A cross-technology benchmark for incremental graph queries

Georg Hinkel, Antonio Garcia-Dominguez, René Schöne, Artur Boronat, Massimo Tisi, Théo Le Calvar, Frederic Jouault, József Marton, Tamás Nyíri, János Benjamin Antal, et al.

Software and Systems Modeling (2022) 21:755–804
https://doi.org/10.1007/s10270-021-00927-5

REGULAR PAPER

A cross-technology benchmark for incremental graph queries

Georg Hinkel[1] · Antonio Garcia-Dominguez[2] · René Schöne[3] · Artur Boronat[4] · Massimo Tisi[5] · Théo Le Calvar[5,6] · Frederic Jouault[7] · József Marton[8] · Tamás Nyíri[8] · János Benjamin Antal[9] · Márton Elekes[9] · Gábor Szárnyas[10]

Received: 12 December 2020 / Revised: 31 August 2021 / Accepted: 14 September 2021 / Published online: 8 December 2021

Abstract

To cope with the increased complexity of systems, models are used to capture what is considered the essence of a system. Such models are typically represented as a graph, which is queried to gain insight into the modelled system. Often, the results of these queries need to be adjusted according to updated requirements and are therefore a subject of maintenance activities. It is thus necessary to support writing model queries with adequate languages. However, in order to stay meaningful, the analysis results need to be refreshed as soon as the underlying models change. Therefore, a good execution speed is mandatory in order to cope with frequent model changes. In this paper, we propose a benchmark to assess model query technologies in the presence of model change sequences in the domain of social media. We present solutions to this benchmark in a variety of 11 different tools and compare them with respect to explicitness of incrementalization, asymptotic complexity and performance.

Keywords Graph queries · Graph analytics · Model-driven engineering · Performance benchmark · Graph databases · relational databases · Incremental queries · Incremental computing

1 Introduction

Models are a highly valuable asset in any engineering process as they capture the knowledge of a system in a formal abstraction. This abstraction allows to reason on properties in order to obtain insights on the underlying physical system through analysis.

These insights need to be refreshed as soon as the models of the system change in order to stay meaningful. However, for large systems it is often not viable to recalculate the entire model analysis for every change. Rather, it is desirable to

Communicated by Alexander Egyed.

✉ Márton Elekes
elekes@mit.bme.hu

Georg Hinkel
georg.hinkel@gmail.com

Antonio Garcia-Dominguez
a.garcia-dominguez@aston.ac.uk

René Schöne
rene.schoene@tu-dresden.de

Artur Boronat
artur.boronat@leicester.ac.uk

Massimo Tisi
massimo.tisi@imt-atlantique.fr

Théo Le Calvar
theo.le-calvar@imt-atlantique.fr

Frederic Jouault
frederic.jouault@eseo.fr

József Marton
marton.jozsef@vik.bme.hu

Gábor Szárnyas
gabor.szarnyas@cwi.nl

[1] Wiesbaden, Germany

[2] Aston University, Birmingham, UK

[3] Technische Universität Dresden, Software Technology Group, Dresden, Germany

[4] School of Informatics, University of Leicester, Leicester, UK

[5] IMT Atlantique, LS2N (UMR CNRS 6004), Nantes, France

[6] DIRO, Université de Montréal, Montreal, Canada

[7] ERIS, ESEO-TECH, Angers, France

[8] Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics, Budapest, Hungary

[9] Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary

[10] CWI, Amsterdam, The Netherlands

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.

Requirements regarding these insights often change over time. This makes it very important to express such analyses in a maintainable and understandable form. Recently, multiple model query technologies [8,9,18,46,65] have been proposed to aid this problem by deriving an incremental change propagation from a declarative specification. Further, of course one could also use existing and established non-incremental query technology, particularly if some parts of the analysis are more complex and may not be supported by tools that target incremental change propagation. Finally, it is also an option to translate the model into dedicated analysis methods to reuse query technology not based on models. Given this plethora of options, it is difficult to estimate the differences and find the trade-offs between these approaches in terms of understandability, conciseness, efficiency and others. Does the tool fit into my technology space? Is it useful to rely on the incrementalization of a tool or is it better—at least performance-wise—to implement change propagation explicitly? How long does it take to recover from an application crash? How much development effort will be necessary to implement change propagation? How does it scale? Can I speed it up by adding more CPU cores? Is the tool extensible or can it happen that my analysis is not supported at some point?[1]

To aid this comparison and assess how current modelling technologies are capable of offering a concise and understandable language for model analysis, yet still offer a good performance in the presence of frequent model changes, we propose the "Social Media" benchmark. In this benchmark, two queries should be formulated that analyse a model of a social media network. In social networks, new Posts, Comments and likes arrive at a very high frequency and thus cause analyses of the entire network to invalidate quickly. In our benchmark, the analyses shall find the most influential posts and comments, according to selected criteria. While the first query is rather simple in the sense that it can easily be solved with standard query operators, the second query is more complex as graph algorithms are used that are not directly supported by query technologies.

Graph queries are difficult from an incrementalization point of view [31] as they capture a rich family of algorithms and they are often 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 how with the computation was performed in the batch (or initial) scenario. However, there is no comparison yet how and how efficient such dynamic algorithms can be included into incremental query technologies.

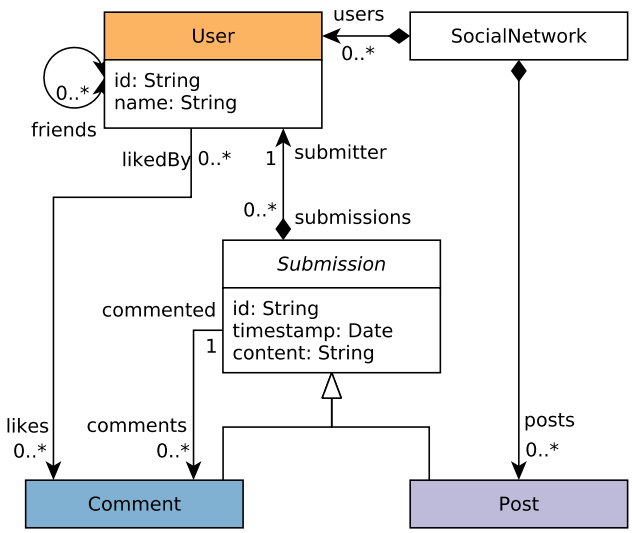


Fig. 1 Metamodel (graph schema) of the social network model

We collected 11 implementations of this benchmark that cover most of the possible approaches mentioned above and have mostly been created by authors or active developers of the respective tools. Therefore, we claim that we can compare the tools through the solutions for this benchmark. The tools cover a wide area of model query technology and database management systems; thus, the analysis of the solutions using those tools allows us to reason on the previously mentioned questions.

The benchmark was originally proposed at the Transformation Tool Contest (TTC) 2018 as live contest [51]. At the TTC, several solutions have been submitted [12,17,34,49,78]. After the TTC, we invited further researchers working in incremental query technology to provide solutions in their favourite tools. In this paper, we present all of these solutions and compare them with respect to their features, complexity of change propagation and overall performance.

The remainder of this paper is structured as follows: Sect. 2 presents the benchmark and the queries that it consists of. Section 3 explains the solutions. Section 4 compares the solutions with respect to the declarativeness of the query language, the used data model, the explicitness of the incrementalization, the durability, support for parallelism and the asymptotic complexity of change propagations. Section 5 presents experimental performance results and their analysis. Section 6 discusses related work, and Sect. 7 concludes the paper.

[1] These questions have been selected by the authors. We do not claim that this list of questions is complete and a proper analysis of the needs of developers is out of scope for this paper.

2 The social media benchmark

In this section, we present the benchmark which we use to compare existing incrementalization approaches. First, Sect. 2.1 explains the metamodel that we use to model a social network graph, and then, Sect. 2.2 presents the two queries used in the benchmark. We describe the change sequences in Sect. 2.3 and the phases of the benchmark in Sect. 2.4.

2.1 Metamodel and change sequences

In this benchmark, we use the data from the 2016 DEBS Grand Challenge,[2] adapted from the LDBC Social Network Benchmark [5,30] and the SIGMOD 2014 Programming Contest [27]. For this version of the benchmark, we created an Ecore metamodel to describe the social network, translated the series of events in the original data sources to models and change sequences and manually tuned the obtained change sequences.[3]

The metamodel of the social network is depicted in Fig. 1. The social network consists of Users, Posts and Comments. Users may have friends, which is a unidirectional edge (i.e. it behaves as a symmetric relationship). Submissions form a tree with a Post in its root and Comments as the rest of its nodes. Users may like Comments (likes edge).

2.2 Queries

In the scope of the proposed benchmark, we focus on two model queries. The first query (Q1) is rather easy and is expected to be directly supported by the tools. It shall return the most controversial Posts in the social media network. The second query (Q2) is more sophisticated, and we expected it not to be directly supported by tools and force the solutions into case-specific extensions. It shall return the most influential Comments. The results of both queries are concatenated to strings in order to use them for automated correctness checks.

2.2.1 Query 1: Most controversial posts

We consider a Post as controversial, if it starts a debate through its Comments. For this, we assign a score of 10 for each Comment that belongs to a Post. Hereby, we consider a Comment belonging to a Post, if it is a reply to either (1) the Post itself or (2) another Comment that already belongs to the Post. In addition, we also value if Users liked Comments, so we additionally assign a score of 1 for each User that has liked a Comment (Users are counted per Comment liked, i.e. a User can contribute multiple times to the overall score). In short, the score of a Post is calculated by taking all Comments which are in the Submission tree rooted in the Post, and scoring each of them as the 10 plus the number of Users that liked the Comment, then summing up these scores.

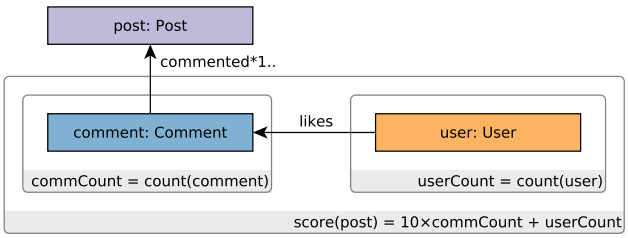


Fig. 2 Graph pattern for Q1

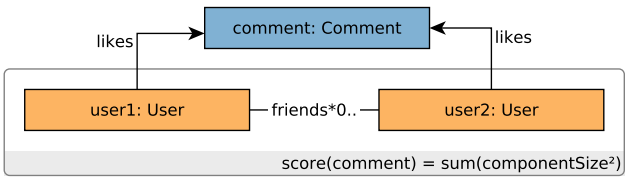


Fig. 3 Graph pattern for Q2

The goal of the query is to find the three Posts with the highest score. Ties are broken by timestamps, i.e. more recent Posts should take precedence over older Posts. The result string of this query is a concatenation of the Posts ids, separated by the character | (Fig. 2).

2.2.2 Query 2: Most influential comments

In this query, we aim to find Comments that are liked by groups of Users. We identify groups through the friendship relation. Hereby, Users that liked a specific Comment form an induced subgraph where two Users are connected if they are friends (but still, only Users who have liked the Comment are considered). The goal of the second query is to find connected components in that graph. We assign a score to each Comment which is the sum of its squared component sizes.

Similarly to the previous query, we aim to find the three Comments with the highest score. Ties are broken by timestamps, i.e. more recent Comments should take precedence over older Comments. The result string is again a concatenation of the Comment IDs, separated by the character | (Fig. 3).

2.3 Change sequences

To measure the incremental performance of solutions, the benchmark uses generated *change sequences*. The changes are available in the form of models. An excerpt of the change

[2] https://web.archive.org/web/20190131182518/https://debs.org/debs-2016-grand-challenge-social-networks/.

[3] We introduced changes that would actually affect the results of the queries defined in Sect. 2.2.

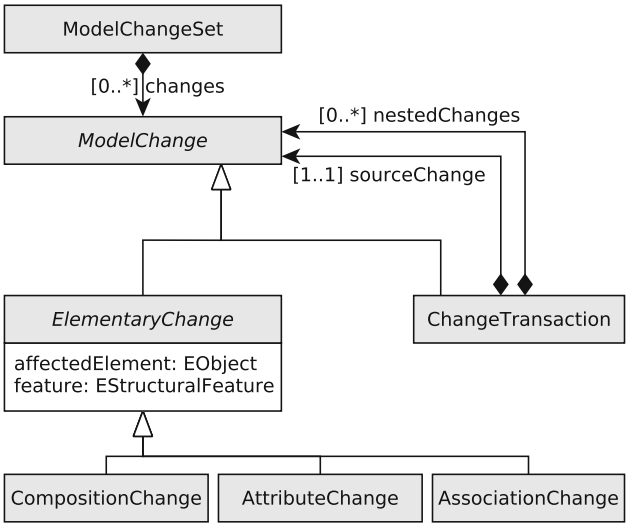


Fig. 4 Metamodel of model changes (simplified)

metamodel is depicted in Fig. 4. There are classes for each elementary change operation that distinguish between the type of a feature, whether it is an attribute, association or composition change. Subclasses further specify the kind of operation, i.e. whether elements are added, removed or changed. In these concrete classes, the added, deleted or assigned items are included.[4] The change metamodel also supports change transactions where a source change implies some other changes, for example, setting opposite references.

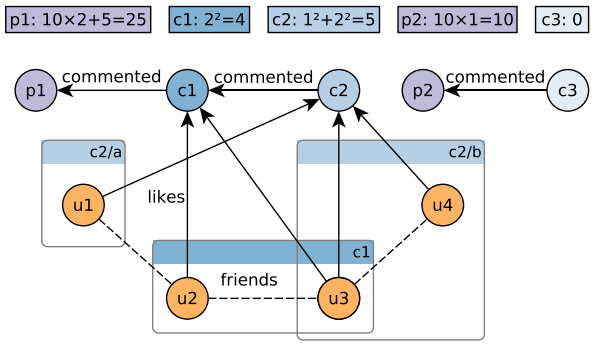
The elementary changes can be categorized into the following five change types:

- A new User is added
- A new Post is added
- A new Comment is added as a reply to an existing Submission
- A Comment is liked
- Two existing Users become friends

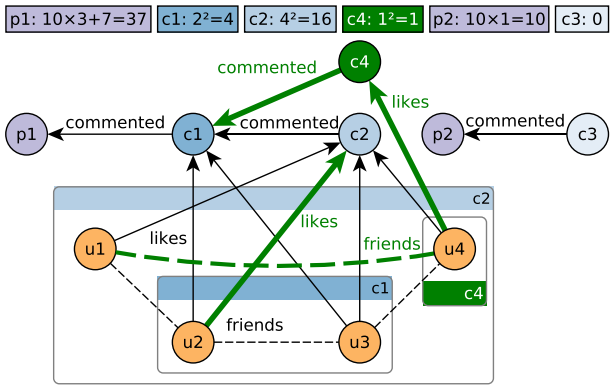
The changes are always additive, i.e. nodes or edges are never deleted. During the contest, these change sequences were made available in NMF and EMF. However, solutions are also allowed to transform the change models into their own, internal change representation. In such case, the transformation of the change representation is excluded from the time measurements.

An example of an update consisting of five elementary changes is depicted in Fig. 5.

[4] In a composite insertion, the added element is contained, otherwise only referenced.



(a) Initial model and scores. Comment `c2` has two components: `c2/a` consists of User `u1`, while `c2/b` consists of Users `u3` and `u4`. Its total score is the sum of the component sizes, i.e. $1^2 + 2^2 = 5$. The result `p1|p2` for Q1 and `c2|c1|c3` for Q2.



(b) Updated model after performing a change sequence that inserted five entities: (1) a friends edge between Users `u1` and `u4`, (2) a likes edge from User `u2` to Comment `c2`, (3) a Comment node `c4` with (4) an outgoing commented edge to Comment `c1` and (5) an incoming likes edge from User `u4`. The changes have increased the score of Post `p1` by 10 + 2 and also resulted in Comment `c2` having a single component of size 4, therefore being awarded a score of $4^2 = 16$. Comment `c4` received a score of $1^2 = 1$. The result `p1|p2` for Q1 and `c2|c1|c4` for Q2.

Fig. 5 Example model: initial and updated state with expected query results. The undirected friends edges are denoted with dashed lines

2.4 Benchmark phases

The benchmark consists of the following execution phases:

1. **Initialization:** Setup modelling framework.
2. **Loading:** Load initial models.
3. **Initial evaluation:** Compute selected query for initial model.
4. **Updates:** *Change sequences* are applied to the model. Each change sequence consists of several elementary change operations. After each change sequence, the query result must be consistent with the changed source models, either by entirely creating the view from scratch (*batch*

evaluation) or by propagating changes to the view result (*incremental evaluation*).

Solutions are mainly competing on their *update* performance, but are also expected to keep the times of the *loading* and *initial evaluation* at a reasonable level.

3 Solutions

In this section, we present the solutions developed for the Social Media benchmark from Sect. 2. We present the following solutions:

- Modelling tools and graph databases running in a managed runtime (JVM[5] or CLR[6]):
  - A solution using the plain .NET query API and NMF to represent the models in memory (Sect. 3.2) and a solution using the incrementalization system of NMF (Sect. 3.3).
  - Multiple solutions using the model management system Hawk (Sect. 3.4).
  - A solution using the incremental reference attribute grammar tool JastAdd (Sect. 3.5).
  - A solution using the model transformation language YAMTL (Sect. 3.6), with a batch variant and two incremental variants (implicit/explicit incrementality).
  - A batch solution using the ATL model transformation language with the EMFTVM batch execution engine (Sect. 3.7).
  - A solution written in Xtend (Sect. 3.8).
  - Two incremental solutions based on AOF (Sect. 3.9): (1) the AOF solution with Xtend surface language, (2) the ATL Incremental solution with ATL surface language.
  - Batch and incremental solutions using the Neo4j graph DBMS (Sects. 3.10–3.11).
- Tools using relational or matrix-based representations, running on native runtimes:
  - Batch and incremental solutions using the PostgreSQL relational DBMS (Sects. 3.12–3.13).
  - Batch and incremental solutions formulated in linear algebra using the GraphBLAS API and the SuiteSparse:GraphBLAS library (Sects. 3.14–3.15), implemented in C++.
  - A solution using the Differential Dataflow programming model implemented in Rust (Sect. 3.16).

[5] Java Virtual Machine, the Java runtime.

[6] Common Language Runtime, the .NET runtime.

In broad terms, there have been recurrent approaches for solving the two queries. To help reduce the length of the solution descriptions, a first subsection will provide a broad categorization of the approaches followed by the various solutions for each query. This will be followed by a subsection for each solution: it will start with a description of the used tool, followed by detailed explanations on how they implemented the queries. Given that most solutions have been developed by the tool authors or expert developers, we assume that the solutions represent the best or close to optimal solution possible for each tool.

Solutions created during the contest were described in the proceedings of the event [35]. Preliminary version of the Neo4j and GraphBLAS solutions was discussed in [29] and [28], respectively.

3.1 Common solution approaches

Further examination of the various solutions showed that several common patterns were recurrent through the approaches chosen to solve the two queries. In this section, we will provide a broad classification that will be used in the solution descriptions, in order to allow them to focus on their technology-specific aspects.

3.1.1 Query 1: Most controversial posts

The solutions for this query can largely be characterized across two dimensions: whether they were incremental in their maintenance of the scores, and how they maintained the set of the top three elements. Table 1 summarizes this classification.

Regarding the first dimension, we acknowledge similarly to Giese and Wagner [36] that there can be different degrees of incrementality in a transformation.[7] Specifically, we consider these three variants:

- Non-incremental or *batch* solutions, which re-compute the scores from scratch after each change.
- *Partially incremental* solutions, which re-compute scores of Posts impacted by changes from scratch.
- *Fully incremental* solutions that update the scores of the Posts impacted by changes as needed, without recomputing them from scratch (for instance, a new like would simply increase the previous score by 1).

In regard to the second dimension, four broad variants were found:

[7] While there are some similarities between our "partially/fully" incremental labels and the "effectively/fully incremental" labels of Giese and Wagner, we do not claim our definitions to be equivalent, as they are specific to these queries.

Table 1 Classification of approaches for Q1, ordered by solution name and support for incrementality

Solution	Incrementality	Sorting
AOF	⊗	Incremental
ATL	○	Full
ATL incremental	⊗	Incremental
Differential Dataflow	⊗	Full
GraphBLAS	○	Offline top-x
GraphBLAS incremental	⊗	Offline top-x
Hawk	⊘	Full
Hawk (IU)	⊘	Full
Hawk (IUQ)	⊘	Online top-x
JastAdd	⊘	Offline top-x
Neo4j	○	Offline top-x
Neo4j incremental	⊗	Incremental
NMF reference	○	Full
NMF incremental	⊗	Incremental
PostgreSQL	○	Online top-x
PostgreSQL incremental	⊗	Online top-x
Xtend	○	Online top-x
YAMTL-B	○	Online top-x
YAMTL-II	⊘	Online top-x
YAMTL-EI	⊗	Online top-x

Notation: ⊗ yes; ⊘ to some extent; ○ no

- *Full sorting* solutions, which use a standard sorting algorithm (typically the one in the standard libraries for the chosen language) to sort the elements by score, incurring a cost of $O(n \log n)$.
- *Incremental sorting* solutions, which use a dedicated data structure to maintain the elements sorted as they appear and as their individual scores change (e.g. a balanced search tree).
- *Offline top-x* solutions, which keep in memory the full list of scores and do a single pass over the elements, maintaining only the top-x elements in an auxiliary list. This has a reduced cost of $O(n)$.
- *Online top-x* solutions, which only keep in memory the Posts with the top-x scores at any time. This effectively has the same sorting cost of $O(n)$, but reduces the amount of memory needed.

Note that the *online top-x* is only feasible due to the fact that the possible changes mentioned in Sect. 2.3 never remove likes or delete comments: this means that scores are monotonically increasing. This optimization will typically be one that a generic "top-x" incremental operator would not do, as the sorting key could go down as well as up in value in the general case.

3.1.2 Query 2: Most influential comments

The solutions can be characterized across three dimensions: whether they maintained the set of connected components in the graph incrementally, the approach used to compute the set of connected components, and how they sorted the scored results. The general classification is summarized in Table 2.

For the first dimension of incrementality, this is generally easier than in the previous solution, as it is simply whether after each change, they update the set of connected components, or re-compute it. For the second dimension, five broad approaches were followed:

- The most popular approach was to reuse *Tarjan's algorithm* for computing the set of connected components in a graph [84], which has a worst-case performance of $O(|V| + |E|)$.
- The next most popular approach was a *naive recomputation* of the component that each node belonged to after each change, by using breadth-first or depth-first traversals from that node.
- Some solutions took advantage of the fact that the changes never remove links, using a simpler *union-find* data structure and using the union operation whenever a new edge was added. In a way, this is similar to how some solutions of query 1 used this "monotonically growing" graph to simplify the way to maintain the top-x elements.
- There were a few individual solutions that used *specialized algorithms*, such as FastSV [88], or label propagation.

The third dimension is the approach followed to compute the top scoring results, which uses the same four broad variants as in Sect. 3.1.1.

3.2 Reference solution: C# query syntax

As reference, we use a solution using NMF [45,50] for the model representation and the standard .NET collection operators through the C# query syntax. During the TTC and throughout the remainder of this paper, this solution is used as a reference in terms of performance and understandability as it represents a solution if one just uses what mainstream programming languages (in this case C#) can provide without a dedicated incremental model query technology.

3.2.1 Tool description

The query syntax and the underlying collection operators are features built into the C# programming language and actively used by millions of developers.

Table 2 Classification of approaches for Q2, sorted by tool and incrementality

Solution	Incremental conn. components	Algorithm	Sorting
AOF	⊗	Breadth-first traversal	Incremental
ATL	○	Depth-first traversal	Full
ATL incremental	⊗	Breadth-first traversal	Incremental
Differential Dataflow	⊗	Fixed-point label propagation	Full
GraphBLAS	○	FastSV	Offline top-x
GraphBLAS incremental	⊘	FastSV on overestimation + merge	Online top-x
Hawk	○	Tarjan	Full
Hawk (IU)	○	Tarjan	Full
Hawk (IUQ)	○	Tarjan	Online top-x
JastAdd	⊘	Depth-first/Kosaraju	Offline top-x
Neo4j	○	Union-find variant	Offline top-x
Neo4j incremental	⊗	Breadth-first traversal	Incremental
NMF reference	○	Tarjan	Full
NMF incremental	⊗	Edge changes	Incremental
PostgreSQL	○	Breadth-first traversal	Online top-x
PostgreSQL incremental	⊘	Overestimation + breadth-first traversal	Online top-x
Xtend	○	Tarjan	Online top-x
YAMTL-B	○	Weighted quick-union-find with path compression	Online top-x
YAMTL-II	⊗	Weighted quick-union-find with path compression	Online top-x
YAMTL-EI	⊗	Weighted quick-union-find with path compression	Online top-x

Notation—⊗ yes; ⊘ to some extent; ○ no. Overestimation means that all connected components are re-computed that might be affected by the changes

3.2.2 Query 1

```
1 string.Join('|',
2   (from post in SocialNetwork.Posts
3    let score = post.Descendants().OfType<Comment>()
4                    .Sum(c => 10 + c.LikedBy.Count)
5    orderby (score, post.Timestamp) descending
6    select post.Id).Take(3));
```

Listing 1 A solution for Q1 using the standard collection operators of .NET.

