An NMF solution to the TTC 2018 Social Media Case

Georg Hinkel

Am Rathaus 4b, 65207 Wiesbaden, Germany
georg.hinkel@gmail.com

Abstract

This paper presents a solution to the Social Media benchmark, used as live contest at the Transformation Tool Contest (TTC) 2018. We demonstrate how the implicit incrementalization abilities of NMF can be used to automatically obtain an incremental algorithm for the presented case. We evaluate the solution against the reference solution and show that the incremental change propagation is faster than batch recomputation by multiple orders of magnitude for large models.

1 Introduction

If the knowledge of a system is captured in a formal model, one often wants to analyze such models in order to conclude properties of the underlying system. To be meaningful, the analysis results must be synchronized with the model. However, in environments where changes happen frequently, it is often not viable to recalculate the entire analysis in the presence of local changes. Rather, it is desirable to propagate these changes to the analysis results incrementally, i.e. only recalculate those parts of the analysis results that are affected by a given change.

One environment where changes happen frequently are social networks, as used by the Social Media benchmark, used as live contest of the Transformation Tool Contest 2018 [1]. In this benchmark, solutions are asked to analyze a model of a social network for two queries. While one query is rather simple in terms of the query operators used in the reference solution, the second query is more complex as graph algorithms are used.

Graph algorithms are difficult from an incrementalization point of view, because they are typically described in an imperative pseudo-code that usually utilizes state. Using a generic incrementalization system, this often spans a state-space that is too large for an incrementalization to be efficient. To aid this situation, dynamic algorithms are known for a number of graph problems that offer strategies to propagate graph changes efficiently, often with a radically different approach than in the batch (or initial) scenario (cf. [2]).

In particular, the social media benchmark exercises the problem of finding strongly connected components in a graph. The reference solution uses Tarjan's algorithm [3] for this. Therefore, it is a major challenge of the benchmark for solutions to specify how the strongly connected components should be computed incrementally while preserving a concise notation, ideally as in the reference solution.

In this paper, we present a solution to the social media benchmark using *NMF Expressions* [2], [4], an extensible incrementalization system part of the .NET Modeling Framework (*NMF*, [5], [6]). In particular, the solution demonstrates the extensibility of *NMF Expressions* as we extend the incrementalization system with a dynamic algorithm for finding strongly connected components.

The remainder of this paper is structured as follows: Section 2 gives a brief overview on NMF Expressions. Section 3 describes our solution. Section 4 evaluates our solution with respect to the reference solution and other initial solutions from the live contest. Section 5 gives a short conclusion.

In: A. Garcia-Dominguez, G. Hinkel and F. Krikava (eds.): Proceedings of the 11th Transformation Tool Contest, Toulouse, France, 29-06-2018, published at http://ceur-ws.org

2 Incrementalization with NMF Expressions

The goal of *NMF Expressions* is to give developers an automated tool at hand providing them with advantages of incremental evaluation for arbitrary expressions. [4], [7] Unlike many other approaches, our approach works implicitly, so developers only have to specify their analyses and *NMF Expressions* takes care of how to turn this into an algorithm that will evaluate the analysis in an incremental fashion. On the other hand, the traditional batch mode specification is still available so that *NMF Expressions* yields a choice whether to run a given expression incrementally or in batch mode.

In the incremental mode, the approach creates a dynamic dependency graph (DDG) from a given expression and observes changes. These changes originate from elementary update notifications and are propagated through the dependency graph.

The high abstraction level in the DDG is achieved by a manual incrementalization of analysis operators yielding valid results as a consequence of the underlying formalization as a categorial functor. *NMF Expressions* includes a library of such manually incrementalized operators, including most of the Standard Query Operators (SQO)[1], but developers are free to add new operators by simply annotating a method with an incremental implementation.

3 Solution

We describe the solutions to the two queries individually in separate sections.

