database. After this, the query was run again, using the incrementally pre-computed scores.

Listing 2: Excerpt of full query for influential comments (Q2)

```
var scored = Comment.all.collect(c | Sequence { c.id, c.score(), c.timestamp }).asSequence;
Native('java.util.Collections').sort(scored, new Native('org.hawk.ttc2018.queries.ResultComparator'));
return scored.subList(0, scored.size.min(3));

operation Comment score(): Integer {
  /* shared state variable declarations */
  for (user in self.likedBy) {
    if (not indexes.containsKey(user.id)) {
      user.strongconnect(self, /* state variables */);
    }
  }
  return components.collect(component | component.size * component.size).sum();
}

-- Tarjan's strongly connected components algorithm
operation User strongconnect(comment: Comment, ...) {
  /* ... */
}
```

Listing 3: Q1: derived attribute definition

```
return self.closure(p|p.comments).flatten.collect(p | 10 + p.likedBy.size).sum();
```

Hawk needed some minor improvements to solve this problem. A new version control manager was implemented so it could watch a single file instead of a folder, and date indexing was changed to a more machine-parsable format (ISO 8601). There was an unexpected scalability issue in the default way that EMF loads XMI files: ID-based intra-model references were heavily slowing down the process, as EMF was repeatedly navigating the entire model to find a match for the target of each reference. This was solved with a custom EMF resource factory that enabled intrinsic ID→EObject maps and that deferred reference resolution until after loading.

3.2 Incremental update (IU)

For very large models, implementing the "update" phase as a reload and rewrite of the model instance file could be very slow. To minimize disk I/O, a custom model updater component was created (267 lines of Java), which would notice new change sequence files, load those change sequences, and update the backend directly without touching the initial model instance file. Hawk was made (in around 130 more lines of Java) to watch the entire folder with the instance file and the change sequence files, with a file filter that would only see the initial model instance, and would later allow it to see the change sequence files in the expected order.

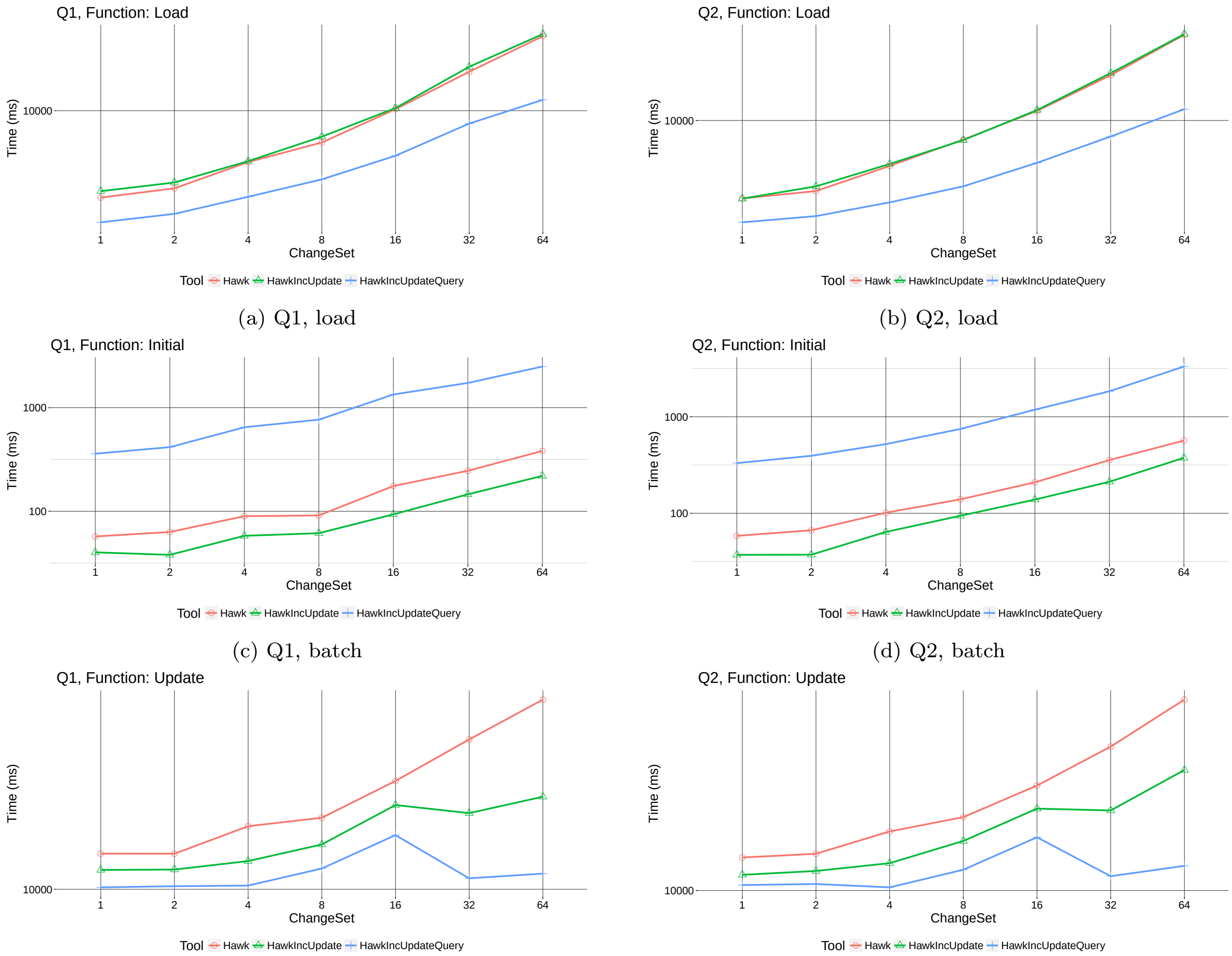
Other than that, the approach for querying was the same as in Section 3.1: using derived attributes to pre-compute scores incrementally, then running the queries leveraging them after each update.