The reference solution for Q1 is depicted in Listing 1. It is a full sorting batch implementation. Lines 3–4 of this listing compute the score of a given Post by summing up the scores for the Comments of a given Post. To get all Comments of a Post, the solution simply uses the `Descendants` operation available to any NMF model element and combines it with a simple type filter. This implicitly utilizes the composition hierarchy to obtain a collection of all elements contained somewhere in the given Post. Line 5 simply states that the collection of Posts should be ordered by the score and the timestamp, before Line 6 specifies that we are only interested in the first three entries.

3.2.3 Query 2

```
1 string.Join('|',
2   (from comment in SocialNetwork.Descendants().OfType<Comment>()
3    let layering = Layering<IUser>.CreateLayers(
4        comment.LikedBy,
5        u => u.Friends.Intersect(comment.LikedBy))
6    let score = layering.Sum(l => Square(l.Count))
7    orderby (score, comment.Timestamp) descending
8    select comment.Id).Take(3));
```

Listing 2 A solution for Q2 using the standard collection operators of .NET and an implementation of Tarjan's algorithm.

The reference solution for Q2 is depicted in Listing 2, a full sorting batch implementation using Tarjan's algorithm. It is very similar to Listing 1 in its overall structure except for the slightly more complex score calculation due to the required computation of connected components. For this, we are using the class `Layering`, an implementation of Tarjan's algorithm [84]. This implementation requires specifying the underlying graph through its nodes (for each Comment, this is the set of Users that have liked the Comment) and a function obtaining incident nodes of a given node (in this case the friends of a given User that have also liked the Comment). The result of the scoring computation is then again ordered and the IDs of the first three elements are returned.

3.3 NMF incremental

3.3.1 Tool description

NMF Expressions [46,52] is an incrementalization system that uses the feature of the C# language to compile lambda expressions into models of the code, instead of machine code. These models are then used to derive a dynamic dependency graph (DDG) from a given expression and observe changes. These changes originate from elementary update notifications and are propagated through the dependency graph.

Based on the underlying formalization as a categorial functor, *NMF Expressions* is able to include custom incrementalizations of functions and includes a library of such manually incrementalized operators, including most of the Standard Query Operators (SQO).[8]

3.3.2 Query 1

While *NMF Expressions* includes an incrementalization of most SQOs, the `Take` method used in the reference solution for Q1 is not supported, same as any other query operators of the SQO that deal with indices. The reason for this is that it is costly to find out the index of an element in, for example, a filtered, unsorted 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 top-x queries, i.e. 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 with the size of the search tree). The computation of the scores is, however, using the incrementalization system and fully incremental.

[8] http://msdn.microsoft.com/en-us/library/bb394939.aspx. Even though the term SQO may not be common to many developers, the syntax is very commonly used in the .NET community, in particular through the LINQ technology that translates queries to SQL or other declarative languages.

```
1 query = Observable.Expression(() =>
2    SocialNetwork.Posts.TopX(3,
3       post => ValueTuple.Create(
4         post.Descendants().OfType<Comment>()
5             .Sum(c => 10 + c.LikedBy.Count),
6         post.Timestamp))
7 );
```

Listing 3 Incremental NMF Solution for Q1.

The solution to Q1 is shown in Listing 3. We simply calculate the top-3 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 3 to 6 determine how this tuple is created: we iterate over all Comments in a Post and, for each, sum up 10 plus the number of Users that have liked the Comment.

Lines 1 and 7 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.

3.3.3 Query 2

The solution for Q2 is very similar to the solution of Q1, except for the fact that for each Comment, it involves finding connected components in a graph spanned by Users that liked the Comment and their friendship relations. There is no incremental implementation for finding connected components available that is built into *NMF*.

From an algorithmic point of view, the incrementalization of the connected components will be rather simple: we cache the current set of connected components and re-compute whenever an edge is added to the graph between two nodes of different components. This incremental algorithm can be isolated into a class `ConnectedComponents` that is theoretically reusable in different contexts.

With this algorithmics class, we can solve Q2 as in Listing 4. Similar to Q1, the solution is an incremental solution with incrementally maintained full sort and an incremental version of Tarjan's algorithm. In Lines 1 and 2, we create a function that, given a Comment and a User, creates the collection of Users who (1) are friends with the given User and (2) like the Comment. Because we do not care about changes of the incident 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

```
1  Func<IComment, Func<IUser, IEnumerableExpression<IUser>>>
2  friendsBuilder = c => (u =>
             u.Friends.Intersect(c.LikedBy));
3
4  query = Observable.Expression(() =>
5    SocialNetwork.Descendants().OfType<IComment>()
6       .TopX(3, comment => ValueTuple.Create(
7          ConnectedComponents<IUser>.Create(
8                comment.LikedBy,
9                friendsBuilder(comment))
10             .Sum(group => Squared(group.Count())),
11         comment.Timestamp))
12 );
```

Listing 4 Incremental NMF Solution for Q2.

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 4 and 12, we frame the actual analysis with *NMF Expressions*, allowing it to create the DDG for the inner analysis that is in Lines 5 to 11. In particular, we first iterate all elements in the social network model and filter for Comments in Line 5. From these, we pick the topmost elements according to the tuple of scores and timestamps, similar to the solution of Q1. 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 (Lines 7 to 9). Given these connected components, we calculate the sum of the squared sizes (Line 10) and break ties using the timestamps (Line 11).

3.3.4 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 behaviour 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.

Q1 cannot profit from transactions at all, because whether the changes to any DDG node come at once or one after another does not matter. For Q2, this is slightly different because we are recalculating the connected components in multiple cases. If a change sequence contained multiple events that would cause to re-compute the connected components for a Comment, this calculation is only needed once if the changes are propagated in a transaction.

The transactional support in *NMF Expressions* is very easy for a developer to use: all that needs to be done is to put the changes inside a transaction. Because in the scope of the benchmark, these changes come in a dedicated change sequence object, we just need to wrap the application of such a change sequence in a transaction as done in Listing 5.

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

Listing 5 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. This is done by changing the execution engine implementation as depicted in Listing 6.

```
1 ExecutionEngine.Current = new ParallelExecutionEngine();
```

Listing 6 Enabling parallel change propagation in NMF.

The parallel change propagation then allows changes within a transaction to be propagated in parallel on different threads, synchronizing at each DDG node.

3.4 Hawk

3.4.1 Tool description

Eclipse Hawk is a tool to manage models that have been fragmented (e.g. for versioning purposes) by incrementally indexing the various connected fragments into a common graph database [8]. *Hawk* can watch over a local folder or a version control system and update the graph whenever the model files change. For this case study, *Hawk* was configured with the ability to index EMF models, maintain a graph database using one of three backends (Neo4j, SQLite, or Greycat [42]) and run queries in a dialect of the Epsilon Object Language (EOL) [61]. Greycat implements a graph-oriented data model on top of existing key-value stores, such as LevelDB or RocksDB. For the present work, LevelDB was chosen as the underlying key-value store.

Further, *Hawk* provides the concept of "derived attributes", which extend a type with pre-computed expressions which are updated incrementally as the graph changes. These attributes are also indexed for fast lookup.

3.4.2 Query 1

Several versions of the first query were implemented. For the sake of clarity, we will call these "batch", "incremental update" (IU) and "incremental update and query" (IUQ). As

mentioned in Table 1, all Hawk variants have partial incrementality. Batch and IU use full result sorting, and IUQ uses an "online top-x" approach.

The batch mode is the most direct use of *Hawk*. In this version, the tool is told to watch a folder that contains the initial version of the model. An EOL script replays the change sequences on the model, and *Hawk* updates the graph based on the new versions of the model. The Post type is extended with the `score` derived attribute, which is updated incrementally by Hawk. The definition of `score` is as follows, using `self` as the Post being extended:

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

Listing 7 Hawk Q1: Derived attribute `score` for Post.

With this derived attribute, it is simple to implement the main query itself. However, in order to sort the Posts with the Java Collections `sort` method, a native Java class implementing the `Comparator` interface is needed:

```
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));
```

Listing 8 Hawk Q1: Use of derived attribute `score` in the batch and incremental update solutions.

The incremental update mode uses an alternative graph update class, `ChangeSequenceAwareUpdater`. This speeds up the process by directly applying the change sequence on the graph, without touching the original model file. The query itself remains the same as the one in the batch mode, using the same derived attribute.

Finally, the incremental update and query mode reuses the same custom updater, while changing the way the query is run. A graph change listener is attached to *Hawk*: on each new model version, only the updated Posts are re-scored before selecting the new top-3 elements. The re-scoring is done by invoking the expression in Listing 7 on each Post that has been updated.

3.4.3 Query 2

The second query goes through the same three versions: *batch*, *incremental update* and *incremental update and query*. The actual query is noticeably more complex, since it essentially requires implementing Tarjan's strongly connected components algorithm [84] in about 37 lines of EOL. As indicated in Table 2, Tarjan is re-run after each change, and IUQ uses an "online top-x" sorting approach instead of doing a full sort.

In this case, the Comment class is extended with a `score` derived attribute. A high-level view of the query shows how it loops over the Users that liked the Comment, detecting connected components in each of them and computing the score of the Comment as the sum of the squares of the sizes of each component:

```
var components : Sequence;
var indexes : Map;
for (user in self.likedBy) {
  if (not indexes.containsKey(user.id)) {
    // strongconnect is 37 lines of EOL code
    user.strongconnect(self);
  }
}
return components.collect(c | c.size * c.size).sum();
```

Listing 9 Hawk Q2: Outline of the derived attribute `score` for Comment.

The queries for the *batch* and *incremental update mode* are the same, with a collection of all the IDs, scores and timestamps, and sorting to get the top-3 elements:

```
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));
```

Listing 10 Hawk Q2: Use of derived attribute `score` in the batch and incremental update solutions.

The *incremental update and query mode* works the same as in the previous query. A graph change listener detects the Comments that should be re-scored, and the new scores are merged with the old ones to keep the top-3 elements up to date.

3.5 JastAdd

3.5.1 Tool description

Attribute Grammars [60] can be used to describe the structure of context-free data along with their static semantics. Reference Attribute Grammars (RAGs) [43] extend this paradigm such that attributes are allowed to return other nodes of the abstract syntax tree (AST) as a result of their computation.

The tool *JastAdd* [44] is based on Reference Attribute Grammars and offers *non-terminal attributes* [86] and *circular attributes* [62], used to represent a part of the model and to compute parts of the second query, respectively. In *JastAdd*, a grammar specified in BNF with inheritance and attributes specified written in Java are woven together to generate plain

Java code. It contains a Java class for every non-terminal specified within the grammar, uses the code of the attributes inside its methods and has additional boilerplate code, e.g. to handle caching.

JastAdd does not work with EMF models directly, but requires users to transform the input metamodel into an AST representation using a dedicated syntax [67].

```
1 abstract ModelElement ::= <Id:Long> ;
2 SocialNetwork : ModelElement ::= User* Post* ;
3 User : ModelElement ::= <Name:String> ;
4 abstract Submission : ModelElement ::= <Timestamp:Long>
            <Content:String> Comment* ;
5 Comment : Submission ::= /<Post:Post>/ ;
6 Post : Submission ::= ;
7 rel User.friends* -> User ;
8 rel User.submissions* -> Submission ;
9 rel User.likes* <-> Comment.likedBy* ;
```

Listing 11 JastAdd Grammar for a `SocialNetwork`.

Listing 11 shows the grammar representing the metamodel of Fig. 1. All identifiable nodes inherit from `ModelElement` giving them a unique number. Also, the root node `SocialNetwork` is identifiable to later be able to insert new Users and Posts. The two non-containment references friends and submissions are replaced by explicit unidirectional relations, whereas the likes and likedBy edges are modelled using an explicit bidirectional relation (Line 9 in Listing 11).

3.5.2 Query 1

To solve the first query, all Comments referring to a Post are computed, and afterwards, the score of this Post can be calculated with the following attribute:

```
1 syn int Post.score() {
2   int result = 0;
3   for (Comment comment :
            commentsForPost()) {
4     result += 10 +
              comment.likedBy().size();
5   }
6   return result;
7 }
```

After gathering all scores, a simple iteration over all Posts and keeping the top-3 is performed to get the final result.

3.5.3 Query 2

For the second query, two variants are evaluated, showing two different approaches within the JastAdd solution. Those variants differ in the way how the Users liking the same Comment are calculated. For the first variant, a *circular attribute* `User.getCommentLikerFriends` (shown below) is used.

```
1 syn Set<User> User.getCommentLikerFriends(Comment comment) circular
            [new HashSet<User>()];
2 eq User.getCommentLikerFriends(Comment comment) {
3   Set<User> s = new HashSet<>();
4   s.add(this);
5   for (User f : friends()) {
6     for (Comment otherComment : f.likes()) {
7       if (otherComment == comment) {
8         s.add(f);
9         for (User commentLikerFriend :
10              f.getCommentLikerFriends(comment)) {
11          s.add(commentLikerFriend);
12        }
13      }
14    }
15  }
16  return s;
17 }
```

This attribute will start with the set containing the User it should compute the friends for. Then, if another friend likes the Comment, it adds this friend and calls itself recursively. The recursion always terminates as the number of Users is finite and the circular attribute is only invoked again if the returned set has changed.

The second variant follows the approach described in [66] highlighting reuse of application-independent analysis. In fact, the algorithm for computing an SCC presented in that work was reused without modification for solving the subproblem of query 2 to compute Users liking the same Comment. To integrate it, a mapping from User to `Component` was established, while the `friend` relationship served as a basis to connect those components.

To compute the score of a Comment using the second variant, the squared size of every set of components is summed:

```
1 refine Queries eq Comment.score() {
2   int score = 0;
3   Set<Set<Component>> sccs = this.toDependencyGraph().SCC();
4   for (Set<Component> userSet : sccs) {
5     int usize = userSet.size();
6     score += usize * usize;
7   }
8   return score;
9 }
```

To get the final result, the same iteration as for the first query is used (not shown here).

3.6 YAMTL

3.6.1 Tool description

YAMTL [16] is a model transformation language for EMF models, with support for incremental execution [18], designed as an internal DSL of Xtend. The solution to the Social Media benchmark uses query rules that only consist of input patterns, whose filters define the queries, and uses the YAMTL pattern matcher for evaluating them. Queries are defined using Xtend and the Java Collections Framework. The YAMTL solutions implement the online top-x approach, keeping the best three candidates at all times, with the operation `bestThreeCandidates`.

Three variants of the solutions have been implemented: *batch* (YAMTL-B), *implicitly incremental* (YAMTL-II) and *explicitly incremental* (YAMTL-EI). YAMTL-B disables dependency tracking enabling a faster initial transformation but subsequent updates compute queries from scratch. YAMTL-II detects which matches need to be re-computed according to updates to the input model, without requiring additional logic in the queries. For each impacted match, the filter expression is re-evaluated from scratch, and the solution is therefore partially incremental. YAMTL-EI exposes model updates affecting an impacted match so that these can be processed explicitly in the filter expression, and the solution is therefore fully incremental. The additional logic that handles incremental updates explicitly in YAMTL-EI has been highlighted in grey. YAMTL-B and YAMTL-II solutions can be obtained by deleting this code. Enabling incremental evaluation and explicit handling of updates is done via configuration parameters.

3.6.2 Query 1

Q1 is implemented as a query rule, whose input pattern is formed by an `in` element that will match Posts that contain Comments as indicated in the filter. In particular, the EMF method `post.eAllContents()` fetches all contained Comments within the matched Post. The implementation of the query is shown in Listing 12.

```
rule('Q1').in('post', SN.post).filter[
    val post = 'post'.fetch as Post
    var score = 0
    if (post.comments?.size > 0) {
        val map = this.fetch('dirtyObjects')
            as Map<EObject,List<YAMTLChangeType>>
        if (map === null || map.isEmpty) {
            val allComments = post.eAllContents
            while (allComments.hasNext) {
                score += 10 + (allComments.next
                        as Comment).likedBy.size
            }
        }
        else {
            for (comment: map.keySet) {
                score += 10 + (comment as Comment).likedBy.size
            }
        }
        threeBestCandidates.addIfIsThreeBest(post,
                score)
        return true
    }
    false
].query
```

Listing 12 Q1 in YAMTL.

The expression `this.fetch('dirtyObjects')` returns the objects that are added under the Post object being matched. For each such added object that is a Comment, the score is computed. The solution to Q2 shows how this case could be handled.

3.6.3 Query 2

The computation of connected components has been implemented using Sedgewick and Wayne's weighted quick union-find with path compression algorithm [80].[9] The query, including logic handling updates explicitly, is shown in Listing 13. The instantiation of the class `FriendComponentUtil_UF` computes the connected components of the graph whose nodes are the set of Users who liked the Comment, i.e.`comment.likedBy`. Comment scores are stored so that they can be subject to updates.

The expression `this.fetch('dirtyFeatures')` returns the collection of features for the Comment being matched, which have been updated so that they can be handled explicitly. The original union find algorithm has been extended to enable incremental updates of the computed components when a Comment is liked by a User (`addLikedBy()`) and when a new friendship is declared (`addFriendship()`). When there are less than three Comments with a score different from zero, Comments from `candidatesWithNilScore` are used to complete the list.

```
rule('Q2').in('comment', SN.comment).filter[
    val comment = 'comment'.fetch as Comment
    var score = 0
    var matches = false
    if (comment.likedBy.size > 0) {
        var FriendComponentUtil_UF fc
        fc = componentMap.get(comment)
        if (fc === null) {
            fc = new FriendComponentUtil_UF(comment.likedBy)
            componentMap.put(comment, fc)
        } else {
            val map = this.fetch('dirtyFeatures')
                as Map<EObject,List<YAMTLFeatureValueChange>>
            for (e: map.entrySet) {
                for (fv: e.value) {
                    switch(fv.featureName) {
                        case 'likedBy': fc.addLikedBy(fv.value as User)
                        case 'friends': fc.addFriendship(e.key as User,
                            fv.value as User)
                    }
                }
            }
        }
        score = fc.score
        threeBestCandidates.addIfIsThreeBest(comment, score)
        matches = true
    } else {
        if (threeBestCandidates.size <= 3)
            candidatesWithNilScore.add(comment)
    }
    matches
].query
```

Listing 13 Q2 in YAMTL.

[9] This solution is different from the one presented at the contest [17] and has been developed to facilitate the analysis together with the other tools involved in this study.

```
1 query topPosts = SN!Post.allInstances()->sortedBy(e | e.timestamp)->sortedBy(e |
        e.score)->reverse()->subSequence(1, 3);
2 helper context SN!Submission def : allComments : Sequence(SN!Comment) =
3     self.comments->union(self.comments->collect(e | e.allComments)->flatten());
4 helper context SN!Post def : countLikes : Integer = self.allComments->collect(e | e.likedBy.size())->sum();
5 helper context SN!Post def : score : Integer = 10*self.allComments->size() + self.countLikes;
```

Listing 14 Q1 in ATL.

```
1  query topComments = SN!Comment.allInstances()->sortedBy(e | e.timestamp)->sortedBy(e | e.score)->reverse()->subSequence(1,
          3);
2  helper context SN!Comment def : score : Integer = self.allComponents->collect(c | c.size()*c.size())->sum();
3  helper def : allFriends(u: SN!User, s:Sequence(SN!User)) : TupleType(component : Sequence(SN!User), remaining :
          Sequence(SN!User)) = ...
4
5  helper context SN!Comment def : allComponents : Sequence(Sequence(SN!User)) =
6    self.likedBy->iterate(u;
7      acc : TupleType(components : Sequence(Sequence(SN!User)), visited : Sequence(SN!User)) =
8        Tuple{components=Sequence{}, visited=Sequence{}} |
9      if (acc.visited->includes(u))
10       then acc
11       else let component : TupleType(component : Sequence(SN!User), remaining : Sequence(SN!User)) =
12         thisModule.allFriends(u, self.likedBy->excluding(acc.visited)).
13               component in
14           Tuple{components = acc.components.append(component), visited = acc.visited->union(component)}
15     endif).components;
```

Listing 15 Q2 in ATL.

3.7 ATL

3.7.1 Tool description

ATL [55] is one of the most common model transformation languages. The solution to the Social Media benchmark only uses ATL queries, i.e. the ATL constructs that allow users to define expressions in the ATL flavour of OCL. ATL queries can call helper OCL functions and libraries and are evaluated over the source model(s) by the ATL virtual machine, that is optimized for model operations.

The vanilla ATL that was used to implement the benchmark does not have support for incremental execution. Classical engines that execute ATL incrementally [56,63] do not support the incremental evaluation of OCL expressions. Changes on the source model trigger the recomputation only of the impacted OCL expressions, but the whole expressions are re-computed. In Sect. 3.9, we describe a solution that achieves incremental OCL expression evaluation for ATL code, by compiling it towards AOF.

The solution is a pure ATL query and executed on the most recent ATL virtual machine (EMFTVM). Since ATL queries are OCL expressions, this solution includes a complete encoding of the case study as declarative and functional OCL code.

3.7.2 Query 1

The full code for Q1 is presented in Listing 14. The recursive `allComments` helper gathers the set of Comment for a given Post, and a score for the Post is computed by the given formula (Line 11) considering the number of Comments and likes to these Comments. The main query `topPosts` sorts the set of Posts by score (and timestamp) and picks the top-3 Posts.

3.7.3 Query 2

The code for Q2 is shown in Listing 15. In particular, the `allComponents` helper implements a one-pass algorithm for the detection of all the connected components. The algorithm iterates on the liker Users: if the liker has not been visited, then compute a new component by the `allFriends` helper. The `allFriends` helper (whose implementation is not shown in the listing) is just a standard depth-first traversal, limited to the subgraph `s`. Finally, a score is computed for each Comment (Line 5), and the top-3 Comments are identified similarly to Q1 (Lines 1–2).

3.8 Xtend

3.8.1 Tool description

Xtend[10] [13] is a modern Java dialect suited for rapid prototyping thanks to its flexibility and expressiveness. Like ATL, the vanilla Xtend does not support incremental execution.

3.8.2 Query 1

We have written a first batch implementation of Q1 and Q2 in pure Xtend using the Eclipse Modeling Framework (EMF) plugin to perform loading and navigation into models. In a second step, we optimize this solution using Java 8 Streams to parallelize some operations on collections. The Xtend code used for the implementation of Q1 (Listing 16) shows that this mechanism is used two times: (1) to process all Posts in parallel and (2) to compute the sum of all likes received by Comments of a Post in the `computeScore` method. For better performance, we have also implemented a specific stream operation, called `Greatest3`, to avoid sorting the whole list of Posts while only the top-3 Posts can be considered.

```
def private queryQ1() {
  return socialNetwork.posts.parallelStream.collect(Collector.of([
    new Greatest3(Comparator.comparingInt[
        if(it === null) { Integer.MIN_VALUE } else { computeScore}
      ].thenComparing(Comparator.comparing[timestamp])
    )
  ], [$0.add($1)], [$0.merge($1)], [asList])).map[id].join("|")
}
def private computeScore(Post p) {
  val comments = p.eAllContents.filter(Comment).toList
  return comments.size*COMMENT_SCORE +
  comments.parallelStream.mapToInt[likedBy.size].sum*LIKE_SCORE
}
```

Listing 16 Q1 in Xtend with Java Streams.

3.8.3 Query 2

The code of Q2 (Listing 17) is similar to the implementation of Q1 except for the `computeScore` method. Indeed, the second query requires to find connected groups of Users through the friend relationship. For this purpose, the `computeScore` method uses a connected components algorithm based on Tarjan's algorithm [84].

3.9 AOF and ATL incremental

3.9.1 Tool description

Active operations [9] are OCL-like operations equipped with incremental propagation algorithms. They may thus be used to incrementally evaluate OCL expressions [19, Section 5] such as the one found in ATL-like model transformations. It is therefore possible to use active operations to write incremental queries and transformations.

```
def private queryQ2() {
  return socialNetwork.posts
    .map[eAllContents.filter(Comment).toList]
    .flatten.toList.parallelStream
    .collect(Collector.of([
      new Greatest3(Comparator.comparingInt[
          if(it === null) {Integer.MIN_VALUE} else {computeScore}
        ].thenComparing(Comparator.comparing[timestamp]))
    ], [$0.add($1)], [$0.merge($1)], [asList])).map[id].join("|")
}
def private computeScore(Comment c) {
  val layering = new Layering[User u |
    u.friends.filter[likes.contains(c)]]
  return layering.CreateLayers(c.likedBy).
    map[size*size].reduce[$0+$1] ?: 0
}
```

Listing 17 Q2 in Xtend with Java Streams.