3.1 Query 1: Most controversial posts

Meanwhile *NMF Expressions* includes an incrementalization of most SQOs, the `Take` method used in the reference solution for query 1 is not supported, same as any other query operators of the SQO that deal with indices. The reason for this is that the current implementation of the SQOs is mostly ignorant to collection indices and generally reports -1 as index of any change, meaning that clients would have to find out the index themselves, if they need the index of a collection. The reason for this is that it is not trivial and it is costly to find out the index of an element in a filtered list, given the index in the source collection. This always requires a linear scan, meanwhile the SQO implementations generally try to propagate changes in constant or near constant time (logarithmic effort for updating sorted lists). Therefore, the fact that `Take` is not supported is simply because it cannot be implemented efficiently, at least not in the general case.

However, we identified analyses that sort elements for a given criteria and then report on a small number of elements with the best scores as a rather common pattern. Therefore, we added dedicated support for this kind of analysis operators into *NMF*. This operator is called `TopX`. It is essentially a combination of an incremental sort (using balanced binary search trees) and a simple poll for the first `x` elements upon any change of the balanced binary search tree, assuming that `x` is small (in comparison to the size of the search tree).

```
1 query = Observable.Expression(() =>
2   SocialNetwork.Posts.TopX(3, post => ValueTuple.Create(post.Descendants().OfType<Comment>().Sum(c => 10 + c.LikedBy.Count), post.
        Timestamp))
3 );
```

Listing 1: Incremental NMF Solution for query 1

The solution to query 1 is shown in Listing 1. We simply calculate the topmost three posts where the score of a post is calculated as a tuple of the score and the timestamp in order to break ties. The result of the lambda expression in Line 2 is an array of tuples of the posts and their scores (actual score and timestamp).

Lines 1 and 3 surround this lambda expression with a call to *NMF Expressions* to obtain an incrementalization of this analysis. With that call, we tell *NMF Expressions* to create a DDG for us. The return value of this function is the root node of this DDG. This node implements a generic interface `INotifyValue<>` that provides the current analysis result as well as an event to notify clients when the analysis result changes. In the benchmark solution, we do not make use of this event but repeatedly query the current value of the DDG node. This call is fast, because each node in the DDG always references its current value.

[1] `http://msdn.microsoft.com/en-us/library/bb394939.aspx`

3.2 Query 2: Most influential comment

The solution for query 2 is very similar to the solution of query 1, except for the fact that it involves finding strongly connected components in a graph where there is no incremental implementation available that is built into *NMF*.

From an algorithmic point of view, the incrementalization of the strongly connected components will be rather simple: We cache the current set of strongly connected components. Whenever an edge is added to the graph between two nodes that are already in the same strongly connected component or an edge is removed between nodes that are in different strongly connected components, the change is not propagated because it does not influence the strongly connected components. In all other cases, the cached set of strongly connected components is discarded and recalculated entirely. To get notified of graph updates, we keep a DDG node for each incident edges of a node as well as a DDG node for the set of all nodes. To simplify the problem a bit, we assume that the references to these collections will be static[2].

To implement an incremental version of the algorithm for strongly connected components, we need to implement a class that describes this algorithm and implements the `INotifyEnumerable` interface. This interface denotes a collection-valued node of the DDG. To implement this interface, we need to provide implementations for three methods.

The first is a set of nodes `Dependencies` that the DDG node for the incremental computation of strongly connected components depends upon. That is, changes of (the values of) these nodes will cause a recalculation of the incremental strongly connected components. In our case, this is for each node in the graph the set of incident edges and the base set of nodes in the graph.

The second is an iterator of the current values. In our case, these values are strongly connected components and are thus collections themselves. To implement this, we simply keep a reference to the most up to date strongly connected components and return those.

Third, we need to implement the actual change propagation in a method called `Notify`. This method receives a list of changes in dependent DDG nodes as a parameter and returns a change notification of its own value, or a singleton instance if nothing has changed. In this method, we check whether these dependent changes affect the cached set of strongly connected components. If this is the case, we simply recompute all connected components.