3.3 Incremental update and query (IUQ)

While pre-computing a derived attribute value, Hawk will track which graph nodes are visited, so it will know which values need to be recomputed when the graph changes. This has some I/O overhead, as it implies creating special edges and manipulating indexes.

Listing 4: Q1: derived attribute-based query

```
var scored = Post.all.collect(p | Sequence {p.id, p.score, p.timestamp}).asSequence;
Native('java.util.Collections').sort(scored, new Native('org.hawk.ttc2018.queries.ResultComparator'));
return scored.subList(0, scored.size.min(3));
```



(a) Q1, load

(b) Q2, load

(c) Q1, batch

(d) Q2, batch

(e) Q1, updates

(f) Q2, updates

Figure 2: Average total execution times by stage and model size for the Hawk solutions

As an experiment, it was decided to replace the use of derived attributes with a custom *graph change listener* component that would be notified of any changes in the model and invoke the EOL score() operation on any posts and comments it deemed necessary, revising the top 3 rankings as required. For instance, if a post had a comment added, its score would need to be revised. People friending and unfriending each other would also trigger the recomputation of the scores of their comments. This logic was manually designed after careful study of Q1 and Q2. This approach took around 230 lines of Java.

Generally, updating the rankings would require keeping all the scores in memory and rearranging elements as their scores went up and down. However, the change sequences included in the case never reduce scores: comments are never deleted, and people do not unfriend each other. This simplified matters to keeping the top 3, checking if re-scored elements were already in the top 3, or if they should push a previous element out.

4 Results

During the contest, the benchmarking scripts included in the case were run, and some initial results were produced and presented. However, since then, Hawk has been optimised and several bugs in our original solutions have been fixed (e.g. Q2 had an incorrect algorithm for detecting strong components). This section presents our latest results at the date of this writing.