The AOF implementation [53] of active operations supports EMF models and is based on the Observer design pattern, although alternative execution strategies [22] have been explored. It is implemented in Java and can be used from Java, Xtend, or ATL code. Each mutable value is wrapped in an observable box, which is either a collection, or a singleton value.

Though AOF provides enough basic active operations to implement the case study, creating specific operations sometimes helps [54] achieve a better performance. For this case study, we developed four new operations:

1. `sortedBy` returns a sorted copy of its source collection using one or more criteria using balanced binary trees.
2. `take` returns the *n* first elements of a collection.
3. `allContents` retrieves all model elements contained in a given source element, filtering them by type.
4. `layering` implements an incremental connected component algorithm.

From these, `layering` is more specific to some graph-related transformations, and the others are relatively generic.

We present two variants of this solution:

1. The *AOF* solution is written in Xtend, as shown in Listings 18 and 19.
2. The *ATL Incremental* solution is written in ATL and leverages the ATOL [23] compiler that translates it to Java code that makes use of AOF. It is basically a transliteration of the Xtend code from Listings 18 and 19 into ATL syntax, The main advantage of this variant w.r.t. the AOF variant is that it makes it possible to use the declarative ATL syntax.

3.9.2 Query 1

The implementation of Q1 in AOF is depicted in Listing 18. The actual computation is stored in a hash table such that it

[10] https://www.eclipse.org/xtend/.

does not have to be computed repeatedly. Within the score calculation, we use a method that iterates through the containment hierarchy in conjunction with a type filter, similar to the NMF solution. To calculate the score based on likes, we use the lifting mechanism of OCL that implicitly lifts the property `likedBy` to collections.

```
def private queryQ1() {
    return socialNetwork._posts
        .sortedBy([computeScore], [_timestamp.asOne(null)])
        .take(3).collect[id]
}
val scoreByPost = new HashMap<Post, IOne<Integer>>
def private computeScore(Post p) {
    return scoreByPost.get(p) ?: {
        val comments = p._allContents(Comment)
        val score = comments.size * COMMENT_SCORE +
                    comments.likedBy.size.sum * LIKE_SCORE
        val r = score.asOne(0)
        scoreByPost.put(p, r)
        r
    }
}
```

Listing 18 Q1 in Xtend using AOF.

The ATL Incremental implementation of Q1 is shown in [23, Listing 6]. It is very similar to the code from Listing 14, with the most notable difference being that the ATOL compiler does not support the `query` keyword.

3.9.3 Query 2

The solution for Q2 is depicted in Listing 19. Again, the actual score calculation is moved to a helper method. The score calculation itself is then making use of the `layering` operation.

```
def private queryQ2() {
    return socialNetwork._allContents(Comment)
        .sortedBy([computeScore], [_timestamp.asOne(null)])
        .take(3).collect[id]
}
val scoreByComment = new HashMap<Comment, IOne<Integer>>
def computeScore(Comment c) {
    return scoreByComment.get(c) ?: {
        val s = c._likedBy.layering[u |
            u._friends
             .selectMutable[f | f._likes.select[it == c].notEmpty]
        ].collectMutable[it?.size?.square ?: emptyOne].sum
        scoreByComment.put(c, s)
        s
    }
}
```

Listing 19 Q2 in Xtend using AOF.

3.10 Neo4j Batch

3.10.1 Tool description

Neo4j is a graph database management system using the *property graph* data model. Such graphs consist of labelled entities, i.e. nodes and edges, which can be described with *properties* encoded as key-value pairs. Neo4j uses the Cypher query language [33] which offers both read and update constructs [37]. While the main focus of Neo4j is to run graph queries in an online transaction processing (OLTP) setup, it also supports graph analytical algorithms with the Graph Data Science library[11] [71].

```
MATCH (p:Post)
OPTIONAL MATCH (p)<-[:COMMENTED*]-(c:Comment)
OPTIONAL MATCH (c)<-[:LIKES]-(u:User)
RETURN p.id AS id, 10*count(DISTINCT c) + count(u) AS score,
       p.timestamp AS timestamp
ORDER BY score DESC, timestamp DESC LIMIT 3
```

Listing 20 Neo4j batch implementation of Q1.

3.10.2 Query 1

Q1 c Cypher query in Listing 20. The Cypher language uses node labels (e.g. Post, Comment, User), edge types (e.g. COMMENTED, LIKES) to express graph patterns. The query matches every node with label Post, then all its Comments via a series of COMMENTED edges, then the Users via direct LIKES edges. The **`OPTIONAL MATCH`** clause denotes an optional pattern, where variables are set to NULL values if there is no match. The **`RETURN`** clause is used to group and aggregate. The results are grouped by the id and timestamp properties of the Posts, aggregated, and then, the top-3 scores are returned. The aggregation counts the likes using the number of Users (a User can like multiple Comments) and counts the number of Comments (**`DISTINCT`** is used to remove duplicate Comments).

```
1  MATCH (c:Comment) WHERE (c)<-[:LIKES]-(:User)
2  CALL gds.wcc.stream({
3    nodeQuery: "MATCH (c:Comment)<-[:LIKES]-(u:User)(**)
4                WHERE id(c)=" + id(c) + "
5                RETURN id(u) AS id",
6    relationshipQuery: "MATCH (u1:User)<-[:FRIEND]->(u2:User)
7                        RETURN id(u1) AS source, id(u2) AS target",(**)
8    validateRelationships: false})
9  YIELD componentId
10 WITH c, componentId, count(componentId) AS componentSize(**)
11 WITH c, componentSize * componentSize AS componentSize_2
12 RETURN c.id AS id, sum(componentSize_2) AS score, c.timestamp
13   ORDER BY score DESC, c.timestamp DESC LIMIT 3(**)
14 UNION ALL
15 MATCH (c:Comment)(**)
16 WHERE NOT (c)<-[:LIKES]-(:User)
17 RETURN c.id AS id, 0 AS score, c.timestamp
18   ORDER BY c.timestamp DESC LIMIT 3(**)
```

Listing 21 Neo4j batch implementation of Q2.

3.10.3 Query 2

Listing 21 shows the batch solution for Q2 using the variant of the union-find algorithm [68] implemented in the Neo4j Graph Data Science library. The procedure `gds.wcc.stream` is used to find connected components of the subgraph given by Cypher queries matching the nodes and the edges. For each Comment with likes, the first Cypher query in Lines 3-7 selects Users who like the Comment, and the second query selects all `FRIEND` edges as pairs of Users. The library loads

[11] https://neo4j.com/docs/graph-data-science/1.5/.

each subgraph into an in-memory projected subgraph before running the computations. The procedure returns the ID of the component containing the User node. Lines 10–13 calculate the squared sum of the component sizes and select the top-3 scores. On Lines 15–18, the query enumerates the top-3 Comments without likes and the **UNION** of the two sets are returned. For a detailed comparison of strategies to compute Q2, we refer the reader to [29].

3.11 Neo4j incremental

3.11.1 Tool description

The incremental solution for Neo4j uses node properties and new nodes to materialize the result of previous iterations. For every batch of updates these elements are refreshed, then the top-3 scores are collected. While Q1 can be computed efficiently with only Cypher constructs, the solution for Q2 uses the fixed-point calculation, dynamic node manipulation and reachability procedures of Neo4j's APOC stored procedure library.[12]

3.11.2 Query 1

To incrementally evaluate Q1, we initially compute the score for each Post as in the Neo4j Batch solution but, instead of returning it, we store it in the score property as shown in Listing 22. Based on this property, the current top-3 scores can be computed using Listing 25. The score property is indexed to improve lookup times. After new elements of an update are inserted, the score property of new Posts is initialized to zero, Listings 23-24 maintain the property for new Comment nodes and LIKES edges, and then, Listing 25 is used to get the top-3 elements.

```
1 MATCH (p:Post)
2 OPTIONAL MATCH (p)<-[:COMMENTED*]-(c:Comment)
3 OPTIONAL MATCH (c)<-[:LIKES]-(u:User)
4 WITH p, 10*count(DISTINCT c) + count(u) AS score
5 SET p.score = score
```

Listing 22 Neo4j incremental implementation of Q1 – initial evaluation.

```
1 WITH $commentVertex AS c
2 MATCH (p:Post)<-[:COMMENTED*]-(c:Comment)
3 SET p.score = p.score + 10
```

Listing 23 Neo4j incremental implementation of Q1 – maintenance after the insertion of new Comment nodes.

```
1 WITH $likesEdge AS l
2 MATCH (p:Post)<-[:COMMENTED*]-(:Comment)<-[l:LIKES]-(:User)
3 SET p.score = p.score + 1
```

Listing 24 Neo4j incremental implementation of Q1 – maintenance after the insertion of new LIKES edges.

[12] https://neo4j.com/labs/apoc/.

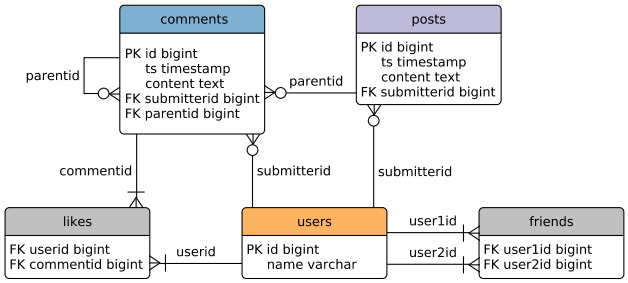


Fig. 6 ER diagram of the database schema

```
1 MATCH (p:Post)
2 WHERE p.score >= 0 // query hint to use index
3 RETURN p.id AS id, p.score AS score, p.timestamp AS timestamp
4 ORDER BY score DESC, timestamp DESC LIMIT 3
```

Listing 25 Neo4j incremental implementation of Q1 – get top-3 results.

3.11.3 Query 2

The incremental Neo4j solution for Q2 materializes the components of the subgraph for each Comment *comm* by converting the $\text{User} \xrightarrow{\text{LIKES}} \text{Comment}_{comm}$ edges to a Component node and inserting two edges:

$$\_\ \text{Comment}_{comm} \xrightarrow{\text{COMPONENT}} \text{Component},$$

$$\_\ \text{Component} \xrightarrow{\text{USER}} \text{User},$$

where the Component node connects all Users who know each other directly or via friends who also liked the Comment *comm*. The conversion is executed for each component one by one using the fixed-point query execution mechanism of APOC. To achieve this, first the solution marks the nodes of each subgraph with dynamically named labels (Listing 41) and finds reachable nodes using the APOC library (Listing 42). The incremental evaluation is performed by merging the components then maintaining their sizes and the resulting scores (Listings 43–44).

3.12 PostgreSQL Batch

3.12.1 Tool description

To study the usability and performance of relational database management systems (RDBMSs), we implemented a batch solution in PostgreSQL. Figure 6 shows the database schema capturing the social network model. Instances of each node type (e.g. Comment) and the edge type with many-to-many cardinality (friends) are stored in relations (tables) with the following schemas:

- *comments*(id, ts, content, submitterid, parentid)
- *posts*(id, ts, content, submitterid)
- *likes*(userid, commentid)
- *users*(id, name)
- *friends*(user1id, user2id)

Each relation representing a node has a primary key. Many-to-many edges are represented in *association tables* with two foreign keys. Many-to-one edges are stored as a foreign key in the table representing the node at the endpoint of the edge with a cardinality of "one". Additionally, indexes were defined on the foreign keys. This supports the SQL optimizer in choosing arbitrary join orders.

Evaluating both Q1 and Q2 require checking transitive reachability between nodes, a common recursive query which cannot be expressed in first-order logic or relational algebra [4]. However, it is possible to express such queries using a relational database by

1. either defining additional data structures and running a sequence of SQL queries in a loop until reaching a fixed point [26] or
2. or using SQL:1999's `WITH RECURSIVE` construct, which allows the formulation of recursive queries.

In this solution, we use `WITH RECURSIVE` as it is widely available in modern SQL implementations [89].

3.12.2 Query 1

To evaluate Q1, for each Comment, we need to first find the root Post of the Comment-chain, which is computed as the transitive closure of the Comment–parentid–Comment/Post edge type.

Having the transitive closure enables us to match the corresponding ⟨Post, Comment, User⟩ triples, where the Comment is a response rooted in the Post and the User is someone who liked the Comment. We use two left outer joins to ensure that Posts without Comments and Comments without likes are kept with NULLs. This is followed by an aggregation computing the score and finalized with a top-3 selection as shown in Listing 26.

3.12.3 Query 2

To evaluate Q2, we need to determine the connected components of the induced subgraphs on the User–friends–User edge type, which is computed using the transitive closure on the graph.

```
with recursive
  comments_with_ancestors(id, ancestorid) as (
     select c.id, c.parentid AS ancestorid
       from comments c
   union
     select cr.id, c.parentid AS ancestorid
       from comments_with_ancestors cr
          , comments c
      where cr.ancestorid = c.id
  )
select p.id, 10*count(distinct c.id) + count(l.userid) as score
  from posts p
       left join comments_with_ancestors c on (p.id = c.ancestorid)
       left join likes l on (c.id = l.commentid)
 group by p.id, p.ts
 order by score desc, p.ts desc limit 3
```

Listing 26 Batch PostgreSQL solution for Q1.

The outline of our Q2 implementation is shown in Listing 27. (The full query is given in Listing 30.) The implementation defines interim views using four subqueries and computes the result with the final (5th) query:

1. *comment_friends*(commentid, user1id, user2id): for a given Comment, pairs of Users who liked it and are friends.
2. *comment_friends_closed*(commentid, head_userid, tail_userid): the *transitive closure of cf relations*, i.e. for a given Comment, all pairs of Users who are reachable from each other through friends edges.
3. *connected_components*(commentid, userid, componentid): for a given Comment, the Users who belong to a given component of the friendship graph.
4. *comment_component_sizes*(commentid, componentid, component_size): for a given Comment, the component IDs and their size.
5. Finally, the resulting relation contains the top-3 Comments with the highest scores.

3.13 PostgreSQL incremental

To improve performance for repeated query executions, we have extended the PostgreSQL Batch solution (Sect. 3.12) with support for incremental updates. This section discusses the changes introduced in the schema and presents the queries that maintain the results upon changes.

3.13.1 Tool description

For the incremental solution, we have extended the database schema as shown in Fig. 7. Compared to the batch schema, this has three auxiliary relations and an extra attribute: (1) *q1_scoring* for maintaining the scores and timestamps of the Posts as defined in Q1, along with (2) *cf* and (3) *cfc* with the same semantics as in the Batch solution of Q2. The attribute (4) *comments.postid* is where Q1 maintains the root Post reference for each Comment, which is then used in Q1. Additionally, we have employed a *horizontal parti-*

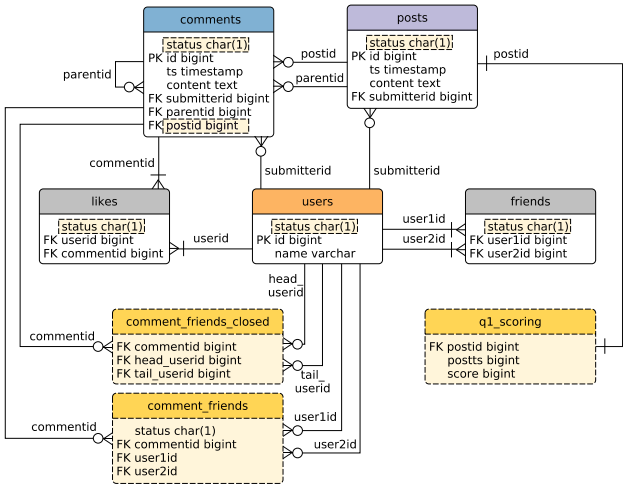


Fig. 7 ER diagram of the database schema, extended for incremental processing with 3 tables, 1 attribute to hold the root Post reference of the Comments and 5 status attributes added to the existing tables (highlighted in yellow boxes with dashed borders)

tioning strategy [1] on each table. The partitioning uses an attribute status consisting of a single character "B" or "D". During a particular update phase, rows that were already in the database are stored in the "B" (*before*) partition, while rows that have just been inserted are temporarily stored in the "D" (*diff*) partition. At the end of each update phase, rows from the *diff* partition migrate to the *before* partition. For both queries, we implemented algebraic incremental view maintenance with delta queries derived using the rules given in [38] and [81, Appendix E].

3.13.2 Query 1

The incremental solution for Q1 consists of three steps, each affecting the *q1_scoring* auxiliary relation, i.e. an initial step, then, for each update of the graph, a sequence of interim result maintenance and final result retrieval queries.

1. The initial step consists of two substeps:
   (a) We compute the root Post reference for all Comments (Listing 31) and materialize in attribute *comments.postid*.
   (b) We compute the score for all Posts (Listing 32)—similarly to the batch solution of Q1 except that we materialize each Post in relation *q1_scoring* with its timestamp and score.
2. The interim result maintenance is again split into two substeps:
   (a) In preparation, we compute the root Post reference for all new Comments (Listing 33) and materialize in attribute *comments.postid*.
   (b) Posts' score maintenance (Listing 34) is implemented using three subqueries, each computing the increment in score for certain types of changes:
   – The *diff_posts* subquery calculates the score of the new Posts. The Comments and likes of the new Posts must also be new, so the *diff* partition of all 3 relations are joined together. A left outer join is applied to include each new Post regardless of whether it has received any Comments and/or likes yet.
   – The *diff_comments* subquery calculates the extra score for old Posts gained by new Comments and the likes for them. This is expressed as the inner join of the *before* partition of Posts and *diff* partition of Comments. Then, a left outer join of the *diff* partition of likes is applied, as only new likes can refer to new Comments, and we need to calculate scores of all new Comments regardless of whether it received any likes.
   – The *diff_likes* subquery computes extra score for old Posts based for the new likes of their old Comments. This is expressed as the inner join of the *before* partition of Posts and Comments, and the *diff* partition of likes.
   In each subquery, one operand of the inner join is the *diff* partition of either the *posts*, *comments* or *likes* relations. Their usually small record count can be exploited by the SQL query optimizer to speed up joins. Using the calculations above, *q1_scoring* is updated by increasing the scores of old Posts and inserting new Posts along with their scores.
   The relational algebraic formula for this interim result maintenance query is given in Fig. 13 and proved in Fig. 14.
3. The retrieval step is a simple top-3 query on *q1_scoring* (Listing 35).

3.13.3 Query 2

Similarly to Q1, the incremental solution for Q2 again consists of three steps, i.e. an initial step, then, for each update of the graph, an alternating sequence of interim result maintenance and final result retrieval queries. Both the initial and the maintenance steps are further divided into two queries affecting the *cf* and *cfc* interim relations.

1. During the initial step, we first compute the *before* partition of the *cf* relation (Listing 36), then its closure in the *cfc* relation (Listing 38). The queries to perform this are analogous to the corresponding subqueries of the PostgreSQL Batch solution for Q2 (Listing 27).
2. The maintenance step consists of two substeps:

```
WITH RECURSIVE
  comment_friends(commentid, user1id, user2id) AS ( ... ),
                    --1
  comment_friends_closed(commentid, head_userid, tail_userid)
            AS --2
    SELECT l.commentid
         , l.userid AS head_userid, l.userid AS tail_userid
      FROM likes l
     UNION
    SELECT cfc.commentid, cfc.head_userid, f.user2id AS
              tail_userid
      FROM comment_friends_closed cfc, comment_friends f
     WHERE cfc.tail_userid = f.user1id
       AND cfc.commentid = f.commentid
), comment_components AS (
                                                  --3
    SELECT commentid, head_userid AS userid
         , min(tail_userid) AS componentid
      FROM comment_friends_closed
     GROUP BY commentid, head_userid
), comment_component_sizes AS (
                                              --4
    SELECT cc.commentid, cc.componentid, count(*) AS
              component_size
      FROM comment_components cc
     GROUP BY cc.commentid, cc.componentid
)
            --5 consider all comments including those without
                      likes
SELECT c.id AS commentid
     , coalesce( sum( power(ccs.component_size, 2) ), 0) AS
              score
  FROM comments c
  LEFT JOIN comment_component_sizes ccs ON (ccs.commentid =
            c.id)
 GROUP BY c.id, c.ts
 ORDER BY sum( power(ccs.component_size, 2) ) DESC NULLS LAST
        , c.ts DESC LIMIT 3
;
```

Listing 27 Batch PostgreSQL solution snippet for Q2.

(a) Updating the *cf* relation with a union of new *comment_friends* edges induced (Listing 37). Considering the *friends.user1id* as the left side on the join, and *friends.user2id* as its right side, a new edge is included by either:
   i. the new likes on the left side of all friendships with all likes on the right side,
   ii. the old likes on the left side of new friendships and all likes on the right side, and
   iii. old likes on the left side of old friendships and new likes on the right side.

(b) Updating the transitive closure is done in three successive subqueries (denoted as `comment_friends_closed_stage{0,1,2}` in Listing 39):
   i Each new like on a particular Comment is a new zero-length path in the closure.
   ii Instead of performing the computation on the *cf* graph, we use new edges (*comment_friends_diff*) to reach a fixed point in fewer steps.
   iii The optimization in the previous step omitted the edges in the existing transitive closure, and hence, we add these to the result as a final step.

3. The retrieval step, similarly to the PostgreSQL Batch solution for Q2, computes the component sizes, and then, the scores for all Comments are computed to retrieve the top-3 scored Comments (Listing 40).

3.14 GraphBLAS Batch

3.14.1 Tool description

GraphBLAS is a recently proposed standard built on the theoretical framework of matrix operations on semirings [57], which allows concise and portable formulation of graph algorithms [58]. The goal of GraphBLAS is to create a layer of abstraction between the graph algorithms and the graph analytics framework, separating the concerns of the algorithm developers from those of the framework developers and hardware designers. The GraphBLAS standard defines a C API [20] that can be implemented on a variety of hardware components including GPUs.

Data format An untyped graph can be represented as an adjacency matrix $\mathbf{A} \in \mathbb{N}^{|V|\times|V|}$, where rows and columns both represent nodes of the graph and element A_{ij} represents the number of edges from node i to node j. If the number of edges is not important, the adjacency matrix is defined over Boolean values, i.e. $\mathbf{A} \in \mathbb{B}^{|V|\times|V|}$, with $A_{ij} = 1$ if there is an edge from node i to node j and $A_{ij} = 0$ otherwise. *Bipartite graphs* can be represented with a non-square adjacency matrix $\mathbf{A} \in \mathbb{N}^{|V_1|\times|V_2|}$, where V_1 and V_2 are the sets of vertices in the two partitions. Typed graphs such as the ones used in this paper can be represented as using a bipartite adjacency matrix for each edge type, where V_1 represent the source nodes and V_2 represents the target nodes, e.g. $\mathbf{Likes} \in \mathbb{B}^{|users|\times|comments|}$. The graphs used in practical applications such as social networks are *sparse*. The adjacency matrices representing these graphs are also sparse, i.e. most elements in their adjacency matrix are zero values. These *sparse matrices* can be represented efficiently using matrix compression techniques such as CSR (Compressed Sparse Row).

A graph can also be stored as an incidence matrix $\mathbf{B} \in \mathbb{B}^{|V|\times|E|}$, where rows and columns represent nodes and edges (resp.). For undirected graphs, each column contains two 1 values in the positions of the source and the target nodes of the edge, and other elements are 0. Incidence matrices are sparse for all graphs with more than a few nodes.

Notation We follow the notation conventions of GraphBLAS as presented in [25]. Table 3 contains the list of GraphBLAS operations and methods used in this paper. Matrices are typeset in **bold** and start with an uppercase letter, e.g. **Friends**. Vectors are typeset in **bold** and start with a lowercase letter, e.g. **scores**. Additionally, sets are typeset in *italic* and start with a lowercase letter, e.g. *posts*.