The result is a class that is generically reusable for the calculation of strongly connected components in incremental analyses. As such, it can be separated in an algorithmics library and reused whenever an analysis requires the calculation of strongly connected components.

With this algorithmics class, we can solve query 2 as in Listing 2.

```
1 Func<IComment, Func<IUser, IEnumerableExpression<IUser>>> friendsBuilder = c => (u => u.Friends.Intersect(c.LikedBy));
2 query = Observable.Expression(() =>
3     SocialNetwork.Descendants().OfType<IComment>().TopX(3, comment => ValueTuple.Create(
4         ConnectedComponents<IUser>.Create(comment.LikedBy, friendsBuilder(comment))
5             .Sum(group => Squared(group.Count())),
6         comment.Timestamp
7     ))
8 );
```

Listing 2: Incremental NMF Solution for query 2

In Line 1, we create a function that given a comment, creates the incident edges for a user. Because we do not care about changes of the incidents function, the function to get the connected users is used as compiled code, using the `Func` class. This is slightly more efficient than the format of lambda expressions that *NMF Expressions* can use for the incrementalization. However, *NMF Expressions* currently has some problems to integrate compiled lambda expressions, so we need to specify this function separately, outside the scope of *NMF Expressions*.

In Lines 2 and 8, we frame the actual analysis with *NMF Expressions*, allowing it to create the DDG for the inner analysis that is in Lines 3-7. In particular, we first iterate all elements in the social network model and filter for comments in Line 3. From these, we pick the topmost elements according to the tuple of scores and timestamps, similar to the solution of query 1. To calculate the score of a post, we simply run the analysis of connected components where the incident nodes of a given user are the subset of his friends that also liked that comment (Line 4). Given these strongly connected components, we calculate the sum of the squared sizes (Line 5).

[2]Here we allow mutable collections. In the example, the incident edges of a user are always the set of his friends intersected with the users who liked a specific comment. The contents of this collection may change, but the collection itself does not.

3.3 Transactions and Parallelism

NMF Expressions has some support for transactions and parallelism. The support for transactions means that a DDG node is only processed if all of its dependencies have been processed. Because each transaction may invalidate a different set of DDG nodes, this implies that changes in the DDG need to be processed in a two-pass fashion: In the first pass, the set of potentially affected DDG nodes is calculated while in the second pass, the changes are actually propagated. Because the transactional behavior guarantees that each DDG node is only updated at most once in each transaction, the overhead of two passes may be saved. However, whether or not this is the case largely depends on the analysis and the change sequence.

The beauty of the transactional support in *NMF Expressions* is that it is very easy for a developer to use it: All that needs to be done is simply to put the changes inside a transaction. Because in the scope of the benchmark, these changes come in a dedicated change sequence object, all we need to change for transactional behavior is to wrap the application of such a change sequence in a transaction as done in Listing 3.

```
1 ExecutionEngine.Current.BeginTransaction();
2 changes.Apply();
3 ExecutionEngine.Current.CommitTransaction();
```

Listing 3: Wrapping the application of the change sequence in a transaction

Lastly, *NMF Expressions* also allows to propagate the changes within such a transaction in parallel, a feature that is still under development. This is done by changing the static execution engine implementation. However, the results show that the parallel execution had some problems with query 2 and the solution ran into exceptions.

4 Evaluation

The following measurements have been performed on an Intel i7-8550U clocked at 1.8GHz in a system with 8GB RAM running Windows 10. We only depicted the times to propagate changes as *NMF Expressions* clearly targets the incremental case. The results for Query 1 are depicted in Figure 1. The results for Query 2 are depicted in Figure 2. Both figures show the average time to propagate one of 20 change sequences in a model of different sizes. Note that both the model size axis as well as the time axis are logarithmic.