The scripts have been run again with the 1.2.0.20180628 interim version of Hawk (commit with SHA1 799722 on its GitHub repository[1]), using Oracle Java 1.8.0_102 in an Ubuntu 18.04.1 LTS installation with the Linux 4.15.0-33-generic kernel. The machine is a Lenovo Thinkpad X1 Carbon laptop with an i7-6600U CPU and 16GiB

[1] `https://github.com/mondo-project/mondo-hawk`, pending move to Eclipse.org.

of RAM, using a solid-state disk. The scripts checked that all initial and updated results were the expected ones. Figure 2 collects the various plots that were produced by the benchmarking scripts included in the case. These show the average total times for the three key stages in the benchmark: loading the model, running the initial query, and updating the results through 20 change sequences. 5 runs were used initially for NU/IU/IUQ, and later 2 more runs of NU were executed to replace outliers produced by unexpected delays in some queries[2].

The obtained results are in line with our expectations, given that Hawk uses disk-backed data structures for its querying, unlike other solutions that are entirely in-memory. As for the various stages:

- Loading takes about the same time for NU and IU (31.8s/32.1s on average for size 64 and Q2, respectively): they perform almost the same work. IU also indexes a few types by "id", but this is not that significant. IUQ does not use derived attributes, so it is faster here (11.65s on average for size 64 and Q2).
- While NU and IU run the same initial query, IU is slightly faster than NU (0.57s/0.38s on average for size 64 and Q2, respectively). Indexed attributes seem to have a side effect in Neo4j that speeds up the query. In fact, we tried 5 runs of Q1 on size 1 while postponing indexed attribute registration to the first update, and IU became as fast as NU in the Initial stage. Reverting the change made IU go faster again.

  IUQ is the slowest for this first query, since it does not take advantage of any pre-computed scores. Still, the performance hit (3.3s on average for size 64 and Q2) is less than the hit NU/IU take on the Loading stage. It is a choice between speeding up later queries and being able to start querying as soon as possible.
- For the Updates stage, NU is slower than IU, which is slower than IUQ. For size 64 and Q2, NU spends 74.1s in total updating results over the 20 change sequences, while IU takes 26.6s and IUQ only takes 11.7s.

  There is a spike in size 16 for Q2. This change sequence happens to trigger the recomputation of many scores in our graph change listener. This is confirmed by a similar but smaller spike happening also in IUQ.

5 Conclusions

Three solutions were created using Hawk for the TTC 2018 Social Media case. The naive update (NU) solution provides new results by applying the changes to the original file, updating the Hawk graph and the pre-computed scores, and running the query again. The incremental update (IU) solution applies the changes to the graph directly, without touching the original file, saving I/O, and its use of indexed attributes has beneficial side effects. The incremental update and query (IUQ) solution does not use derived attributes, but instead takes advantage of our manual impact analysis of Q1 and Q2 and listens to changes in the graph that should trigger the recomputation of the score of a post or comment.

As we expected, IUQ was faster than IU, which was faster than NU. However, it was interesting to see just how much lighter could "incrementality-by-static-analysis" (IUQ graph change listeners) be over "incrementality-by-tracking" (IU derived attributes). The difference in speed when updating the graph directly from deltas was remarkable: it may be worth extending Hawk so it can understand EMF change notifications and react accordingly. Finally, we are also considering the idea of simplifying the use of queries with derived attributes, by packaging them together into "analysis bundles" of sorts. This could have been useful for NU and IUQ.

References

[BSK15] Konstantinos Barmpis, Seyyed Shah, and Dimitrios S. Kolovos. Towards Incremental Updates in Large-Scale Model Indexes. In Gabriele Taentzer and Francis Bordeleau, editors, *Modelling Foundations and Applications*, number 9153 in Lecture Notes in Computer Science, pages 137–153. Springer International Publishing, July 2015.

[GDBK+17] Antonio Garcia-Dominguez, Konstantinos Barmpis, Dimitrios S. Kolovos, Ran Wei, and Richard F. Paige. Stress-testing remote model querying APIs for relational and graph-based stores. *Software & Systems Modeling*, pages 1–29, June 2017.

[KPP06] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Object Language (EOL). In *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, pages 128–142, 2006.

[2] We are unsure on whether these were due to disk activity from other processes or a rare race condition in a part of Hawk: this only happened 4 times across all 42 total runs of NU (7 runs × 6 model sizes). Further investigation is underway.