Implementation For this solution, we used SuiteSparse:GraphBLAS [25], a parallel GraphBLAS imple-

Table 3 Notation of the GraphBLAS operations and methods used in this paper (based on [25])

Operation/method	Name	Notation
`mxm`	Matrix–matrix multiplication	$\mathbf{C}\langle\langle\mathbf{M}\rangle\rangle = \mathbf{A} \oplus.\otimes \mathbf{B}$
`vxm`	Vector–matrix multiplication	$\mathbf{w}\langle\langle\mathbf{m}\rangle\rangle = \mathbf{u} \oplus.\otimes \mathbf{A}$
`mxv`	Matrix–vector multiplication	$\mathbf{w}\langle\langle\mathbf{m}\rangle\rangle = \mathbf{A} \oplus.\otimes \mathbf{u}$
`eWiseAdd`	Element-wise addition,	$\mathbf{C}\langle\langle\mathbf{M}\rangle\rangle = \mathbf{A} \oplus \mathbf{B}$
	Union of patterns	$\mathbf{w}\langle\langle\mathbf{m}\rangle\rangle = \mathbf{u} \oplus \mathbf{v}$
`eWiseMult`	Element-wise multiplication,	$\mathbf{C}\langle\langle\mathbf{M}\rangle\rangle = \mathbf{A} \otimes \mathbf{B}$
	Intersection of patterns	$\mathbf{w}\langle\langle\mathbf{m}\rangle\rangle = \mathbf{u} \otimes \mathbf{v}$
`extract`	Extract submatrix	$\mathbf{C}\langle\langle\mathbf{M}\rangle\rangle = \mathbf{A}(I, J)$
	Extract subvector	$\mathbf{w}\langle\langle\mathbf{m}\rangle\rangle = \mathbf{u}(I)$
`apply`	Apply unary operator	$\mathbf{C}\langle\langle\mathbf{M}\rangle\rangle = f(\mathbf{A})$
		$\mathbf{w}\langle\langle\mathbf{m}\rangle\rangle = f(\mathbf{u})$
`select`	Select elements	$\mathbf{C}\langle\langle\mathbf{M}\rangle\rangle = \text{SELECT}(\mathbf{A}(i, j) == k)$
	equal to scalar k	$\mathbf{w}\langle\langle\mathbf{m}\rangle\rangle = \text{SELECT}(\mathbf{u}(i) == k)$
`reduce`	Reduce matrix to column vector	$\mathbf{w}\langle\langle\mathbf{m}\rangle\rangle = [\oplus_j \mathbf{A}(:, j)]$
	Reduce matrix to scalar	$s = [\oplus_{ij} \mathbf{A}(i, j)]$
`transpose`	Transpose matrix	$\mathbf{C}\langle\langle\mathbf{M}\rangle\rangle = \mathbf{A}^\mathsf{T}$
`extractTuples`	Extract (i, j, A_{ij}) tuples	$\{(i, j, A_{ij})\} \leftarrow \mathbf{A}$
	Extract (i, u_i) tuples	$\{(i, u_i)\} \leftarrow \mathbf{u}$
`build`	Build matrix from tuples	$\mathbf{A} \leftarrow \{(i, j, A_{ij})\}$
	Build vector from tuples	$\mathbf{u} \leftarrow \{(i, u_i)\}$
`nvals`	Number of non-empty elements	$nvals = \lvert\mathbf{A}\rvert$
		$nvals = \lvert\mathbf{u}\rvert$

Matrices and vectors are typeset in bold, starting with uppercase ($\mathbf{A}$) and lowercase ($\mathbf{u}$) letters, respectively. Scalars including indices (s, i, j) are lowercase italic, while index arrays (I, J) are uppercase italic. $\oplus$ and $\otimes$ are the addition and multiplication operators of an arbitrary semiring, and masks $\langle\langle\mathbf{M}\rangle\rangle$ are used to selectively write to the result. The *pattern* of a matrix/vector is the position of its non-empty elements

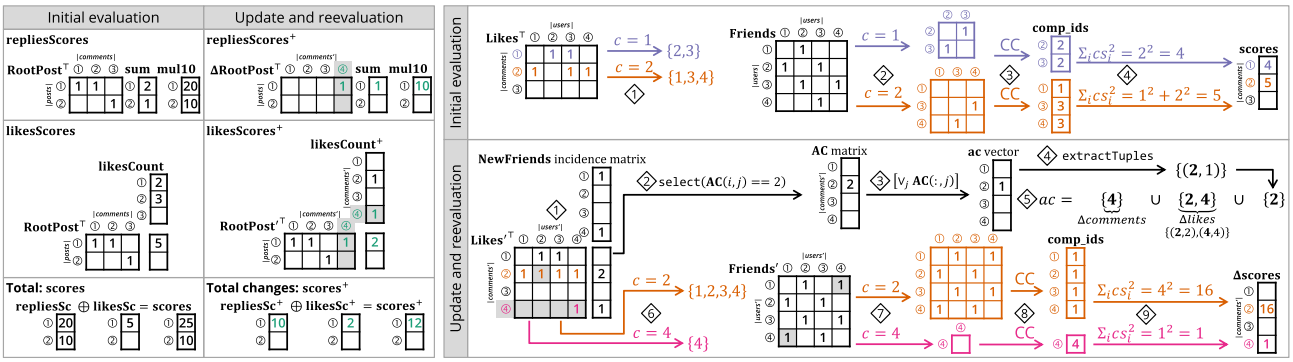
mentation and LAGraph [64], a library of GraphBLAS algorithms. We also implemented an incremental solution in GraphBLAS, described in Sect. 3.15.

3.14.2 Query 1

Q1 (Algorithm 1) computes the score for each Post, then selects the top-3 Posts. We use an auxiliary **RootPost**$^\mathsf{T}$ adjacency matrix to denote the Post a Comment belongs to (possibly through a series of Comments). The algorithm receives direct Comments on Posts (**CommentedP**$^\mathsf{T}$) and Comments on other Comments (**CommentedC**$^\mathsf{T}$) separately. Starting from the former (Line 7), BFS is used to find child Comments level by level and collect them to the Post they belong to until there are no more left (Lines 8-10). In Line 12, the row-wise summation of **RootPost**$^\mathsf{T}$ matrix produces the number of (direct and indirect) Comments per Post, and then, an `apply` operation multiplies the vector elements by 10. Line 14 sums up the number of likes the Post has via its Comments. For each Post, the **RootPost**$^\mathsf{T}$ adjacency matrix selects the elements of the **likesCount** vector corresponding to the Comments of the Post and then sums up the values. The score for each Post is the element-wise sum of the two vectors (Line 15). Figure 8a shows an example calculation.

Algorithm 1 GraphBLAS algorithm for Q1.

1: **Input**
2: **CommentedP**$^\mathsf{T} \in \mathbb{B}^{|posts|\times|comments|}$ ▷ adjacency matrix
3: **CommentedC**$^\mathsf{T} \in \mathbb{B}^{|comments|\times|comments|}$
4: **likesCount** $\in \mathbb{N}^{|comments|}$ ▷ # of incoming likes
5: **Output**
6: **scores** $\in \mathbb{N}^{|posts|}$
7: **RootPost**$^\mathsf{T}$ = **Next** = **CommentedP**$^\mathsf{T}$
8: **while** |**Next**| $\neq 0$ **do** ▷ selected Comments are not empty
9: **Next** = **Next** $\oplus.\otimes$ **CommentedC**$^\mathsf{T}$ ▷ child Comments
10: **RootPost**$^\mathsf{T}$ = **RootPost**$^\mathsf{T} \oplus$ **Next**
▷ collect all Comments to the Posts they belong to
11: **end while**
12: **sum** $= \left[\oplus_j \mathbf{RootPost}^\mathsf{T}(:, j)\right]$ ▷ row-wise sum
13: **repliesScores** $= 10 \otimes$ **sum** ▷ apply multiply-by-10 op.
14: **likesScores** = **RootPost**$^\mathsf{T} \oplus.\otimes$ **likesCount**
15: **scores** = **repliesScores** $\oplus$ **likesScores**
16: **return scores**



(a) Q1. **(b)** Q2. CC: connected components, **comp_ids**: component IDs, cs_i: size of component i.

Fig. 8 Execution of the GraphBLAS algorithms on the example graph (Fig. 5): the *Initial evaluation* step (used by both the batch and incremental solutions) and the *Update and re-evaluation* step (used by the incremental solution). Recall that the update in the example inserts the following relevant entities (highlighted with grey background): a friends edge between Users `u1` and `u4`, a likes edge from User `u2` to Comment `c2`, a Comment node `c4` with a root Post of `p1` and an incoming likes edge from User `u4`

3.14.3 Query 2

A batch evaluation example of Q2 (Algorithm 2) is depicted in the upper part of Fig. 8b. Q2 computes the score for each Comment and then selects the top-3 Comments. To collect the Users of each subgraph, in Lines 8 and 10, Step ① extracts the elements of the $\mathbf{Likes}^{\mathsf{T}}$ matrix as $(c, u, 1)$ tuples and collects them into sets of User IDs (u) per Comment (c). To produce the subgraph for each Comment, Step ② extracts a submatrix based on the Users selected (Line 11). Step ③ finds connected components in the induced subgraph using the FastSV algorithm [88] of the LAGraph library (Line 12). This produces a vector containing the component ID for each User. Step ④ yields the squared sum of component sizes, i.e. the score for each Comment (Line 14).

Algorithm 2 GraphBLAS algorithm for Q2.

```
1: Input
2:     comments = {1, ..., n}
3:     Likes^T ∈ B^{|comments|×|users|}
4:     Friends ∈ B^{|users|×|users|}
5: Output
6:     scores ∈ N^{|comments|}
7: function ScoreVecQ2
8:     likes_tuples ← Likes^T                                  ▷ ①
9:     for all c ∈ comments do
10:        users_c = {u : (c, u, _) ∈ likes_tuples}            ▷ ①
11:        Friends_c = Friends(users_c, users_c)               ▷ ②
12:        comp_ids = FastSV(Friends_c)                        ▷ ③
13:        scores(c) =
14:            Σ_{k∈comp_ids} |Select(comp_ids(i) == k)|^2     ▷ ④
15:    end for
16:    return scores
17: end function
```

3.15 GraphBLAS incremental

3.15.1 Tool description

To incrementalize the GraphBLAS Batch solution (Sect. 3.14), we have reworked the solution to compute changes to the results upon updates instead of running a full re-evaluation.

Approach The incremental version performs a full batch evaluation for the first run and computes the scores. Then, for each update only those parts of the model are re-evaluated which might be affected by the update. Finally, the previous top-3 scores and the new ones are compared to maintain the result. Incremental computation of Q1 uses fine-grained maintenance as it stores the scores of each Post and updates them when new Comment nodes and likes edges appear. The granularity of the incremental version of Q2 is coarser, as it collects the Comments whose scores can change and re-evaluates them.

Notation The updated variables are denoted with prime, e.g. the updated version of vector **scores** is **scores**′, which contains the scores for the new nodes and the updated scores for the existing ones. The changes can be stored as *increment matrices/vectors*, denoted with a superscript plus symbol ($^+$) and are applied with the $\oplus$ operation to the original values, e.g. $\mathbf{scores}' = \mathbf{scores} \oplus \mathbf{scores}^+$. Another option is to store the changed values, denoted with Δ, and apply them by overwriting the existing values: the new vector is initialized with the original one: $\mathbf{scores}' = \mathbf{scores}$, then the new values overwrite the existing ones via a mask, which preserves the unaffected values from modification: $\mathbf{scores}'\langle\langle\Delta\mathbf{scores}\rangle\rangle = \Delta\mathbf{scores}$.

3.15.2 Query 1

To incrementally evaluate Q1, Algorithm 3 updates the scores as follows. Lines 9 and 10 compute the increment of the score induced by new Comments. In Line 11, the number of likes the Comments newly received is summed up per Post. The two types of increments are summed up in Line 12. For subsequent evaluations, the scores are updated using the increment vector (Line 13). To find the top-3 scores, only the previous maximum values and the Posts with updated scores are considered. Line 14 yields the updated score values by assigning the $\mathbf{scores}'$ vector via the $\mathbf{scores}^+$ increment vector as a mask, which allows changes in the result only if the mask has an element at the corresponding position. Figure 8a shows an example calculation. Using the output of this algorithm, merging the previous top-3 scores and the new ones yields the new result.

Algorithm 3 GraphBLAS update algorithm for Q1.

1: **Input**
2: $\quad\mathbf{scores} \in \mathbb{N}^{|posts'|}$ ▷ previous scores
3: $\quad\mathbf{likesCount}^+ \in \mathbb{N}^{|comments'|}$ ▷ new incoming likes
4: $\quad\Delta\mathbf{RootPost}^\mathsf{T} \in \mathbb{B}^{|posts'|\times|comments'|}$ ▷ new root Posts[13]
5: $\quad\mathbf{RootPost}'^\mathsf{T} \in \mathbb{B}^{|posts'|\times|comments'|}$ ▷ all root Posts
6: **Output**
7: $\quad\Delta\mathbf{scores} \in \mathbb{N}^{|posts'|}$ ▷ only changed scores
8: $\quad\mathbf{scores}' \in \mathbb{N}^{|posts'|}$ ▷ all updated scores
9: $\mathbf{sum} = \left[\oplus_j \Delta\mathbf{RootPost}^\mathsf{T}(:, j)\right]$ ▷ # of new comments
10: $\mathbf{repliesScores}^+ = 10 \times \mathbf{sum}$
11: $\mathbf{likesScores}^+ = \mathbf{RootPost}'^\mathsf{T} \oplus.\otimes \mathbf{likesCount}^+$
12: $\mathbf{scores}^+ = \mathbf{repliesScores}^+ \oplus \mathbf{likesScores}^+$ ▷ increment vec.
13: $\mathbf{scores}' = \mathbf{scores} \oplus \mathbf{scores}^+$ ▷ update scores
14: $\Delta\mathbf{scores}\langle\langle\mathbf{scores}^+\rangle\rangle = \mathbf{scores}'$ ▷ collect updated scores
15: **return** $\Delta\mathbf{scores}, \mathbf{scores}'$

3.15.3 Query 2

The incremental evaluation of Q2 is depicted in the lower part of Fig. 8b. Algorithm 4 returns the Comments with new scores ($\Delta\mathbf{scores}$) by re-evaluating the Comments which the updates might impact on. Merging the previous top-3 scores and the new ones yields the new result. (New scores overwrite existing ones.) The first phase of the algorithm (Steps ①–⑤, Lines 14–20) collects the Comments which might be affected by the updates (affected comments, *acSet* set), and then, the second phase (Steps ⑥–⑨, Line 21) computes the new scores of these Comments using the batch algorithm described in Algorithm 2.

A Comment might be affected by an update if (1) it is a new Comment, or (2) the Comment receives a new incoming likes edge from a User, resulting in a new component or the expansion of an existing one, or (3) two Users who like the Comment become friends, which merges the components the Users belong to (if they previously belonged to different components). Case (3) is covered by Lines 14-18, where Steps ①–④ compute the Comments which might be affected by new friends edges. **NewFriends** incidence matrix represents each new friendship by a column having two 1-/valued elements for the two Users. For each new friendship (i.e. pair of Users), Step ① computes how many User of the pair likes each Comment (0, 1, or 2). During the matrix–matrix multiplication, each column of new friendships selects two columns of $\mathbf{Likes}'^\mathsf{T}$ matrix and sums them up into **AC** matrix (Line 15). In Line 16, Step ② keeps only 2-/valued elements, i.e. where both Users of a friendship liked the Comment, so both of them are present in the subgraph and the new friendship might merge components. Then, Step ③ produces a row-wise sum using the *logical or* operator $\vee$ (Line 17). Step ④ extracts $(c, 1)$ tuples from the result vector and collects the Comment IDs from these tuples, then Step ⑤ collects all the Comments which might be affected by the update (Line 19). Cases (1) and (2) are covered by Line 20, which produces the union of all the three cases. The next steps re-evaluate the scores of these Comments (Line 21).

Algorithm 4 GraphBLAS update algorithm for Q2 using the ScoreVecQ2 function of Algorithm 2.

1: **Input**
2: $\quad comments' = comments \cup \Delta comments$
3: $\quad\Delta likes \subseteq comments' \times users'$
4: $\quad\Delta friends \subseteq users' \times users'$ ▷ friend pairs in both orders
5: $\quad\mathbf{Likes}'^\mathsf{T} \in \mathbb{B}^{|comments'|\times|users'|}$
6: $\quad\mathbf{Friends}' \in \mathbb{B}^{|users'|\times|users'|}$
7: **Output**
8: $\quad\Delta\mathbf{scores} \in \mathbb{N}^{|comments'|}$
9: **Data**
10: $\quad\mathbf{NewFriends} \in \mathbb{B}^{|users'|\times\frac{1}{2}|1friends|}$ ▷ incidence matrix
11: $\quad\mathbf{AC} \in \{0, 1, 2\}^{|comments'|\times|1friends|}$ ▷ affected Comments
12: $\quad\mathbf{ac} \in \mathbb{B}^{|comments'|}$ ▷ Comments affected by new friends
13: $\quad acSet \subseteq comments'$ ▷ affected Comments, to reevaluate
14: $\mathbf{NewFriends} = \text{BuildIncidenceMatrix}(\Delta friends)$
15: $\mathbf{AC} = \mathbf{Likes}'^\mathsf{T} \oplus.\otimes \mathbf{NewFriends}$ ▷ ①
16: $\mathbf{AC} = \text{Select}(\mathbf{AC}(i, j) == 2)$ ▷ ②
17: $\mathbf{ac} = \left[\vee_j \mathbf{AC}(:, j)\right]$ ▷ ③
18: $acTuples \leftarrow\!\shortmid \mathbf{ac}$ ▷ ④
19: $acSet = \{c : (c, 1) \in acTuples\}$ ▷ ⑤
20: $acSet = \Delta comments \cup \{c : (c, u) \in \Delta likes\} \cup acSet$ ▷ ⑤
21: $\Delta\mathbf{scores} =$ ScoreVecQ2$(acSet, \mathbf{Likes}'^\mathsf{T}, \mathbf{Friends}')$ ▷ ⑥–⑨
22: **return** $\Delta\mathbf{scores}$

[13] Computed from previous $\mathbf{RootPost}^\mathsf{T}$ and new commented edges.

3.16 Differential dataflow

3.16.1 Tool description

Dataflow-based computational models were proposed to perform complex analytics on high-volume data sets: the *timely dataflow* [69,70] model targets batch processing, while its extension, *differential dataflow* [65], targets incremental processing.

We created two differential dataflow-based solutions:

.NET-based implementation Naiad[14] is a data processing prototype system developed at Microsoft Research[15] between 2011 and 2014. Naiad supports both the timely and differential dataflow computational models. The Naiad implementation was written in C#.

Rust-based implementation The computational models of Naiad have been implemented in the Rust programming language[16] [59] as two separate projects: Timely Dataflow[17] and Differential Dataflow.[18]

For the sake of brevity, we only discuss the Rust-based implementation here—the one using Naiad is available online but has been omitted from this paper.

Data representation To represent the graph, the Differential Dataflow solution uses unordered collections containing tuples with the following semantics:

- `comms`: (id, ts, content, creator, parent)
- `posts`: (id, ts, content, creator)
- `knows`: (user1, user2)
- `likes`: (user, comment)

3.16.2 Query 1

The implementation of Q1 (shown in Listing 28) uses the `comms`, `posts`, and `likes` collections. In Line 8, we extract the Comments and their direct parent objects, which can either be a Post or another Comment. In Lines 19–13, we determine the direct replies for each Post. Next, in Lines 14–25, we collect all transitive replies for each Post using the `iterate` operator, which executes the specified function until a stable result set is reached. In Lines 26–31, we collect the number of likes for each Comment, each having a cardinality of 1 (corresponding to their contribution to the overall score). Then, in Line 32, we collect each Comment that belongs to the Post with a cardinality of 10 (as they are worth 10 points). Next, in Lines 33–38, we concatenate the collections containing the likes and the reply Comments and count their total cardinality. Finally, in Lines 39–41, we add the timestamps of each Post and get the top-3 values.

```
1  fn query_1<G, Submission, User, Date>(
2    comms: &Collection<G, (Submission,Date,String,User,Submission)>,
3    posts: &Collection<G, (Submission,Date,String,User)>,
4    likes: &Collection<G, (User, Submission)>,
5    probe: &mut ProbeHandle<G::Timestamp>,
6  ) -> TraceAgent<OrdKeySpine<String, G::Timestamp, isize>>
7  {
8    let comms_parents = comms.map(|(id, _, _, _, parent)| (parent,
                id));(**)
9    let direct_replies = posts (**)
10     .map(|(post_id, _, _, _)| (post_id.clone(), ()))
11     .join_map(&comms_parents,
12       |post_id, _dummy, comm_id| (post_id.clone(), comm_id.clone())
13     ); (**)
14   let all_replies = direct_replies (**)
15     .iterate(|transitive| {
16       let comms_parents = comms_parents.enter(&transitive.scope());
17       let direct_replies = direct_replies.enter(&transitive.scope());
18       return transitive
19        .map(|(post_id, comment_id)| (comment_id.clone(),
                    post_id.clone()))
20        .join_map(&comms_parents,
21          |_parent_comment, post_id, child_comment| (post_id.clone(),
                      child_comment.clone())
22        )
23        .concat(&direct_replies)
24        .distinct();
25     });(**)
26   let liked_comments = likes (**)
27     .map(|(_user,comm)| (comm, ()))
28     .join_map(
29       &all_replies.map(|(post_id, comment_id)| (comment_id,
                  post_id)),
30       |_comment_id, _dummy, post_id| post_id.clone()
31     );(**)
32   let comms_themselves = all_replies.explode(|(post_id,
              _comment_id)| Some((post_id, 10))); (**)
33   let post_score = posts (**)
34     .map(|post| post.0)          // ensure all posts get a score
35     .concat(&liked_comments)     // likes contribute to posts
36     .concat(&comms_themselves)   // comments contribute to posts
37     .count()
38     .join(&posts.map(|post| (post.0, post.1.clone()))); (**)
39   let arrangement =  limit(&post_score, 3) (**)
40     .map(|vec| format!("{}|{}|{}", vec[0], vec[1], vec[2]))
41     .arrange_by_self(); (**)
42   arrangement.stream.probe_with(probe);
43   return arrangement.trace;
44 }
```

Listing 28 Differential Dataflow implementation of Q1.

3.16.3 Query 2

The implementation of Q2 (shown in Listing 29) uses the `comms`, `knows` and `likes` collections. First, in Lines 8–25, we use iterative label propagation to find connected components. Initially, we add the labels based on the `likes` edges and then propagate them across the `knows` edges. When the propagation reaches its fixed point, we will have a list of (user, label, comment) tuples, where each tuple shows that the given User is part of the connected component of the given Comment, and the smallest User id in this connected is the label. Then, in Lines 26–35, we aggregate the labels to get the size of each connected component and get the final score

[14] https://github.com/TimelyDataflow/Naiad

[15] https://www.microsoft.com/en-us/research/project/naiad/

[16] https://www.rust-lang.org/

[17] https://github.com/TimelyDataflow/timely-dataflow.

[18] https://github.com/TimelyDataflow/differential-dataflow.

by summing the squared component sizes. Finally, we add the timestamps of each Comment and get the top-3 values.

```
fn query_2<G, Submission, User, Date>(
  comms: &Collection<G, (Submission,Date,String,User,Submission)>,
  knows: &Collection<G, (User, User)>,
  likes: &Collection<G, (User, Submission)>,
  probe: &mut ProbeHandle<G::Timestamp>,
) -> TraceAgent<OrdKeySpine<String, G::Timestamp, isize>>
{
  let labels: Collection<_, ((User, Submission), User)> = (**)
  likes
    .filter(|_| false)
    .map(|(user, comm)| ((user.clone(), comm), user))
    .iterate(|labels| {
      let knows = knows.enter(&labels.scope());
      let likes = likes.enter(&labels.scope());
      labels
        .map(|((node, comment), label)| (node, (label, comment)))
        .join_map(&knows, |_node, (label, comment), dest|
                    ((dest.clone(), comment.clone()),
                    label.clone()))
        .concat(&likes.map(|(user, comm)| ((user.clone(), comm),
                    user.clone())))
        .reduce(|_key, input, output| {
          // only produce output, if 'input' contains '_key.0'
          if input.iter().any(|(label,_wgt)| *label == &_key.0) {
            output.push((input[0].0.clone(), 1));
          }
        })
    }); (**)
  let comm_score = labels (**)
    .map(|((_node, comment), label)| (label, comment))
    .count()
    .explode(|((_label, comment), count)| Some((comment, count *
                count)))
    .concat(&comms.map(|comm| comm.0.clone()))
    .count()
    .join(&comms.map(|comm| (comm.0.clone(), comm.1.clone())));
  let arrangement = limit(&comm_score, 3)
    .map(|vec| format!("{}|{}|{}", vec[0], vec[1], vec[2]))
    .arrange_by_self(); (**)
  arrangement.stream.probe_with(probe);
  return arrangement.trace;
}
```

Listing 29 Differential Dataflow implementation of Q2.

4 Classification

In this section, we compare the solutions according to our predefined classification criteria, evaluating general tool properties as well as case-specific applications of them. Aligned with the research questions, we used the following classification criteria:

Declarative query language To aid the development of the solutions using the respective tools, it is helpful for the tool to adhere to a standard declarative query language.

Data model For a usage integrated into a modelling tool set, ideally model query tools should be able to use models created and maintained in a modelling environment as they are, without adapters.

Explicitness of incrementalization Ideally, tools are able to incrementalize the query in an implicit manner, i.e. the developer only has to specify the query, but not how changes are propagated or how the query is broken down into chunks. This criterion does not apply to batch solutions, i.e. ones that recalculate the query from scratch after each model change.

Persistence To enable a failsafe operation, it is a desirable property for an analysis tool to continue the analysis even if the process is unexpectedly shut down. For this, data structures to save intermediate results have to be persisted, which is why we call this category query persistence. Further, in order to process models beyond the size of the main memory, it is desirable to persist the models in a database such that solutions do not have to have all model elements in memory. We refer to this as model persistence.