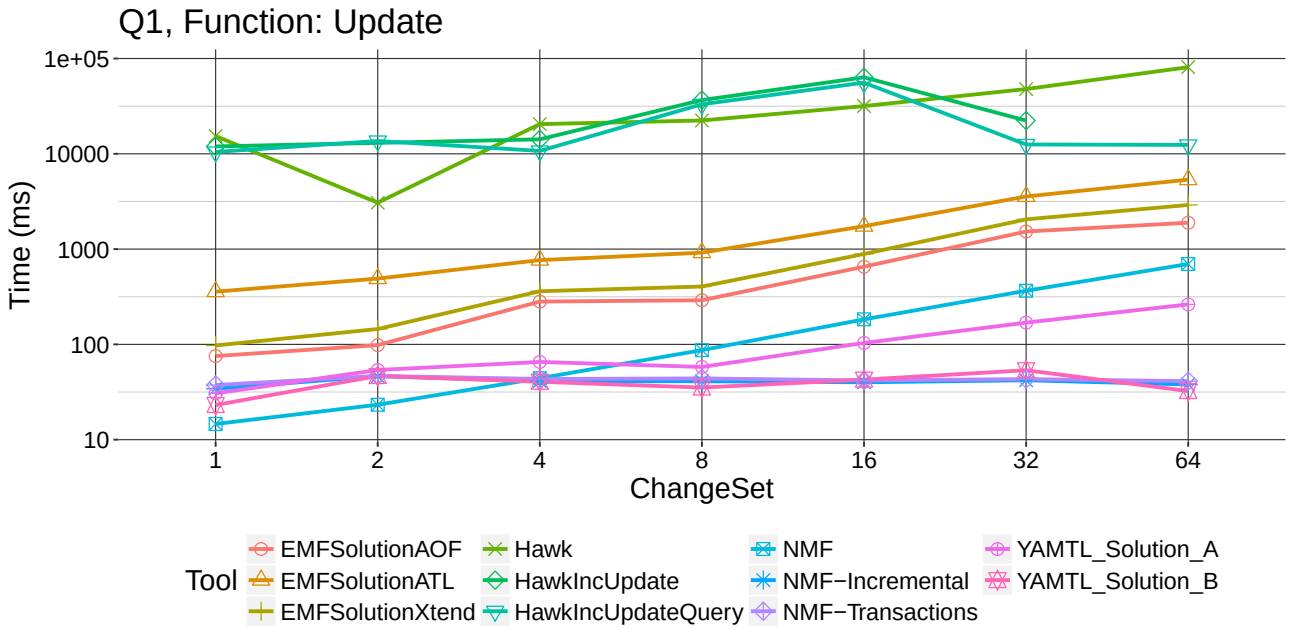

Figure 1: Incremental Evaluation of Query 1

For both queries, one can see that the difference whether or not changes are propagated in transactions is negligible, the curves for the incremental NMF solutions with or without transactions mostly overlap. Second, the slope of these two incremental NMF solutions is much flatter than the NMF reference solution. This indicates that unlike the reference solution that always recalculates the entire analysis results from scratch, the DDGs