Parallelism As modern CPUs have multiple cores, it is desirable for a query technology to be able to make use of these resources by running parts of the query or its incremental change propagation in parallel.

Asymptotic complexity to propagate changes We present an estimated asymptotic complexity of the steps required to propagate changes. For this, we consider the necessary steps to propagate changes to both queries for all of the five possible changes (Sect. 2.3). The derived complexity values are specific to the particular scenario of the Social Media benchmark and thus should not be interpreted as a generic characterization of the tools.

In the following sections, we discuss the criteria for each of the solutions individually and summarize the results afterwards in Table 4, except for the asymptotic complexities that are depicted in Tables 5 and 6.

4.1 Declarative query language

NMF makes use of the SQO (Standard Query Operators), which is a common standard on the .NET platform. Similarly, the Xtend solution uses the standard collection operators of Java. ATL uses (a slightly adapted version of) OCL. The SQL and Neo4j solutions use the standard query languages of these databases: SQL:1999 and Cypher.

EOL used by Hawk is standard in Epsilon, but the community of Epsilon is much smaller than the community of relational databases, .NET or Neo4j.

AOF and YAMTL have their tool-specific query languages which are declarative (both solutions can be executed incrementally from the query specification) but not yet standards.

JastAdd uses its own language to specify synthesized and inherited attributes with tool-specific extensions like circular rules.

Table 4 Comparison of the tools used in the paper

Tool	Version	Data model	Engine	Solution	Decl.	Batch	Implicit	Explicit	DB	MV	Parallel
AOF	v201806	EMF	Java 8	Xtend	⊘	○	⊗	○	○	○	○
ATL	3.8.0	EMF	Java 8	ATL	⊗	⊗	○	○	○	○	○
ATL Incremental	v201904	EMF	Xtend	ATL/AOF	⊗	○	⊗	○	○	○	○
Differential Dataflow	0.11.0	relations	Rust	Rust	○	○	⊗	○	○	○	⊗
GraphBLAS	4.0.3	matrices	C	C++	○	⊗	○	⊗	○	○	⊗
Hawk	2.1.0	EMF	Java 8	EOL	⊘	⊗	⊘	⊗	⊗	⊘	○
JastAdd	2.3.5	EMF	Java 11	Java 11	○	⊗	⊗	○	○	○	○
Neo4j	4.2.4	property graph	Java 11	Java 11	⊗	⊗	○	⊗	⊗	⊗	○
NMF	2.0.169	NMF	C#	C#	⊗	⊗	⊗	○	○	○	⊗
PostgreSQL	12.4	relations	C	Java 11/SQL	⊗	⊗	○	⊗	⊗	⊗	○
Xtend	2.20.0	EMF	Java 8	Xtend	⊗	⊗	○	○	○	○	⊛
YAMTL	0.1.5	EMF	Java 11	Xtend	⊘	⊗	⊗	⊗	○	○	○

Data Model: the data model exposed to the user. *Engine:* programming language used to implement the engine (model transformation engine, database query engine, etc.), *Solution:* programming language and query language (if applicable) used to implement the solution, *Decl.:* the solution specified the queries using a declarative query language, *Batch:* only batch mode is supported, *Implicit:* implicit incrementalization is supported, *Explicit:* a solution with explicit incrementalization was implemented, *DB:* database-backed, i.e. the tool persists the model on disk after each transaction, *MV:* materialized views, *Parallel:* parallelization is supported. *Notation*—⊗ yes; ⊘ to some extent; ○ no; ⊛ yes, using Java 8 streams

Table 5 Asymptotic complexities to propagate a model update for Q1 in selected solutions

Solution	Add User	Add Post	Add Comment	Add like	Add friendship
NMF reference (batch)	$\Theta(p \log p + c)$	$\Theta(p \log p + c)$	$\Theta(p \log p + c)$	$\Theta(p \log p + c)$	$\Theta(p \log p + c)$
ATL (batch)	$\Theta(p \log p + c)$	$\Theta(p \log p + c)$	$\Theta(p \log p + c)$	$\Theta(p \log p + c)$	$\Theta(p \log p + c)$
Xtend	$\Theta(p \log p + c)$	$\Theta(p \log p + c)$	$\Theta(p \log p + c)$	$\Theta(p \log p + c)$	$\Theta(p \log p + c)$
PostgreSQL (batch)	$\Omega((p + c) \log cl + l)$	$\Omega((p + c) \log cl + l)$	$\Omega((p + c) \log cl + l)$	$\Omega((p + c) \log cl + l)$	$\Omega((p + c) \log cl + l)$
YAMTL-B (batch)	$\Theta(p + c)$	$\Theta(p + c)$	$\Theta(p + c)$	$\Theta(p + c)$	$\Theta(p + c)$
NMF	$O(1)$	$O(\log p)$	$O(h + \log p)$	$O(h + \log p)$	$O(1)$
Hawk (IU)	$O(n)$	$O(n)$	$O(n + p \log p)$	$O(n + p \log p)$	$O(n)$
JastAdd	$\Theta(p)$	$\Theta(p)$	$O(p + n)$	$O(p + n)$	$\Theta(p)$
AOF	$O(1)$	$O(\log p)$	$O(h + \log p)$	$O(h + \log p)$	$O(1)$
ATL (incremental)	$O(1)$	$O(\log p)$	$O(h + \log p)$	$O(h + \log p)$	$O(1)$
Differential Dataflow	$O(1)$	$O(\log p)$	$O(h + \log p)$	$O(\log p)$	$O(1)$
YAMTL-II (incremental)	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Hawk (IUQ)	$O(n)$	$O(n)$	$O(n + p \log p)$	$O(n + p \log p)$	$O(n)$
PostgreSQL (incremental)	$O(1)$	$\Theta(\log p)$	$\Theta(\log p + \log c)$	$\Theta(\log p \log c)$	$O(1)$
Neo4j (incremental)	$O(\log u)$	$O(\log p)$	$O(h + \log p + \log c)$	$O(h + \log puc)$	$O(\log u)$
YAMTL-EI (incremental)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

The solutions are categorized as *batch*, *implicit incrementalized* and *explicit incrementalized*

4.2 Data model

Given that the benchmark was originally presented at the TTC, it is not surprising that most solutions can operate directly on EMF models.

NMF uses its own metamodel NMeta but includes a transformation from Ecore and uses the same serialization format [45].

For JastAdd, it is necessary to reformulate the metamodel as grammar.

The solutions for Differential Dataflow, Neo4j, GraphBLAS and PostgreSQL do not operate on models directly but require dedicated adapters to feed the data from models into their input formats (CSV files).

Table 6 Asymptotic complexities to propagate a model update for Q2 in selected solutions

Solution	Add User	Add Post	Add Comment	Add like	Add friendship
NMF reference (batch)	$O(l \cdot f + c \log c)$	$O(l \cdot f + c \log c)$	$O(l \cdot f + c \log c)$	$O(l \cdot f + c \log c)$	$O(l \cdot f + c \log c)$
ATL (batch)	$O(l \cdot f + c \log c)$	$O(l \cdot f + c \log c)$	$O(l \cdot f + c \log c)$	$O(l \cdot f + c \log c)$	$O(l \cdot f + c \log c)$
Xtend	$O(l \cdot f + c \log c)$	$O(l \cdot f + c \log c)$	$O(l \cdot f + c \log c)$	$O(l \cdot f + c \log c)$	$O(l \cdot f + c \log c)$
PostgreSQL (batch)	$\Omega(l \cdot \log lf + l)$	$\Omega(l \cdot \log lf + l)$	$\Omega(l \cdot \log lf + l)$	$\Omega(l \cdot \log lf + l)$	$\Omega(l \cdot \log lf + l)$
YAMTL-B (batch)	$O(l \cdot f \cdot \alpha(l) + c)$	$O(l \cdot f \cdot \alpha(l) + c)$	$O(l \cdot f \cdot \alpha(l) + c)$	$O(l \cdot f \cdot \alpha(l) + c)$	$O(l \cdot f \cdot \alpha(l) + c)$
NMF incremental	$O(1)$	$O(1)$	$O(\log c)$	$O(f^2 + \log c)$	$O(l \cdot f + c \log c)$
Hawk (IU)	$O(c \log c)$	$O(c \log c)$	$O(c \log c)$	$O(f^2 + c \log c)$	$O(l \cdot f + c \log c)$
JastAdd	$\Theta(c)$	$\Theta(c)$	$\Theta(c)$	$O(c + f^2)$	$O(l \cdot f)$
AOF	$O(1)$	$O(1)$	$O(\log c)$	$O(f^2 + \log c)$	$O(l \cdot f + c \log c)$
ATL (incremental)	$O(1)$	$O(1)$	$O(\log c)$	$O(f^2 + \log c)$	$O(l \cdot f + c \log c)$
Differential Dataflow	$O(1)$	$O(1)$	$O(\log c)$	$O(f^2 + \log c)$	$O(l \cdot f + c \log c)$
YAMTL-II (incremental)	$O(1)$	$O(1)$	$O(1)$	$O(l \cdot f \cdot \alpha(l))$	$O(l \cdot f \cdot \alpha(l))$
Hawk (IUQ)	$O(1)$	$O(1)$	$O(1)$	$O(f^2)$	$O(l \cdot f)$
PostgreSQL (incremental)	$\Omega(l \cdot f)$	$\Omega(l \cdot f)$	$\Omega(l \cdot f)$	$\Omega(l \cdot f + c^2 f^2)$	$\Omega(l \cdot f + c^2 f^2)$
Neo4j (incremental)	$O(\log u)$	$O(\log p)$	$O(\log c)$	$O(u \cdot f + \log c)$	$O(c^2)$
YAMTL-EI (incremental)	$O(1)$	$O(1)$	$O(1)$	$O(f \cdot \alpha(l))$	$O(\alpha(l))$

The solutions are categorized as *batch*, *implicit incrementalized* and *explicit incrementalized*

4.3 Explicitness of incrementalization

The NMF reference solution, the ATL solution, the Xtend solution and the batch versions of SQL, Neo4j and GraphBLAS solutions are not incremental.

The incrementalization in the incremental NMF solution and in the AOF solution is implicit: the framework deducts a dependency graph from a model of the code that is available at runtime. This dependency graph is at the level of individual operations, so that only those operations really affected by a change need to be re-computed. However, these frameworks require data structures that explicitly support incrementalization. Both NMF and AOF already contain support for a wide range of common operations and allow Users to add new algorithms and data structures when needed, such as for calculating the connected components in Q2.

In the Hawk solutions, there are two aspects to incrementalization: the way that the persistent graph is updated and the way that the query is run. The batch solution uses derived attributes to cache the scores of Comments and Posts, which are implicitly updated incrementally if the nodes used to compute them change: however, the queries run in batch mode, and every change in the social network requires a full re-serialization to disk before the indexing process can update the graph. The incremental update solution uses the same approach for querying, but it does not re-serialize the social network model: instead, it applies the change sequences directly to the internal graph used for querying. The incremental update and query solution uses an explicitly incrementalized version of the query, using graph listeners to trigger the rescoring of Comments and Posts, and uses the same approach to directly update the underlying graph database from the NMF change sequences.

JastAdd and YAMTL support a completely implicit incrementalization by tracking all read operations to the model from which a dependency graph is set up. If the result of a rule depends on a read operation that is invalidated due to model changes, this attribute instance (JastAdd) or rule instance (YAMTL) is executed again. Therefore, the granularity of the incrementalization is given by the granularity of the attributes/rules.[19]

In the incremental versions of the SQL, Neo4j and GraphBLAS solutions, the incrementality is achieved explicitly. That is, these solutions all keep the scores of Posts or Comments through a table, additional nodes or vectors, respectively. Then, the solutions use explicit algorithms developed for the respective tool to update the scores in case of model updates.

In case of the SQL solution, this effort to explicitly reengineer existing queries for incremental change propagation is guided [38,81]. Such guidance can help the developer incrementalize a given query and could be automated when sufficient tooling becomes available.

The Differential Dataflow solution works similar to NMF and AOF. However, the notion of time during the calculation of the query allows this tools to support temporary state,

[19] YAMTL can also expose model updates affecting a rule instance so that these can be explicitly handled with fine-grained incrementality.

in particular the `iterate` call in Listing 29. This temporary state allows to support also more complex cases such as computing *connected components* in Q2 using only built-in algorithms and data structures [70].

In total, we have nine non-incremental solutions. PostgreSQL, Neo4j, YAMTL-EI and GraphBLAS are explicitly incrementalized, so developers have to approach the problem conceptually differently than in the batch solution. Hawk provides slight implicit incrementalization capabilities, but much of the solution must be provided by the developer in order to get the solution incremental; therefore, we categorized it as an explicit incremental tool. In NMF, JastAdd, AOF, YAMTL-II, Incremental ATL Incremental and Differential Dataflow, the incrementalization happens entirely implicitly, though partially with restrictions. Namely, JastAdd and YAMTL require the developer to adapt to their programming model, whereas NMF and AOF restrict the used operations to those that are supported in the framework, even if this set is extensible. Differential Dataflow does not have such restrictions.

4.4 Persistence

Most of the presented solutions keep the data only in memory without any features for fault-tolerance. This means that the solutions require a separate tooling to ensure a failsafe operation.

The exceptions to this are Hawk, Neo4j and PostgreSQL, which make use of database systems internally and are therefore durable, in the sense that models and intermediate query results for their incremental variants are available even if the application gets terminated unexpectedly.

For many of the other tools, model persistence could be achieved through persistence systems such as Eclipse CDO.[20] However, although the model sizes used in the scope of the benchmark went up to a few million model elements and connections, solutions were always able to handle the models in main memory. Therefore, the need for solutions with model persistence to hit the disk for an update led to performance penalties in comparison with other solutions.

The advantage that Hawk and the incremental versions of Neo4j and PostgreSQL take out of the persistence of the intermediate results is that after a restart, they do not require the performance penalty of an initial run again and can process updates straight away after a restart. The exception to that is the incremental update and query solution for Hawk, since this keeps an in-memory list of the top-3 elements.

[20] https://www.eclipse.org/cdo/.

4.5 Parallelism

Since GraphBLAS is based on matrix operations, it is well suited for parallel processing. Differential Dataflow also supports executing the query in parallel. The Xtend solution makes use of the Java streams feature to enable parallelism, but only on a task level, i.e. the solution runs different parts of the query in a pipeline.

NMF also has some built-in support for parallelism, but it is restricted only to the incremental change propagation. The initial query execution does not profit from the parallel execution.

The other solutions do not make use of parallelism.

4.6 Asymptotic complexity to propagate changes

Recall that in Sect. 2.3, the following change operations were presented: (1) a new User is added, (2) a new Post is added, (3) a new Comment is added to an existing Post, (4) a Comment is liked, and (5) two existing Users become friends.

In the remainder of this section, we denote the number of Users in a model with u, the number of Posts with p, the number of Comments with c and the number of likes with l. A Post may have up to n Comments including transitive replies. The maximum hierarchy depth of Comments is denoted by h. A group of Users that like the same Comment and are connected through friendships has the maximum size f.

4.6.1 NMF Batch, Xtend, ATL

The NMF reference solution and the ATL solution are batch implementations where all changes result in a complete recalculation of the analysis. For Q1, calculating the scores of each Post takes $\Theta(p + c)$ time, plus $\Theta(p \log p)$ for the sorting. This amounts to $\Theta(p \log p + c)$. For Q2, Tarjan's algorithm or a similar algorithm to obtain connected components is executed for each Comment. Tarjan's algorithm has a linear complexity in both nodes and edges where there are at most f^2 edges, leading to a complexity of $O(c \cdot f^2)$. However, because Users only need to be considered for Comments they have liked and every like corresponds to exactly one node in the induced graph for the liked Comment, this can be reduced to $O(l \cdot f)$. Together with $O(c \log c)$ for sorting the Comments, this leads to $O(l \cdot f + c \log c)$ for Q2. For Xtend, the sorting step is avoided by a dedicated stream operator in both queries. Furthermore, the usage of streams in the Xtend solution theoretically allows to calculate the scores of the individual Comments or Posts in parallel.

4.6.2 NMF incremental, AOF, ATL incremental, differential dataflow

In the incremental NMF solution as well as in the AOF, incremental ATL and Differential Dataflow solutions, the analysis is slightly more complex. Adding a new User does not touch the dependency graph for Q1, and thus, the effort for change propagation is constant, same for the change that two Users become friends. When a new Post is added, it does not have any Comments, but has to be inserted in the binary search tree of Posts. This has an effort of $O(\log p)$. Adding a new Comment will ripple through the `Descendants` or `AllInstances` operation and cause an effort of $O(h)$, followed by $\Theta(\log p)$ to update the position of the Post in the binary search tree. When a Comment is liked, this only causes constant effort to update the sum, but $O(h)$ to find the corresponding Post and $\Theta(\log p)$ to update the position in the search tree.

For Q2, the insertion of a new User does not touch the dependency graph because the new User does not have friends and has not liked any Comments yet. Similarly, the insertion of a Post only causes constant effort to the change propagation. The insertion of a Comment only requires to calculate connected components of an empty graph (the Comment is not liked, yet) and inserting it into the binary search tree, which takes $O(\log c)$. Liking a Comment requires to re-compute the connected components for this Comment and updating the score in the search tree, which is $O(f^2 + \log c)$ since the connected components are computed from scratch. If two Users become friends, this potentially changes the subgraphs for all Comments, which has a complexity of $O(l \cdot f + c \log c)$ just as if we were to recalculate the entire query, though in the average case, many new friendships will likely not affect all Comments.

The parallel execution of NMF and Differential Dataflow does not change the complexity to perform a change, and it is rather used to propagate the changes within the transaction in parallel.

4.6.3 Hawk

In Hawk, the batch update process needs to check the entire model for changes, which clearly is $\Omega(p + c + u \cdot f)$. For incremental updates, the time to perform the graph updates is constant. However, in both cases, the indices need to be re-computed. For Q1, this takes $\Theta(n)$ time plus $O(p \log p)$ for sorting the Posts as soon as the score for a Post needs to be re-computed, i.e. when adding or liking a Comment. The incremental query eliminates the sorting step, explicitly taking advantage of the monotony of the scores in this scenario.

For Q2, similarly to the incremental NMF and AOF solutions, the insertion of a User or a Post does not have any effect because they do not invalidate the score of any Comment. However, the solution still requires the sorting, which takes $O(c \log c)$, unless the solution is run in the incremental query mode (in which case the sorting is replaced by constant effort, using the monotony of the scores). If a Comment is liked, the calculation of connected components is invalidated and recalculated for this Comment, which takes $O(f^2)$. A new friendship requires the score of all Comments to be re-evaluated, which is complexity of $O(l \cdot f)$.

4.6.4 JastAdd

JastAdd tracks accesses to model elements when an attribute (score) of a model element is computed and uses this to invalidate these attributes upon a model change. However, the maximum always has to be computed. Adding a new User or two Users becoming friends in Q1 (changes that do not affect any model feature used in computing the scores) has a complexity of $\Theta(p)$. This complexity also dominates the score calculation for a new Post (without Comments). Adding or liking a Comment will invalidate the score for the Post and cause JastAdd to re-compute the score for this Post, which requires a complexity of $O(n)$ in addition to the $\Theta(p)$.

For Q2, finding the maximum takes $\Theta(c)$ and for inserting a User or Post, that is the overall complexity because the new User or Post is not being read in any score calculation. A new Comment only requires to calculate the score for that Comment in addition, which takes only constant effort. Liking a Comment will invalidate the score for this Comment, which again takes $O(f^2)$ and a new friendship takes $O(l \cdot f)$ as this friendship may affect the score for many Comments.

4.6.5 YAMTL

In the YAMTL solutions, only the three best candidates are retained and this simplifies the sorting step, which becomes constant. Discarding a candidate has constant time, and when the candidate is valid, it takes up to two comparisons to update the list.

In Q1, YAMTL-B calculates the scores by traversing all Posts and then all of their Comments, $O(p + c)$. When adding Comments or likes, YAMTL-II re-computes the score for each matched Post affected by a change, traversing all of its contained Comments, $O(n)$. Instead, YAMTL-EI only processes the new Comment that has been added to the Post or that has been liked by another User for each impacted Post, so the update is $O(1)$ in those cases.

In Q2, for each Comment, Sedgewick and Wayne's *weighted quick union find with path compression* initializes components in linear time, $O(l \cdot f)$, where the initial size of components is given by l. Find and union operations have an amortized cost that corresponds to the inverse of Ackermann's function, $O(\alpha(l))$. YAMTL solu-

tions match Comments instead of Posts in Q2. For computing a Comment score, YAMTL's solution applies the union operation for any friend that has liked the Comment, so processing each Comment involves $O(l \cdot f \cdot \alpha(l))$. In propagation mode, YAMTL-B initializes the data structures for the weighted quick union find with path compression for Users l in each Comment, and the cost is $O(l \cdot f \cdot \alpha(l) + c)$. YAMTL-II visits the Comments affected by changes and initializes the connected components of friends from scratch, so the cost is $O(l \cdot f \cdot \alpha(l))$ when adding a like and when adding a friendship. YAMTL-EI is sensitive to finer changes, adding a like by a new User triggers the update of connected components with the friends f of the new User, $O(f \cdot \alpha(l))$, whereas a new friendship only involves computing the union of two components, $O(\alpha(l))$.

4.6.6 PostgreSQL batch

Compared to solutions working on object models, an RDBMS has the problem that references to other models are not directly available as object references but usually lead to an indirect reference that must be resolved using an index. The choice of this index and also the exact query strategy is generally specific to the RDBMS and may differ from our estimation here. In the remainder, we assume search trees are used that resolve such a reference in log time. This allows to manage data beyond main memory limitations but is of course slower than following a direct reference.

For Q1, the root Post reference for each Comment is computed first in $\Omega(c \cdot \log c)$, which stands for the cost of the first self-join of Comments when computing the transitive closure. Then Posts are outer joined with their Comments in $\Omega(p \cdot \log c + p + c)$ steps to produce an interim result size proportional to $p + c$, which is then outer joined with their likes in $\Omega((p + c) \cdot \log l + p + c + l)$ and the count of likes has to be computed in $\Omega(l)$ steps. In-memory solutions, by contrast, can easily read the count as it is usually directly available for array list implementations of collections. Thus, the batch implementation has a complexity of $\Omega(c \cdot \log c + p \cdot \log c + (p + c) \cdot \log l + p + c + l)$, which can be further simplified to $\Omega((p + c) \cdot \log cl + l)$.[21]

In Q2, we first compute the transitive closure of the friendship subgraphs[22] in $\Omega(l \cdot \log lfl + l + f + l)$, which stands for the cost of the first recursive step. The size of the transitive closure is $\Omega(l)$, which is fed as input to the subsequent aggregations that can be done in linear time w.r.t. their input size. The result of the aggregations has again a size of $\Omega(l)$, and as it exists only in memory, no index is available. Thus, the final join to Comments is assumed to be done using hash join with a complexity of $\Omega(c + l)$. Summing up the former gives the batch implementation a complexity of $\Omega(l \cdot \log lfl + l + f + l + l + c + l)$, which can be simplified to $\Omega(l \cdot \log lf + l)$.[23]

4.6.7 PostgreSQL incremental

For Q1, any score update of a Post in the incremental SQL solution yields a complexity of $\Theta(\log p)$. However, adding a new User or friendship does not cause a score update and has therefore constant effort. Apart from that, adding a Post only requires to left join an empty table with constant effort and migrating the Post to the new partition, which we assume is constant effort. Adding a Comment requires to resolve its Post with a complexity of $\Theta(\log p + \log c)$, adding a like causes $\Theta(\log p \cdot \log c)$. For Q2, the result-retrieval is non-trivial even in the presence of an index and has an effort of $\Omega(l \cdot f)$. Adding a User, Post or Comment changes neither *friends* nor *likes* relations, which is why all join operations to update the *comment_friends* relation in stages 0 and 1 inner join an empty table and thus become constant in the maintenance phase. Computing stage 2 and inserting into the *comment_friends* relation require to process all $O(c^2 f^2)$ entries of stage 2. Liking a Comment or a new friendships adds an overhead to the computation of stage 1 and stage 2 (possibly recursively), but we assume this effort to be dominated by $O(c^2 f^2)$.

4.6.8 Neo4j incremental

The Neo4j solution saves the score as a property on Post and Comment nodes and indexes these properties using B-trees.[24] For Q1, this implies that inserting a new Post necessitates a cost of $O(\log p)$ (maintaining the index on the score property). Adding a Comment or a like can result in an increase of the score of the root Post, which costs $O(h)$ for finding the Post and $O(\log p)$ to update the score. For Q2, the incremental solution materializes each Comment's connected components which need to be maintained upon inserting a new likes edge. The non-trivial operations are as follows. When a Comment is initially inserted, it has a score of 0. Maintaining the index on the score property incurs a cost of $O(\log c)$. When a User usr adds a new like to Comment cmt, the solution merges all connected components of cmt containing friends of usr with a cost of $O(u \cdot f)$. Additionally, maintaining the index on the score

[21] This assumes an optimal query execution under the assumption that $p < c$ and $p < l$.

[22] In Listing 30, this is composed of two subqueries for clarity, i.e. comment_friends and comment_friends_closed, but we assume the former is inlined for execution.

[23] This assumes an optimal query execution under the assumption that $f < l$ and $c < l$.

[24] https://neo4j.com/docs/operations-manual/4.1/performance/index-configuration/.

property of the Comments has $O(\log c)$ cost. Finally, adding a new friends edge necessitates checking which components of which Comments can be merged, incurring a cost of $O(c^2)$ and maintaining the index for cost $O(\log c)$.

Apart from this, since Neo4j indexes store identifiers of nodes in search trees and checks uniqueness constraints, adding a User requires an effort of $\Theta(\log u)$, adding a Post $\Theta(\log p)$ and adding a Comment $\Theta(\log c)$. Liking a Comment requires at least $\Theta(\log u + \log c)$ and Users becoming friends at least $\Theta(\log u)$ cost.

4.6.9 GraphBLAS

For GraphBLAS, stating algorithmic complexities for query calculation is difficult because the engine automatically chooses from a variety of algorithms for matrix multiplication given that the matrices are usually very sparse. For example, the adjacency matrix to calculate the Comments of a root Post in the Q1 batch version is sparse because Comments only belong to one root Post. A naïve implementation of calculating this sum would therefore take an effort of $O(p \cdot c)$, but this analysis does not take into account the optimizations that GraphBLAS applies to this computation. Therefore, we do not provide results for asymptotic complexity for GraphBLAS.

4.6.10 Summary

The asymptotic complexities for Q1 are summarized in Table 5. The table is divided into solutions that are not incremental at all at the top (including non-incremental executions of tools that are able to implicitly incrementalize the query), those that are implicitly incremental and derive the algorithm to propagate incremental changes from a declarative specification in the middle, and explicitly incremental solutions at the bottom.

One can see in Table 5 that whereas the non-incremental tools have a linear complexity with respect to the number of Posts and Comments, many incremental tools have a better asymptotic complexity and therefore should scale much better in presence of changes as they have constant or logarithmic update efforts. However, there is a notable difference between NMF and AOF, which take a rather fine-grained approach to incremental change propagation, compared to JastAdd, Hawk and YAMTL that rely on chunks in the form of (derived) attribute or rule calculations. While the latter can speed up recalculation to constant times in some cases, the former approaches can get down to logarithmic efforts even when Comments are added or liked as they allow partial recalculation of the score of a Comment. These complexities also could be beaten by solutions where the incremental change propagation is explicitly specified by the developer, at least using the approaches studied.

The results for asymptotic complexities for Q2 are summarized in Table 6. The solutions are ordered in the same way as in Table 5. While for adding Users, Posts or Comments, the results look very similar to Q1 with incremental approaches achieving a strictly better asymptotic complexity, the results for liking a Comment or especially Users becoming friends are not as good. This is partially because the analysis of asymptotic complexities is not very detailed,[25] but it is also a consequence of the effect that these changes simply have a large impact on the computations that get invalidated. A new friendship changes the connected components in the induced subgraphs of the Comments that both Users have liked.

5 Performance evaluation

In this section, we present the results of the performance measurements with respect to the time required to load the models, to run the initial query evaluation, and to propagate changes. We first present the benchmark setup and experiment design and then analyse the results. Finally, we discuss potential threats to validity.

5.1 Benchmark setup

5.1.1 Input models

We executed the benchmark on models of increasing sizes, denoted by scale factors (SFs) of power of 2. The number of elements per node/edge type in each SF is shown in Table 7. The largest model has 0.86M nodes and 2.25M edges. The number of changes varies between 45 and 132 model elements.

5.1.2 Benchmark framework

The benchmark framework is based on the one of the TTC 2017 Smart Grid case [48] and supports automated build and execution of solutions as well as a correctness check and visualization of the results using R. The correctness is checked by comparing the query result against a pre-computed reference both after the initial transformation and after each update. The source code and documentation of the benchmark as well as metamodels, solutions, input models and change sequences used for benchmarking are publicly available online.[26] The benchmark repository also contains instructions on how to

[25] A more detailed analysis might take the maximum number of Comments a User has liked into account, which is certainly smaller than c.

[26] https://github.com/TransformationToolContest/ttc2018liveContest.

Table 7 Model sizes for each scale factor: number of nodes and edges, number of changes

Type\scale factor	1	2	4	8	16	32	64	128	256	512	1024
Comments	640	1064	2315	5056	9220	18,872	39,212	76,735	148,470	273,418	540,905
Posts	554	889	1845	2270	5518	10,929	18,083	37,228	74,668	167,299	314,510
Users	80	118	190	204	394	595	781	1158	1678	2606	3699
Total number of nodes	1274	2071	4350	7530	15,132	30,396	58,076	115,121	224,816	443,323	859,114
friends	53	102	262	298	904	1827	2752	5695	11,118	24,387	45,386
replyTo	640	1064	2315	5056	9220	18,872	39,212	76,735	148,470	273,418	540,905
likes	6	24	66	129	572	1598	4770	13,374	36,815	102,276	268,432
submitter	1194	1953	4160	7326	14,738	29,801	57,295	113,963	223,138	440,717	855,415
Total number of edges	2533	4207	9118	17,865	34,654	70,970	143,241	286,502	568,011	1,114,216	2,251,043
Total number of changes	67	120	132	104	110	117	68	86	45	112	74

run the benchmark in Docker containers and there are Docker images of all solutions available in Docker Hub.[27]

5.1.3 Benchmark environment

We ran the solutions on a cloud virtual machine with 8 cores of an Intel® Xeon® Platinum 8167M CPU, a base clock speed of 2.0 GHz and a turbo clock speed of 2.4 GHz. Hyper-Threading was turned off. The machine was running the Ubuntu 20.04.2 LTS operating system. To help reproducibility, the experiments were executed in containers managed by Docker 20.10.6. The runtime environments for Java- and .NET-based solutions were OpenJDK 1.8.0 update 282, OpenJDK 11.0.10 and .NET Core 3.1.14. Each tool was measured 5 times, and the geometric mean of the results is used. The timeout value for each run was set to 10 minutes.

The modelling tools used XML-based representations (e.g. XMI files) to load the data and the change set sequences. For the rest of the tools (Neo4j, GraphBLAS, PostgreSQL and Differential Dataflow), both the initial model and the change sequences were loaded from CSV files.

5.2 Analysis

We grouped execution times by query, tool family and phase. To save space, we do not show the exact results, but these can be obtained from the GitHub repository of the benchmark. From the phases listed in Sect. 2.4, we omitted the initialization phase and kept the other three, i.e. (1) loading the models, (2) the initial run and (3) the time to apply a set of changes and update the result accordingly. In the following, we compare the solutions across tool boundaries. For each family of solutions, we only include the results for the best variant. A comparison of the solution variants for NMF, Hawk, JastAdd and YAMTL can be found in the appendix.

5.2.1 Batch solutions

The results for all batch tools are depicted in Fig. 9. One can immediately see that most lines in all of these diagrams have approximately the same slope. This confirms that query execution times grow approximately linearly with the size of the models.

The results for the initial execution times and the times to update the graph are also very similar. This is clear, given that after changing the model, the batch solutions all evaluate the entire query again. Particularly for small models, there is an overhead attached with the first execution, which is why the execution times for updates are slightly smaller. This overhead differs between the tools and is constant, thus hard to perceive on a log scale.

For the update times, one can see clear distances between the graphs, especially for Q1 where plotted execution times appear almost parallel, indicating constant factors, i.e. the solutions have different execution speeds but very similar scalability characteristics. The GraphBLAS batch solution outperforms all other batch solutions by more than one order of magnitude. JastAdd and YAMTL that come next are still faster than NMF by a factor of 3 and faster than the remaining by at least an order of magnitude for Q1.

For Q2, many solutions had severe performance problems which is why they did not complete for larger model sizes. One can also see that in particular for the SQL solution and the JastAdd solution, the slope is steeper for the larger model sizes, indicating the chosen algorithm implementation does not scale equally well.

For Hawk, one can see how the choice of the underlying database implementation affects performance: the perfor-

[27] https://hub.docker.com/r/ftsrg/ttc2018/tags.

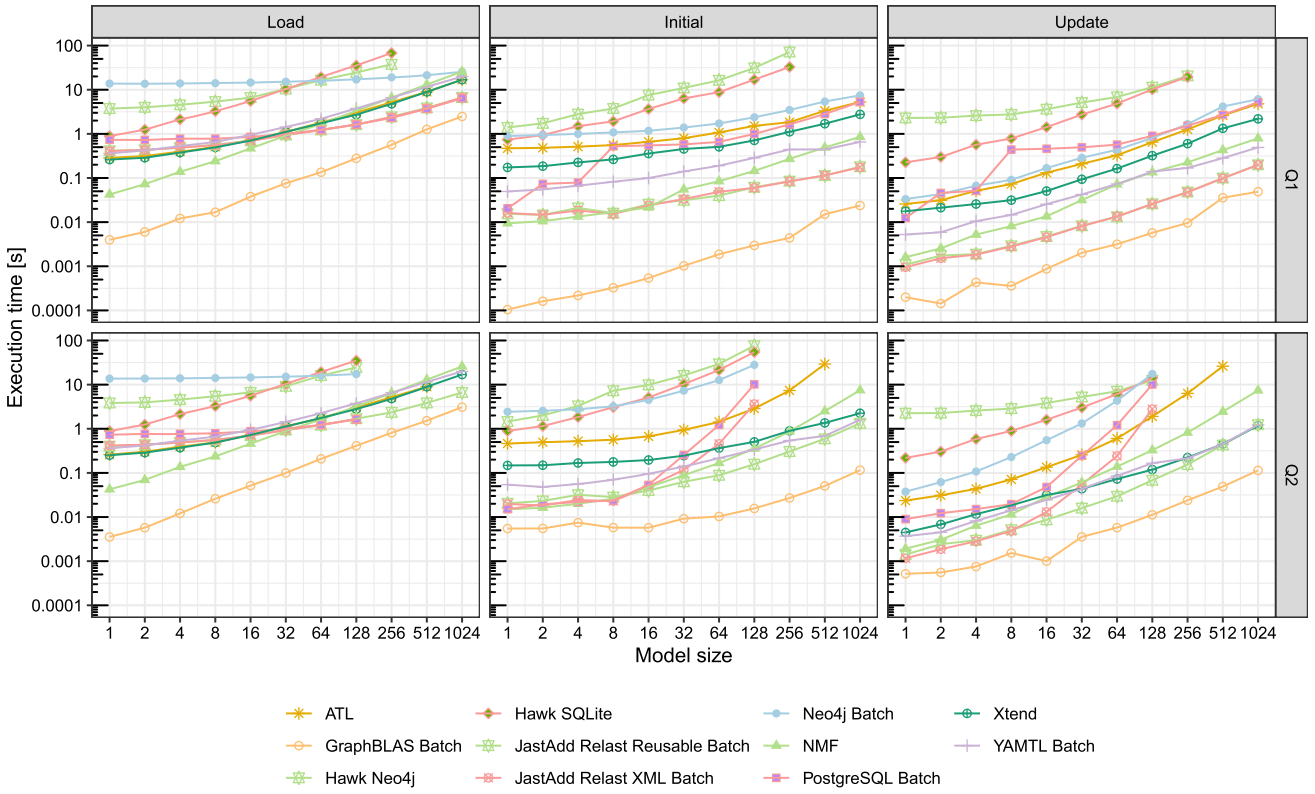


Fig. 9 Execution times of solutions with batch evaluation

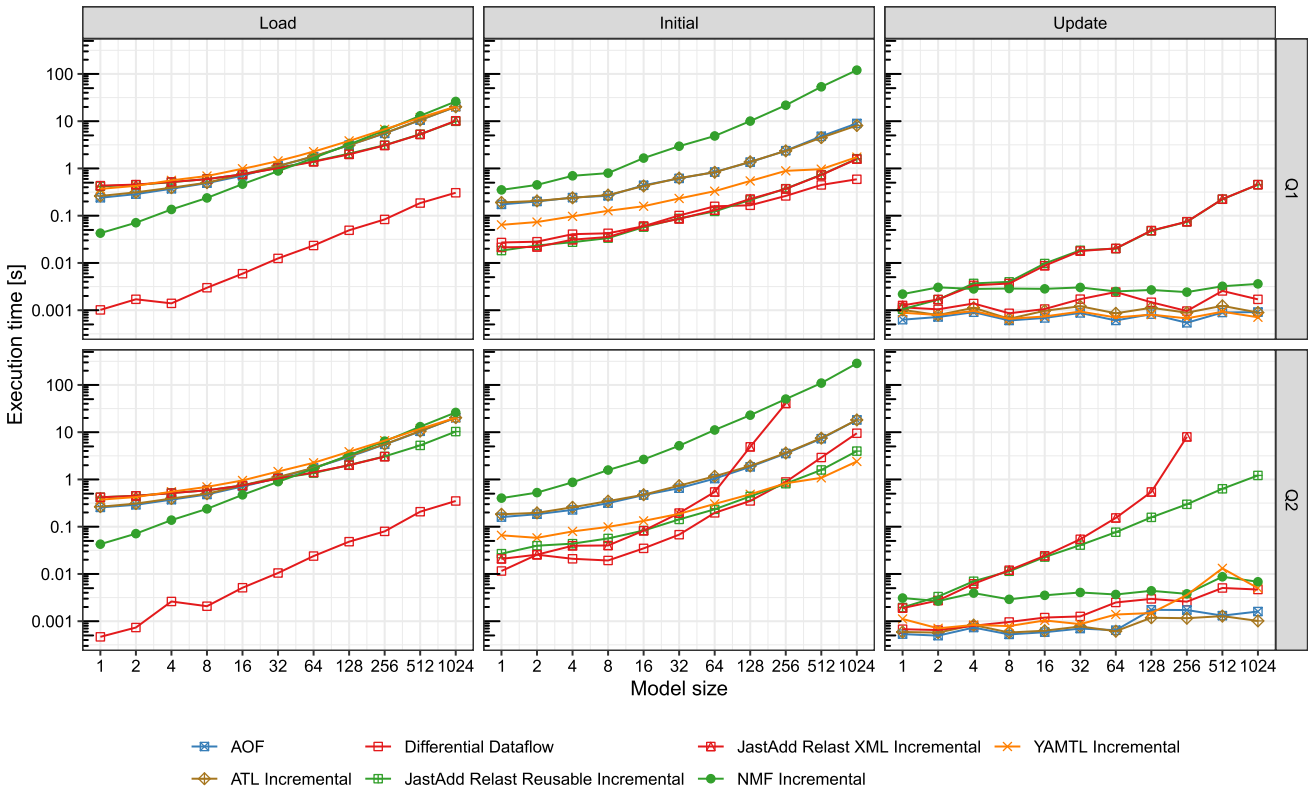


Fig. 10 Execution times of solutions with implicit incremental evaluation

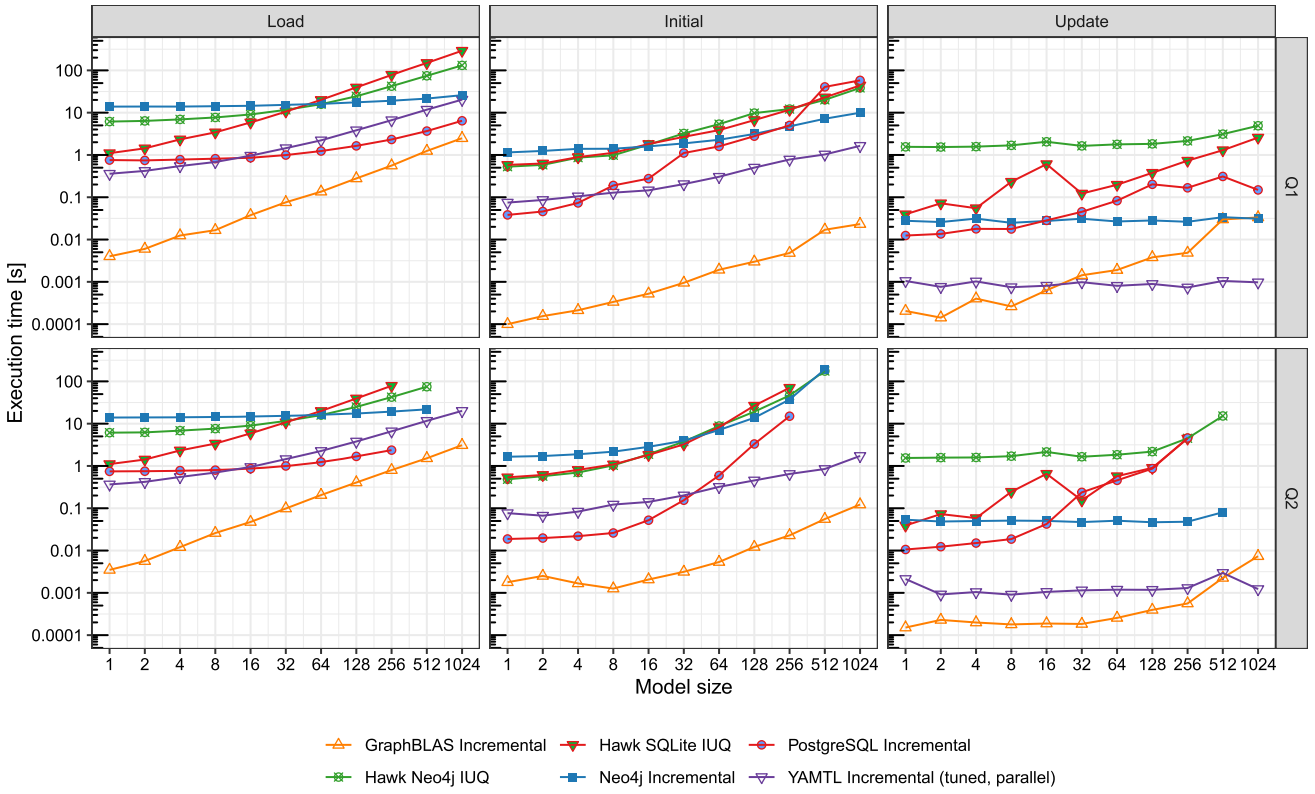


Fig. 11 Execution times of solutions with explicit incremental evaluation

mance using SQLite is mostly better, which is clear given that the entire models for all sizes fit into main memory.

5.2.2 Implicit incremental solutions

For the implicitly incremental solutions, the results depicted in Fig. 10 look very different. While one can see a slope in the load times and in the initial query evaluations, the times to propagate a change sequence appear constant across model sizes for all solutions except JastAdd. These solutions require few milliseconds or less to propagate the changes even for the largest input model sizes.[28] For JastAdd, the execution times for an update have the same slope as for the initial execution, so the incremental change propagation cannot reduce the complexity but allows a constant speedup (or even slowdown, cf. Sect. A.2.3).

Because the Differential Dataflow solution was the only solution in this category that is not using a modelling framework underneath but uses plain CSVs as inputs, it is faster in loading the models by multiple orders of magnitude. This is due to the fact that the parsers used in the modelling tools (EMF or NMF) both work with a much more diverse set of inputs than CSV and therefore have a much higher complexity. Furthermore, both modelling frameworks induce an overhead to the model elements.

With regard to the initial execution, one can see that the slope for AOF and the incremental ATL solution is smaller than for the others. Apparently, AOF has a bigger constant overhead that gets less important as models grow. In contrast, the NMF solution has an initial time that is more than an order of magnitude slower than the other solutions. Apart from this, one can observe that Differential Dataflow, JastAdd and YAMTL are the fastest in the initial run.

5.2.3 Explicit incremental solutions

Surprisingly, the results for the explicitly incremental tools depicted in Fig. 11 are worse than for the implicitly incremental tools in the sense that they all have a slope, i.e. the execution times to update the query results grow with growing model sizes. The only exception here is YAMTL, where the incrementalization itself is obtained implicitly but is manually and explicitly tuned. For GraphBLAS, this seems not critical for the model sizes benchmarked because the solution is faster than the others by multiple orders of magnitude and the time to propagate the changes is still below the tenth of a second. For all the other solutions, however, the scalability is much worse than for most of the implicit incremental

[28] The NMF solution appears to be slower than a millisecond, but this is only due to the fact that the first *Update* phase includes a just-in-time compilation which takes about 10–50 ms.

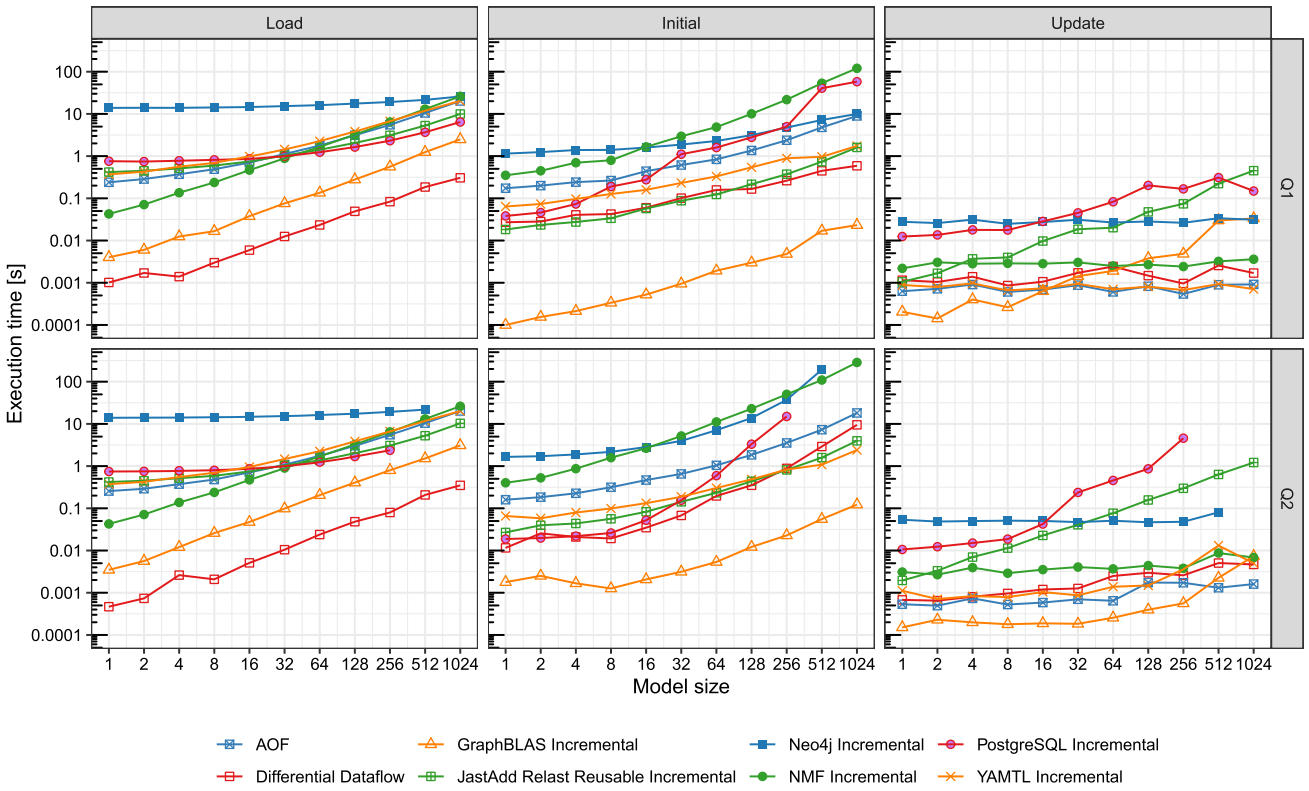


Fig. 12 Execution times of selected incremental solutions

tools: they run out of memory and have runtimes of multiple seconds.

This result is surprising because one would expect that the incrementality of solutions, i.e. to which extent the update times depend on the model sizes depend on the input size, would be better if the solutions are explicitly developed for this use case. However, the results imply the opposite. This is because the implicit incremental tools create very fine-grained dependency graphs that allow implicit incremental tools to keep the effort of propagating a change to a minimum, whereas the explicit solutions apply rather coarse-grained schemes and need to re-evaluate larger parts of the query.

For Hawk, the differences between the Neo4j and the SQLite backend are more severe than in the batch version. While Neo4j is slower, particularly in the *Update* phase, it is capable of processing the largest model, while the SQLite backend runs out of memory.

5.2.4 Selected incremental tools

To allow a better comparison between the implicitly and explicitly incremental solutions, we plotted a selected subset of the incremental tools in Fig. 12. The results show again large differences in the runtime for the initial query computation of about three orders of magnitude. The load times show the differences between solutions that operate on plain CSVs and those that use a modelling framework or have to set up a database.