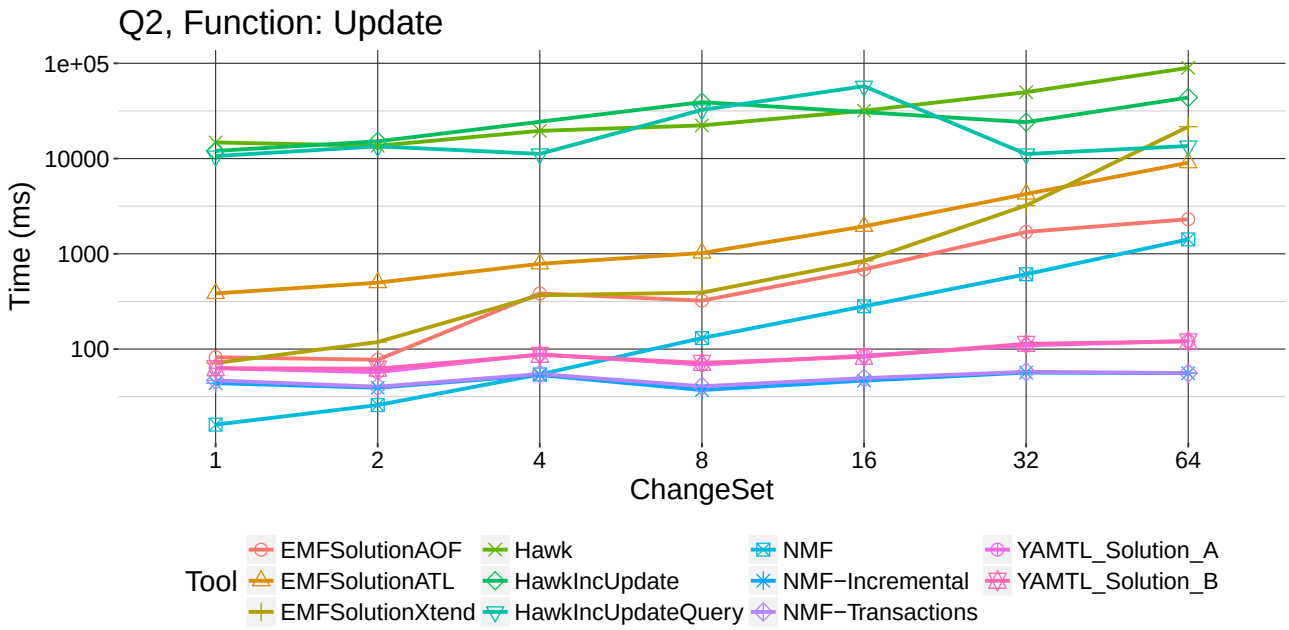

Figure 2: Incremental evaluation of Query 2

in the incremental solutions really enable an incremental change propagation where the time for the change propagation does not depend on the overall model size. For the largest model size depicted, this yields a speedup of more than an order of magnitude compared to the NMF reference solution. For query 2, the incremental NMF solutions are the fastest. For query 1, only the YAMTL solution is slightly faster.

5 Conclusions

Our solution shows the power of implicit incrementalization with NMF but also its pitfalls: We needed to slightly modify the query implementations as we are limited to incrementalizable query operators, but using them, the incremental solutions are very concise and understandable. For Query 2, we needed to implement a custom query operator, but this could be reused for other applications. The performance results show that our solution is one of the fastest ones, beating the reference solution by multiple orders of magnitude.

References

[1] G. Hinkel, "The TTC 2018 Social Media Case," in *Proceedings of the 11th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018) federation of conferences*, ser. CEUR Workshop Proceedings, CEUR-WS.org, 2018.

[2] G. Hinkel, "Implicit Incremental Model Analyses and Transformations," PhD thesis, Karlsruhe Institute of Technology, 2017.

[3] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[4] G. Hinkel and L. Happe, "An NMF Solution to the TTC Train Benchmark Case," in *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences*, ser. CEUR Workshop Proceedings, vol. 1524, CEUR-WS.org, 2015, pp. 142–146.

[5] G. Hinkel, "NMF: A Modeling Framework for the .NET Platform," Karlsruhe Institute of Technology, Tech. Rep., 2016.

[6] G. Hinkel, "NMF: A multi-platform Modeling Framework," in *Theory and Practice of Model Transformations: 11th International Conference, ICMT 2018, Held as Part of STAF 2018, Toulouse, France, June 25-29, 2018. Proceedings*, accepted, to appear, Springer International Publishing, 2018.

[7] G. Hinkel, *Implicit Incremental Model Analyses and Transformations*, ser. The Karlsruhe Series on Software Design and Quality. Karlsruhe Scientific Publishing, 2018, vol. 26, to appear.