The results for the change propagation just confirm what we found in the previous sections: while the explicitly incrementalized solutions have a slope, indicating that the time for updating the results grows with the size of the models, the implicitly incrementalized solutions essentially ignore the size of the models when propagating updates. Furthermore, while the times of the implicitly incremental tools are all within one order of magnitude, the times for the explicitly incremental tools differ significantly and are generally much worse, again with the exception of GraphBLAS.[29]

5.3 Threats to validity

5.3.1 Internal threats to validity

Query Selection The queries used in the benchmark are artificial. They have been created by the first author in such a way that they would represent typical queries. The design goal of the second query in particular was to include some kind of graph algorithm in order to evaluate the flexibility of the tools on a concrete example. We chose the calculation of

[29] Here, we again consider the explicitly incrementalized version of the YAMTL solution as essentially implicitly incrementalized with an additional explicit tuning.

connected components because it is a well-studied algorithm and was applicable for the chosen scenario.

Technologies There is a difference of the used languages, runtimes and technologies because the solutions use different modelling approaches. Therefore, differences in response times may be due to the difference of the used framework instead of the difference in the used incremental tool. Clearing this effect from the measurements would necessitate re-implementing both the solutions and the underlying tools (in many cases large libraries or even database management systems) in another technology stack, an overhead which is not justified by this confounding effect. Further, users will likely choose a tool that actively supports the modelling approach they are using.

Tools versus solutions The paper claims to compare tools where we in fact compare solutions using these tools. However, the solutions have been created by the authors of the tools or at least experienced developers. Therefore, we think that it is realistic to assume that the solutions are optimal for the provided tool. However, this optimum may vary between solutions. A good example is the calculation of the top-3 elements from a collection with scores. Some tools realize this by taking the first elements of a sorted collection, because keeping the sorted collection speeds up the following calls and this way, the analysis is very readable. Others calculate the 3 maximal elements because the sorted collection would be discarded and is therefore not efficient.

Noise during measurements Due to repetition of measurements, we think that the influence of garbage collection and just-in-time compilation is much smaller than the observed differences between incremental and non-incremental tools. However, we ran our benchmarks on virtual machines in the cloud, where we could not control the resource isolation from other tenants.

Different input formats To account for the format of model change sequences, we reproduced the change format also in EMF and generally took the serialization, deserialization and conversion of model changes into the tools' respective native formats out of the time measurements.

5.3.2 External threats to validity

Generalizability It is unclear to what degree the obtained results can be generalized for other applications, input model characteristics and change sequences. Further, it is unclear to what extent the solutions represent the tools, even though many solutions have been implemented by tool authors. The analysis of the algorithmic complexities shows that these complexities heavily depend on the query, the characteristics of the inputs and the change sequences and all of these will be different in other applications. However, we expect that observed difference between the orders of magnitude of the update times for incremental vs. non-incremental tools is (to some degree) representative to real-world use cases which run global graph queries on continuously changing graph models.

Limited types of changes Though the change sequence used in the various case studies has been generated, they depend on the selection of changes and their proportion. In particular, we only considered incremental changes, i.e. additions of Posts, Comments or friendships. We did not consider decremental changes such as deleting Posts, Comments or breaking friendships. Those potentially require additional change propagation rules that did not have to be considered in the scope of this paper. In the scope of a benchmark, it is important to exactly specify the possible changes to the inputs because the presence of decremental changes can have an impact on the choice of algorithms to implement the change propagation [46], so the possibility of decremental changes has an impact even if they do not actually occur.

6 Related work

In this section, we review related work. We begin by reviewing studies that compare existing tools of related fields in Sect. 6.1. In Sect. 6.2, we then review existing incremental tools that we have not so far taken into account for the comparison.

6.1 Comparative studies

The present paper is an outcome of TTC 2018, but this is not the first time that the TTC has considered incremental queries. The Train Benchmark [82] by Szárnyas et al. compared query technology based on an example metamodel and queries motivated by the railway domain. A separate TTC version of this benchmark also exists that focuses more on modelling tools [83]. Both versions of this benchmark have in common that they use homogeneous change sequences. This means that solutions generally only have to react to one kind of model changes that will always affect the query result. In the benchmark presented in this paper, we use heterogeneous change sequences independent of the query, as we think that this is more realistic. Furthermore, we added a query where pure querying technology is not sufficient.

Apart from the Train Benchmark, there have been a number of other TTC contests in the recent years that took incremental change propagation into account. The TTC 2016 live contest was about a meta-transformation to transform

an abstract dataflow model into an executable model transformation, with the goal to achieve an incremental change propagation as well. However, no change sequences were provided and there was only one incremental solution.[30] Similarly, the TTC 2017 Smart Grids case [48] featured a query joining information from two models, if possible with incremental change propagation. However, only two solutions were submitted [47,72] from which only one supported incremental change propagation. The 2017 Families to Persons case [6] focused more on bidirectional change propagation but also considers incremental change propagation (in both directions). In contrast to the present social media benchmark, input and output models are isomorphic in the Families to Persons case. A complete list of the TTC cases up to 2018 can be found in [81, p.112].

A related field to incremental computation is reactive programming, where the goal is to get notifications for changes. An overview of 15 languages for reactive programming was created by Bainomugisha et al. [7]. Reactive programming approaches are built upon an important assumption, namely that signals do not change once they are processed. That is, they operate on a (potentially infinite) sequence of immutable data. This is a contrast to model analysis tools where the model usually has an approximately fixed size, but is mutable.

In the Social Media case, this difference boils down to the question whether one looks at the events that enter the system such as adding a new User, adding a friendship, etc., or whether the state of the system is looked at in its entirety. Model-driven tools make the system state explicit (in a model); meanwhile, reactive programming approaches make it rather implicit. We think that the question which of these is better highly depends on the application scenario.

6.2 Incrementality

Incrementality is a desirable property as it promises to save computational effort when analyses are computed repeatedly. Therefore, it has been a subject of research for decades [75], for example, with the search for incremental compilers [76]. Common to all of these approaches is that they take advantage of assumptions they make on the computation to perform at the cost of limited applicability.

The approach by Reps [77] for attribute grammars is among the first incrementalization systems, which specialized on a limited class of problems. This approach works by using a static dependency graph for attribute evaluations for which Reps has shown that an optimal-time re-evaluation strategy can be found by re-evaluating the attributes in a topologically sorted order of a static dependency graph. This approach rests on the assumption that the data processed by the attribute grammar are immutable. As Reps applies this technique for parse trees, this assumption is reasonable, but it does not hold for models in general. The JastAdd tool in our comparison uses this technology.

Pugh and Teitelbaum [73] then applied memoization to incremental computation. Memoization is applicable to any referential transparent function (such as getting references to the three topmost elements of Q1 or Q2 is) but rests on the assumption that the data structures it operates on are immutable—an assumption not met with models. Immutable data structures cannot represent cyclic data structures natively and therefore make it difficult to create analyses requiring them. In the Social Media benchmark, we have various cross-references between Users to represent their friends. Further, in general it is unclear which functions should be memoized to actually get a performance benefit.

Acar and others created Self-Adjusting Computation (SAC), a framework to support the development of incremental programs [2] using the then newly introduced DDGs. A good overview on SAC is provided by Acar [3]. The rough idea is to memoize the computation made for a given analysis. Closely related, Hammer and others introduced the idea of demanded computation graphs, implemented in Adapton [40,41]. Demanded computation graphs make sure that a change propagation is only performed if the result is actually needed. Both for SAC and Adapton, the modelling technology currently is a big obstacle as they are implemented in programming languages that do not have good support for models.[31] We therefore could not take these approaches into account.

A popular approach to specify queries, especially in graph transformation, is *Graph Patterns*. Bergmann et al. have created IncQuery, an incremental pattern matching system for *Graph Patterns* [10,11]. This approach uses a Rete network [32], a static dependency graph, whose nodes are primitive filter conditions or joins of partial pattern matches. Each node represents a set of (partial) pattern matches. This approach can support mutable models because the notification API of models can be used to determine when matches must be revoked or new matches arise.

In relational databases research, incrementalization manifests in the topic of incremental view maintenance [14]. An overview on the research can be found in [24,39,79]. However, we are not aware of an open-source relational management system that implements any of these techniques.

Other approaches to incremental computation include entirely new programming models that allow an easy incrementalization or parallelization. An example of these approaches is revision-based computing [21].

[30] Nevertheless, a publication of the results has been attempted but failed, mainly due to the lack of significant contributions given the low number of submissions and the difficulty to generalize results.

[31] Furthermore, for SAC there is no publicly available compiler.

7 Conclusion and future work

7.1 Conclusion

We have presented a simple benchmark for incremental graph queries and demonstrated how it can be implemented on 11 tools from different technological spaces (modelling tools, databases, graph analytical frameworks). The results allow us to reason on the questions raised in the introduction.

Does the tool fit into my technology space? A complete overview of the tools considered can be found in Table 4. Most solutions work directly with EMF models. NMF has its own modelling framework but claims compatibility of the serialized models [45]. The solutions in Differential Dataflow, GraphBLAS, Neo4j and PostgreSQL implement conversions from EMF to their native formats.

Is it useful to rely on the incrementalization of a tool or is it better [...] to implement change propagation explicitly? The analysis of asymptotic complexities in Sect. 4.6 has foreshadowed what could be confirmed by actual performance measurements on realistic graph instances in Sect. 5: in the present benchmark, several implicit incrementalization tools were able to keep up and even outperform not only solutions developed for batch execution but also solutions that have been developed explicitly to be incremental, sometimes by multiple orders of magnitude for propagating changes. Just by specifying the query in the format of those tools, developers can gain a performance improvement for incremental change propagation that seems very hard to beat otherwise and gain that essentially for free. Only the GraphBLAS solution without the performance penalty of a modelling framework was faster, but it required the developer to rephrase the problem with matrix multiplications and explicitly deal with incrementality. This is complex, error-prone and hence expensive to develop even for such simple queries. Further, it is in contrast to the implicit solutions, in particular when their front-end language matches common standards such as OCL, EOL, SQO or Java Collections queries.

The performance results demonstrate how the explicitly incrementalized solutions, which use dependency structures on a much coarser granularity, save only few intermediate results and are slower on average than the implicitly incrementalized solutions. This is because every type of change requires dedicated code to update intermediate results. The implicit tools create a much more fine-grained dependency graph that allows to reach better asymptotic complexities and better runtimes. The only exception here is YAMTL-EI, because it is essentially a tuned version of YAMTL-II and therefore uses the same granularity.

The results also show that the implicit incremental tools NMF, AOF, ATL, YAMTL-II and Differential Dataflow have similar performance characteristics (with Differential Dataflow in front due to not having the overhead of a modelling framework), so the choice which of them to use is largely a matter of the context where the problem needs to be tackled (modelling framework, programming language, etc.).

How long does it take to recover from an application crash? The implicit incrementalization tools we compared in the scope of this benchmark do not support query persistence (cf. Sect. 4.4) or currently cannot (yet?) catch up with dedicated solutions where the queries have been explicitly designed for change propagation, even when being slightly bent towards the specific problem (the incremental query version of Hawk). The explicitly incrementalized durable solutions in SQL and Neo4j are more difficult to understand but are performant, making it an interesting future research topic to generalize these kind of solutions and hide their complexity behind a high-level frontend language.

For relational databases, the lack of an out-of-the-box solution for incrementalization is particularly surprising, given the huge body of research in incremental view maintenance [24,39]. Their advantage in the scope of this benchmark would have been clear as the incremental SQL solution is much harder to understand than its batch counterpart. However, these ideas did not yet make their way into the common or open-source database management systems.

How much development effort will be necessary to implement change propagation? Of course, the development effort to implement change propagation heavily depends on the tool and how experienced the developer is with the tool. Because the solutions in this benchmark have been developed by different authors, we did not even attempt to collect data about the development effort necessary. However, the discussion in Sect. 4.3 shows that the magnitude of development effort is quite different for the tools, in particular depending on whether the incrementalization happens implicitly or requires explicit implementation. While for JastAdd and YAMTL (in the implicitly incremental solution), no changes are necessary at all and tools like NMF or AOF only require to implement extensions (which can be shared among multiple projects, so that maybe one day comprehensive libraries exist), other tools such as in particular GraphBLAS, Neo4j and SQL require a completely different approach for supporting incremental change propagation.

How does it scale? The analysis of complexities and the performance results on actual hardware gives an impression of the scalability of the tools for the benchmark queries. As discussed in Sect. 5.3, this does not mean that the same scala-

bility is reached on a different use case. Rather, the discussion in Sect. 4.6 unveils what is actually happening under the covers for a given change and this discussion could be adapted for other use cases as well. The actual performance results then show the implementation constant for the tools in comparison to each other, which likely hold also for other use cases. Thus, the results can be used to estimate the scalability also in other contexts.

Can I speed it up by adding more CPU cores? The parallelism support of the tools compared in this paper is discussed in Sect. 4.5. The results show that while the parallelism for GraphBLAS and Differential Dataflow does bring performance advantages, these are not present (yet?) for NMF or YAMTL.

Is the tool extensible [...]? When originally selecting the queries for the benchmark, we expected that the calculation of connected components could not be handled natively by any approach so that solutions had to prove the extensibility of the tools. However, although this strategy has worked for a number of tools, it has not worked for all of them as some tools (Differential Dataflow, JastAdd to some degree) could indeed handle also Q2 without an extension of the tool. For the database solutions, algorithms such as calculating connected components usually have to be developed from scratch, using relational algebra.

7.2 Future work

We have future plans to extend the Social Media benchmark, both by covering more tools and algorithms in our experiments as well as by making the benchmark more challenging. To cover more techniques, we plan to include modelling tools such as VIATRA [85], relational databases which support intra-query parallelism such as the DuckDB embeddable analytical database [74], and make use of recently developed connected components algorithms [15]. Finally, we plan to incorporate change sequences that include *deletions* [87] which is expected to make it more challenging to perform the incremental maintenance of the queries and the connected components algorithm.

Acknowledgements We would like to thank Frank McSherry for providing an initial implementation of the Differential Dataflow solution. René Schöne was supported by German Federal Ministry of Education and Research within the research project "OpenLicht". Márton Elekes's research was funded by the European Commission and the Hungarian Authorities (NKFIH) through the Arrowhead Tools project (EU Grant Agreement No. 826452, NKFIH Grant 2019-2.1.3-NEMZ ECSEL-2019-00003) and by the NRDI Fund based on the charter of bolster issued by the NRDI Office under the auspices of the Ministry for Innovation and Technology. Gábor Szárnyas was partially supported by the SQIREL-GRAPHS NWO project.

Funding Open access funding provided by Budapest University of Technology and Economics.

A Appendix

A.1 Detailed solution listings

– Listings 30–40 contain the code for the Batch and Incremental PostgreSQL solutions of Q2.

```
WITH RECURSIVE-- recursive stands here regardless of the fact
            that the 2nd subquery is the recursive one
  comment_friends (commentid, user1id, user2id) AS (
    SELECT l1.commentid, f.user1id, f.user2id
      FROM likes l1, likes l2
         , friends f
     WHERE l1.userid = f.user1id
       AND f.user2id = l2.userid
       AND l1.commentid = l2.commentid
)
, comment_friends_closed(commentid, head_userid, tail_userid)
              AS (
-- transitive closure (reachability-only, no path is recorded)
-- of friendship-subgraphs defined by comment likes

    -- start with the users that liked a specific comment.
    -- They are the nodes of the projected users graph for a
                comment
    SELECT l.commentid
         , l.userid AS head_userid, l.userid AS tail_userid
      FROM likes l
  UNION
    SELECT cfc.commentid, cfc.head_userid, f.user2id as
                tail_userid
      FROM comment_friends_closed cfc
         , comment_friends f
     WHERE cfc.tail_userid = f.user1id
       AND cfc.commentid = f.commentid
)
, comment_components AS (
    SELECT commentid, head_userid AS userid
         , min(tail_userid) AS componentid
      FROM comment_friends_closed
     GROUP BY commentid, head_userid
)
, comment_component_sizes AS (
    SELECT cc.commentid, cc.componentid, count(*) AS
                component_size
      FROM comment_components cc
     GROUP BY cc.commentid, cc.componentid
)
-- consider all comments including those without likes
SELECT c.id AS commentid
     , coalesce( sum( power(ccs.component_size, 2) ), 0) AS
                score
  FROM comments c
  left join comment_component_sizes ccs on (ccs.commentid =
              c.id)
 GROUP BY c.id, c.ts
 ORDER BY sum( power(ccs.component_size, 2) ) DESC NULLS LAST
        , c.ts DESC LIMIT 3
;
```

Listing 30 Batch PostgreSQL solution for Q2.

```
WITH RECURSIVE
  -- calculate ancestors of comments
  comments_with_rootpost_stage1(id, ancestorid) AS (
     SELECT c.id, c.parentid AS ancestorid
       FROM comments c
   UNION
     SELECT cr.id, c.parentid AS ancestorid
       FROM comments_with_rootpost_stage1 cr
          , comments c
      WHERE 1=1
         -- join
        AND cr.ancestorid = c.id
  )
  -- calculate rootpost: when ancestor's id matches a post's
             id, then it is the root post
, comments_with_rootpost AS (
    SELECT c.id, c.ancestorid AS postid
      FROM comments_with_rootpost_stage1 c
         , posts p
     WHERE 1=1
       AND c.ancestorid = p.id
)
UPDATE comments c
   SET postid = cr.postid
  FROM comments_with_rootpost cr
 WHERE c.id = cr.id
;
```

Listing 31 SQL query to initialize Comments' root Post reference for the Incremental PostgreSQL solution for Q1.

```
insert into q1_scoring (postid, postts, score)
select p.id, p.ts, 10*count(distinct c.id) +
          count(l.userid) as score
  from posts p
       left join comments c on (p.id = c.postid)
       left join likes l on (c.id = l.commentid)
 group by p.id, p.ts
;
```

Listing 32 SQL initial query for the Incremental PostgreSQL solution for Q1.

```
WITH RECURSIVE
  -- calculate ancestors of new comments
  comments_with_rootpost_stage1(id, ancestorid) AS (
     SELECT c.id, c.parentid AS ancestorid
       FROM comments_d c
   UNION
     SELECT cr.id
            -- when we reach a comment for which the root post
                       is known,
            -- we jump directly to that without traversing the
                       whole comment tree
          , coalesce(c.postid, c.parentid) AS ancestorid
       FROM comments_with_rootpost_stage1 cr
          , comments c
      WHERE 1=1
         -- join
        AND cr.ancestorid = c.id
  )
  -- calculate rootpost: when ancestor's id matches a post's
             id, then it is the root post
, comments_with_rootpost AS (
    SELECT c.id, c.ancestorid AS postid
      FROM comments_with_rootpost_stage1 c
         , posts p
     WHERE 1=1
       AND c.ancestorid = p.id
)
UPDATE comments_d c
   SET postid = cr.postid
  FROM comments_with_rootpost cr
 WHERE c.id = cr.id
;
```

Listing 33 Interim step to maintain root Post reference of all Comments' of the Incremental PostgreSQL solution for Q1.

```
WITH diff_posts AS (
    select p.id, p.ts
         , 10*count(distinct c.id) + count(l.userid) as score
    from posts_d p
        left join comments_d c on (p.id = c.postid)
        left join likes_d l on (c.id = l.commentid)
    group by p.id, p.ts
)
, diff_comments AS (
    select p.id, p.ts
         , 10*count(distinct c.id) + count(l.userid) as score
    from posts_b p
        inner join comments_d c on (p.id = c.postid)
        left join likes_d l on (c.id = l.commentid)
    group by p.id, p.ts
)
, diff_likes AS (
    select p.id, p.ts
         , count(l.userid) as score
    from posts_b p
        inner join comments_b c on (p.id = c.postid)
        inner join likes_d l on (c.id = l.commentid)
    group by p.id, p.ts
)
INSERT INTO q1_scoring AS r (postid, postts, score)
    select id, ts, sum(score) as score
    from (
        select * from diff_posts
        UNION ALL
        select * from diff_comments
        UNION ALL
        select * from diff_likes
        ) t
    group by id, ts
ON CONFLICT (postid) DO UPDATE SET score = EXCLUDED.score +
        r.score
;
```

Listing 34 Interim result maintenance step of the Incremental PostgreSQL solution for Q1.

```
select postid, score
  from q1_scoring
 order by score desc, postts desc limit
        3;
```

Listing 35 SQL result retrieval phase for the Incremental PostgreSQL solution for Q1.

```
INSERT INTO comment_friends (status,
        commentid, user1id, user2id)
    SELECT 'B' AS status
         , l1.commentid, f.user1id,
                f.user2id
      FROM likes l1, likes l2
         , friends f
     WHERE l1.userid = f.user1id
       AND f.user2id = l2.userid
       AND l1.commentid = l2.commentid;
```

Listing 36 Initialization phase for the Incremental PostgreSQL solution for Q2, initializing the *comment_friends* relation.

```
INSERT INTO comment_friends (status,
        commentid, user1id, user2id)
    SELECT 'D' AS status
         , l1.commentid, f.user1id,
                f.user2id
      FROM likes_d l1, likes l2
         , friends f
     WHERE l1.userid = f.user1id
       AND f.user2id = l2.userid
       AND l1.commentid = l2.commentid
  UNION ALL
    SELECT 'D' AS status
         , l1.commentid, f.user1id,
                f.user2id
      FROM likes_b l1, likes l2
         , friends_d f
     WHERE l1.userid = f.user1id
       AND f.user2id = l2.userid
       AND l1.commentid = l2.commentid
  UNION ALL
    SELECT 'D' AS status
         , l1.commentid, f.user1id,
                f.user2id
      FROM likes_b l1, likes_d l2
         , friends_b f
     WHERE l1.userid = f.user1id
       AND f.user2id = l2.userid
       AND l1.commentid = l2.commentid;
```

Listing 37 SQL maintenance phase for the Incremental PostgreSQL solution for Q2: updating the *comment_friends* relation.

```
WITH RECURSIVE
        comment_friends_closed_init(
    commentid, head_userid,
            tail_userid) AS (
-- transitive closure
        (reachability-only, no path is
        recorded)
-- of friendship-subgraphs defined by
        comment likes

    -- start with the users that liked
            a specific comment.
    -- They are the nodes of the
            projected users graph for a
            comment
    SELECT l.commentid, l.userid AS
            head_userid, l.userid AS
            tail_userid
      FROM likes l
  UNION
    -- expand the closure with the
            edges of the projected
            graph,
    -- which is stored in
            comment_friends table
    SELECT cfc.commentid,
            cfc.head_userid, f.user2id
            as tail_userid
      FROM comment_friends_closed_init
             cfc
         , comment_friends f
     WHERE cfc.tail_userid = f.user1id
       AND cfc.commentid = f.commentid
)
INSERT INTO
        comment_friends_closed(commentid,
        head_userid, tail_userid)
    select commentid, head_userid,
            tail_userid
      from comment_friends_closed_init w
           left join
                   comment_friends_closed
                   q using (commentid,
                   head_userid,
                   tail_userid)
     where q.commentid IS NULL;
```

Listing 38 SQL initialization phase for the Incremental PostgreSQL solution for Q2: initializing the *comment_friends* relation's closure.

```
WITH RECURSIVE -- note: though not the 1st query is
          the recursive one, the RECURSIVE keyword
          needs to be at the beginning
comment_friends_closed_stage0 AS (
    -- in order to maintain the transitive closure in
            comment_friends_closed
    -- we build on the transitive closure built so
            far and the new likes.
    -- We need the new likes because users that liked
            a specific comment
    -- are the nodes of the projected users graph for
            a comment
    SELECT commentid, head_userid, tail_userid
      FROM comment_friends_closed
  UNION ALL
    SELECT l.commentid, l.userid AS head_userid,
            l.userid AS tail_userid
      FROM likes_d l
)
, comment_friends_closed_stage1(commentid,
          head_userid, tail_userid) AS (
-- the transitive closure computed so far
          (reachability-only, no path is recorded)
-- is expanded by paths built from the new friendships
    SELECT commentid, head_userid, tail_userid
      FROM comment_friends_closed_stage0
  UNION
    SELECT cfc.commentid, cfc.head_userid, f.user2id
            as tail_userid
      FROM comment_friends_closed_stage1 cfc
         , comment_friends_d f
     WHERE cfc.tail_userid = f.user1id
       AND cfc.commentid = f.commentid
)
, comment_friends_closed_stage2 AS (
-- transitive closure having the new friendships is
          then expanded using the
-- previous transitive closure stage
      SELECT distinct cfc.commentid, cfc.head_userid,
                r.tail_userid
        FROM comment_friends_closed_stage1 cfc
             inner join comment_friends_closed r on
                       (cfc.tail_userid =
                       r.head_userid AND
                       cfc.commentid = r.commentid)
             -- LEFT JOIN and WHERE ... IS NULL is
                       the antijoin
             -- used to eliminate edges already
                       present in the previous closure
             -- this is to prevent unnecessary
                       CONFLICTs in the INSERT
                       statement below.
             left join comment_friends_closed s0 on
                       (cfc.commentid = s0.commentid
                       AND cfc.head_userid =
                       s0.head_userid AND
                       cfc.tail_userid =
                       s0.tail_userid)
      WHERE s0.commentid IS NULL
    UNION
      SELECT commentid, head_userid, tail_userid
        FROM comment_friends_closed_stage1
)
INSERT INTO comment_friends_closed(commentid,
          head_userid, tail_userid)
    select commentid, head_userid, tail_userid
      from comment_friends_closed_stage2 w
           left join comment_friends_closed q using
                     (commentid, head_userid,
                     tail_userid)
     where q.commentid IS NULL
ON CONFLICT DO NOTHING;
```

Listing 39 SQL maintenance phase for the Incremental PostgreSQL solution for Q2: updating the *comment_friends* relation's closure.

```
WITH comment_components AS (
    SELECT commentid, head_userid AS
            userid
         , min(tail_userid) AS
                componentid
      FROM comment_friends_closed
     GROUP BY commentid, head_userid
)
, comment_component_sizes AS (
    SELECT cc.commentid,
            cc.componentid, count(*) AS
            component_size
      FROM comment_components cc
     GROUP BY cc.commentid,
              cc.componentid
)
-- consider all comments including
        those without likes
SELECT c.id AS commentid
     , coalesce( sum(
              power(ccs.component_size,
              2) ), 0) AS score
  FROM comments c left join
          comment_component_sizes ccs
          on (ccs.commentid = c.id)
 GROUP BY c.id, c.ts
 ORDER BY sum(
         power(ccs.component_size, 2) )
         DESC NULLS LAST
     , c.ts DESC LIMIT 3;
```

Listing 40 SQL result retrieval phase for the Incremental PostgreSQL solution for Q2.

- Figure 14 contains the proof for the relational algebra formula for the join of relations with a positive delta (change set).
- Listings 41–42 contain the initial queries of the incremental Neo4j solution for Q2.
- Listings 43–44 contain the incremental maintenance queries of the Neo4j solution for Q2.
- Algorithm 5 shows a GraphBLAS algorithm to build the incidence matrix from an adjacency matrix.

```
MATCH (c)<-[:LIKES]-(u:User)
WITH c, collect(u) AS users
CALL apoc.create.addLabels(users,
        ['Likes_' + c.id]) YIELD node
RETURN count(*)
```

Listing 41 Neo4j incremental implementation of Q2 – initial step that materializes the subgraph by dynamic labelling of Users liking the Comment.

$$\bowtie_{i=1}^{N} (r_i \cup \Delta r_i) = \left(\bowtie_{i=1}^{N} r_i\right) \cup \bigcup_{i=1}^{N} \left(\left(\bowtie_{j=1}^{i-1} (r_j)\right) \bowtie \Delta r_i \bowtie \left(\bowtie_{j=i+1}^{N} (r_j \cup \Delta r_j)\right)\right)$$

Fig. 13 Relational algebra formula for join of relations with positive delta

$$(r_1 \cup \Delta r_1) = (r_1) \cup \bigcup_{i=1}^{1} (\Delta r_1)$$

Inductive step: show that for any $k > 1$, if $P(k-1)$ holds, then $P(k)$ also holds.

$\left(\bowtie_{i=1}^{k-1} (r_i \cup \Delta r_i)\right) \bowtie (r_k \cup \Delta r_k) =$

using the formula for $P(k-1)$

$$= \left(\left(\bowtie_{i=1}^{k-1} r_i\right) \cup \bigcup_{i=1}^{k-1} \left(\left(\bowtie_{j=1}^{i-1} (r_j)\right) \bowtie \Delta r_i \bowtie \left(\bowtie_{j=i+1}^{k-1} (r_j \cup \Delta r_j)\right)\right)\right) \bowtie (r_k \cup \Delta r_k)$$

$$= \left(\bowtie_{i=1}^{k-1} r_i\right) \bowtie (r_k \cup \Delta r_k) \cup \left(\bigcup_{i=1}^{k-1} \left(\left(\bowtie_{j=1}^{i-1} (r_j)\right) \bowtie \Delta r_i \bowtie \left(\bowtie_{j=i+1}^{k-1} (r_j \cup \Delta r_j)\right)\right)\right) \bowtie (r_k \cup \Delta r_k)$$

$$= \left(\bowtie_{i=1}^{k} r_i\right) \cup \left(\bowtie_{i=1}^{k-1} r_i\right) \bowtie \Delta r_k \cup \bigcup_{i=1}^{k-1} \left(\left(\bowtie_{j=1}^{i-1} (r_j)\right) \bowtie \Delta r_i \bowtie \left(\bowtie_{j=i+1}^{k-1} (r_j \cup \Delta r_j)\right) \bowtie (r_k \cup \Delta r_k)\right)$$

the 2nd component of the union is exactly the expression inside the big union for $i = k$*, thus*

$$= \left(\bowtie_{i=1}^{k} r_i\right) \cup \bigcup_{i=1}^{k} \left(\left(\bowtie_{j=1}^{i-1} (r_j)\right) \bowtie \Delta r_i \bowtie \left(\bowtie_{j=i+1}^{k} (r_j \cup \Delta r_j)\right)\right)$$

which is $P(k)$*. Thus the proposition holds by induction.*

Fig. 14 Proof for the relational algebra formula for join of relations with a positive delta (change set)

```
CALL apoc.periodic.commit("
MATCH (c:Comment)<-[:LIKES]-(u1:User)
WITH c, min(u1) AS u1
CREATE (c)-[:COMPONENT]->(comp:Component)
WITH c, u1, comp
CALL apoc.path.subgraphNodes(u1,
        {labelFilter: 'Likes_' + c.id,
    relationshipFilter: 'FRIEND'}) YIELD
            node AS u2
CREATE (comp)-[:USER]->(u2)
WITH c, comp, u2
MATCH (c)<-[l:LIKES]-(u2)
DELETE l
WITH c, comp, count(*) AS componentSize
SET comp.size = componentSize
RETURN count(*)
// limit - to bypass mandatory limit
")
```

Listing 42 Neo4j incremental implementation of Q2 – grouping components using fixed-point calculation and the reachability function `apoc.path.subgraphNodes` of the APOC library.

```
WITH $friendEdgeId AS friendEdgeId
MATCH (comp1:Component)-[:USER]->
   (u1:User)-[friendEdge]->(u2:User)
           <-[:USER]-(comp2:Component),
// comp1 <> comp2, because COMPONENT edges
        are different
      (comp1)<-[:COMPONENT]-(c:Comment)
       -[:COMPONENT]->(comp2)
WHERE id(friendEdge) = friendEdgeId
WITH c, comp1, comp2,
     comp1.size AS comp1Size,
     comp2.size AS comp2Size,
     comp1.size + comp2.size AS newCompSize
// mergeRels: to avoid parallel COMPONENT
        edges
CALL apoc.refactor.mergeNodes([comp1,
        comp2], {mergeRels: true}) YIELD
        node AS newComp
SET newComp.size = newCompSize,
    c.score = c.score -
            comp1Size*comp1Size -
            comp2Size*comp2Size +
            newCompSize*newCompSize
```

Listing 43 Neo4j incremental implementation of Q2 – merging components when a new FRIEND edge is inserted and the two users belonged to separate components.

```
WITH $likesEdgeId AS likesEdgeId
MATCH (c:Comment)<-[likesEdge]-(u:User)
```

```
 3 WHERE id(likesEdge) = likesEdgeId
 4 CREATE (c)-[:COMPONENT]->(uComp:Component {size:
          1})-[:USER]->(u)
 5 WITH c, uComp, u
 6 // OPTIONAL: to proceed if there is no such
          component to merge with
 7 OPTIONAL MATCH
          (c)-[:COMPONENT]->(comp2:Component)
 8   WHERE (comp2)-[:USER]->(:User)<-[:FRIEND]->(u)
 9 WITH c,
10      uComp + collect(comp2) AS components,
11      1 + sum(comp2.size) AS newCompSize,
12      sum(comp2.size * comp2.size) AS
                scoreDecrease
13 // mergeRels: to avoid parallel COMPONENT edges
14 CALL apoc.refactor.mergeNodes(components,
          {mergeRels: true})
15   YIELD node AS newComp
16 SET newComp.size = newCompSize,
17     c.score = c.score - scoreDecrease +
              newCompSize*newCompSize
```

Listing 44 Neo4j incremental implementation of Q2 – creating single-node components for the new LIKES edges, then merge them with all components where the User has friends (if any), and maintain the component sizes and the score of the Comment.

Algorithm 5 Build incidence matrix from edges.

Input
  $es = \{(i_1, j_1), (j_1, i_1), \ldots, (i_m, j_m), (j_m, i_m)\} \subseteq \{1, \ldots, n\}^2$
Output
  $\mathbf{B} \in \{0, 1\}^{n \times m}$ ▷ incidence matrix
function BUILDINCIDENCEMATRIX
  $b\_tuples \leftarrow \emptyset, k \leftarrow 1$
  for all $(i, j) \in es$ **do**
    if $i < j$ **then**
      $b\_tuples \leftarrow b\_tuples \cup \{(i, k, 1), (j, k, 1)\}$
      $k \leftarrow k + 1$
    end if
  end for
  $\mathbf{B} \leftarrow\!\!\mapsto b\_tuples$
  return B
end function

A.2 Comparison of solution variants

A.2.1 NMF

Figure 15 shows the results for the different variants of the NMF solution. The results show that the difference between the standard *incremental mode* and the *transactional mode* is marginal, i.e. the engine could not take advantage of propagating all changes at once instead of propagating each change separately. This is because the changes affect different parts of the dependency graph. However, executing these propagations in different threads also does not lead to speedups because there is usually one change propagation dominating the others. Instead, we see that the additional overhead of synchronization makes the *parallel mode's* change propagation slightly slower in this case.

A.2.2 Hawk

Figure 16 shows the execution times of all different variants of the Hawk solution. Unfortunately, it appears that the initial load process did not complete within the timeout past model size 64 across the solutions: this is due to the use of monolithic single-file models in the benchmark framework, whereas Hawk is optimized towards models which have been fragmented into many files. The results show that the initial load is somewhat slower for the *batch* and *incremental update* (IU) solutions, as they need to calculate the derived attributes used to cache scores. These derived attributes allow the initial execution of the query to be much faster in the batch and IU modes than in the *incremental update and query* (IUQ) mode: the IUQ mode has to do a first full execution of the query. The updates are the slowest in the batch solution due to the need to both recalculate derived attributes and reserialize the changed social network model between updates. The IU mode has slightly faster updates, as it skips the reserialization and instead applies the changes directly to the graph used for querying. The IUQ mode speeds this up even further by replacing derived attributes with graph listeners, using them to detect changed comments and posts and re-score them.

A.2.3 JastAdd

Figure 17 shows the execution times of all different variants of the JastAdd solution. The results show that the approaches using bidirectional relations (Relast variants) are strictly better than their counterparts, both in terms of initial execution and time to propagate updates. As can be expected, the incremental variants are also slower than the batch variants in the initial query computation. However, because the JastAdd solution needs to sort the results also in the incremental case which dominates the complexity, the performance results for the incremental variants are not significantly faster than the batch variants. With unidirectional references as in the original metamodel, the change propagation in the incremental execution is even slower than the batch variant.

A.2.4 YAMTL

Figure 18 shows the execution times of all YAMTL solutions. They indicate that the performance improvements of the YAMTL-EI solutions over the purely implicit variant are rather small, except for Q2 where the time for an update propagation could be further reduced.

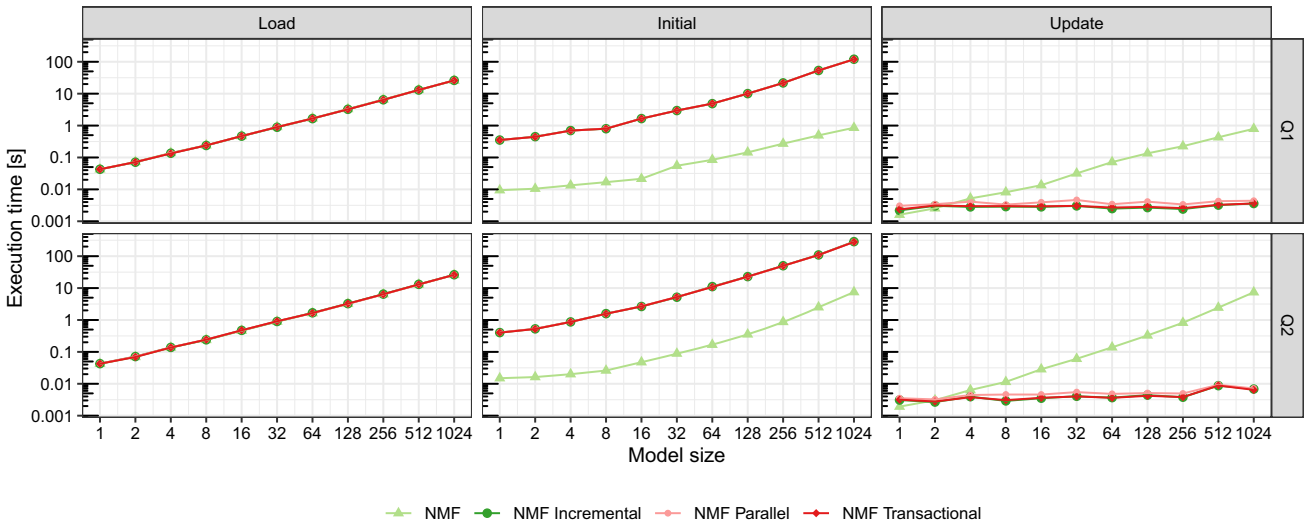


Fig. 15 Execution times of NMF solutions

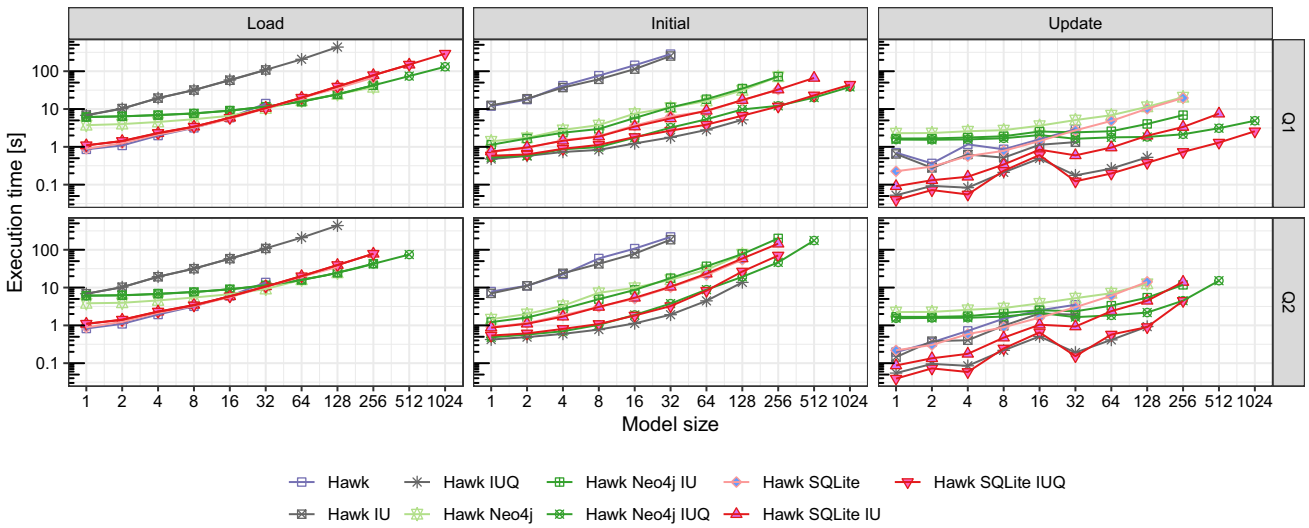


Fig. 16 Execution times of Hawk solutions. Notation: *IU:* Incremental Update, *IUQ:* Incremental Update + Query

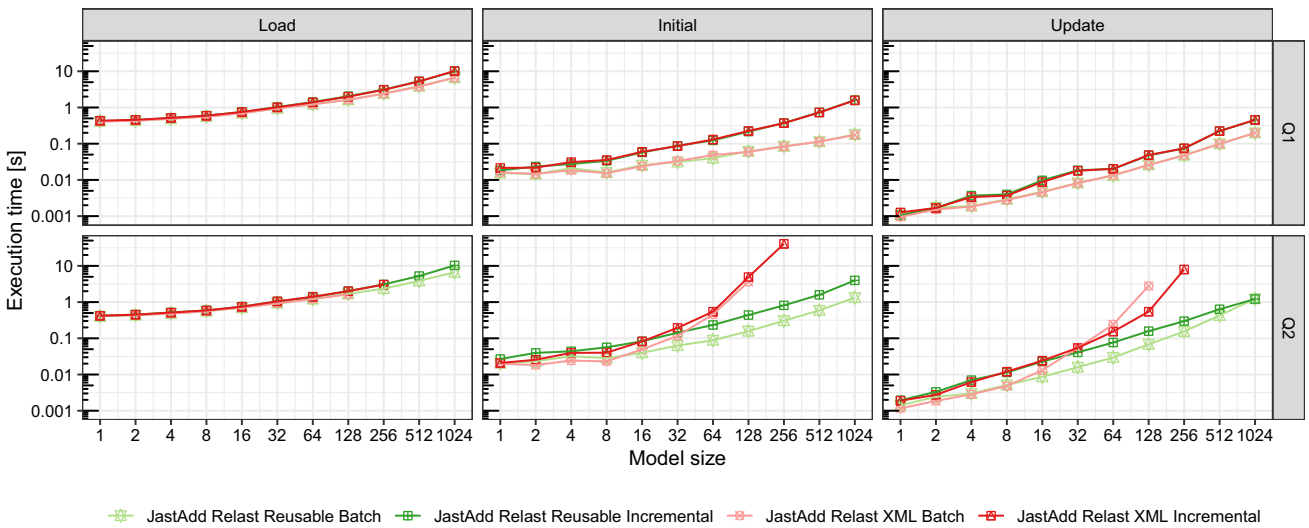


Fig. 17 Execution times of JastAdd solutions

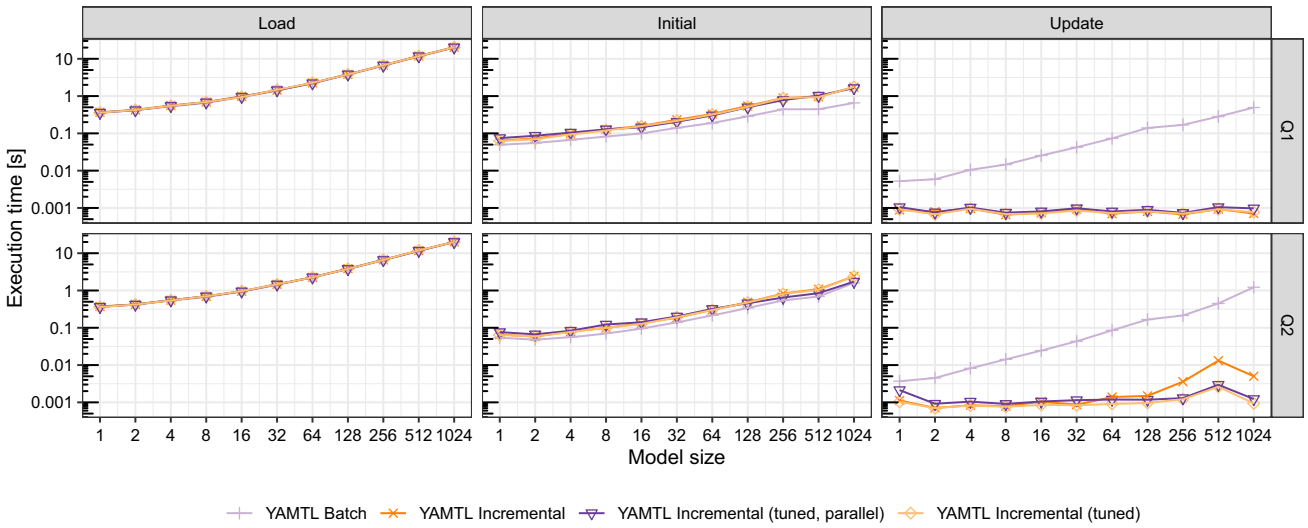


Fig. 18 Execution times of YAMTL solutions

References

1. Abadi, D.: Data partitioning. In: Liu, L., Özsu, M.T. (eds.) Encyclopedia of Database Systems, 2nd edn. Springer, Berlin (2018). https://doi.org/10.1007/978-1-4614-8265-9_688
2. Acar, U.A.: Self-adjusting computation. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, USA (2005)
3. Acar, U.A.: Self-adjusting computation (an overview). In: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, pp. 1–6. ACM (2009)
4. Aho, A.V., Ullman, J.D.: The universality of data retrieval languages. In: POPL, pp. 110–120. ACM Press (1979). https://doi.org/10.1145/567752.567763
5. Angles, R., Antal, J.B., Averbuch, A., Boncz, P.A., Erling, O., Gubichev, A., Haprian, V., Kaufmann, M., Larriba-Pey, J., Martínez-Bazan, N., Marton, J., Paradies, M., Pham, M., Prat-Pérez, A., Spasic, M., Steer, B.A., Szárnyas, G., Waudby, J.: The LDBC social network benchmark. CoRR arXiv:2001.02299 (2020)
6. Anjorin, A., Buchmann, T., Westfechtel, B., Diskin, Z., Ko, H., Eramo, R., Hinkel, G., Samimi-Dehkordi, L., Zündorf, A.: Benchmarking bidirectional transformations: theory, implementation, application, and assessment. Softw. Syst. Model. **19**(3), 647–691 (2020). https://doi.org/10.1007/s10270-019-00752-x
7. Bainomugisha, E., Carreton, A.L., Cutsem, Tv., Mostinckx, S., Meuter, Wd.: A survey on reactive programming. ACM Comput. Surv. **45**(4), 521–5234 (2013). https://doi.org/10.1145/2501654.2501666
8. Barmpis, K., García-Domínguez, A., Bagnato, A., Abherve, A.: Monitoring model analytics over large repositories with Hawk and MEASURE. In: Model Management and Analytics for Large Scale Systems, pp. 87–123. Academic Press (2020). https://doi.org/10.1016/B978-0-12-816649-9.00014-4. http://www.sciencedirect.com/science/article/pii/B9780128166499000144
9. Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.: Active operations on collections. In: Model Driven Engineering Languages and Systems: 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I. Lecture Notes in Computer Science, vol. 6394, pp. 91–105. Springer (2010)
10. Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: A benchmark evaluation of incremental pattern matching in graph transformation. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7–13, 2008. Proceedings, Lecture Notes in Computer Science, vol. 5214, pp. 396–410. Springer (2008). https://doi.org/10.1007/978-3-540-87405-8_27
11. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over EMF models. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) Model Driven Engineering Languages and Systems: 13th International Conference, MODELS 2010, Oslo, Norway, October 3–8, 2010, Proceedings, Part I, Lecture Notes in Computer Science, vol. 6394, pp. 76–90. Springer (2010). https://doi.org/10.1007/978-3-642-16145-2_6
12. Besnard, V., Jouault, F., Calvar, T.L., Tisi, M.: The TTC 2018 Social Media Case, by ATL and AOF. In: Garcia-Dominguez, A., Hinkel, G., Krikava, F. (eds.) Proceedings of the 11th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018) federation of conferences, CEUR Workshop Proceedings. CEUR-WS.org (2018)
13. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing, Birmingham (2013)
14. Blakeley, J.A., Larson, P., Tompa, F.W.: Efficiently updating materialized views. In: SIGMOD, pp. 61–71. ACM Press (1986). https://doi.org/10.1145/16894.16861
15. Bögeholz, H., Brand, M., Todor, R.: In-database connected component analysis. In: ICDE, pp. 1525–1536. IEEE (2020). https://doi.org/10.1109/ICDE48307.2020.00135
16. Boronat, A.: Expressive and efficient model transformation with an internal DSL of Xtend. In: Proceedings of the 21th ACM/IEEE International Conference on MoDELS, pp. 78–88. ACM (2018)
17. Boronat, A.: YAMTL solution to the TTC 2018 social media case. In: Garcia-Dominguez, A., Hinkel, G., Krikava, F. (eds.) Proceedings of the 11th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018) federation of conferences, CEUR Workshop Proceedings. CEUR-WS.org (2018)
18. Boronat, A.: Incremental execution of rule-based model transformation. Int. J. Softw. Tools Technol. Transf. (2020). https://doi.org/10.1007/s10009-020-00583-y
19. Brucker, A.D., Clark, T., Dania, C., Georg, G., Gogolla, M., Jouault, F., Teniente, E., Wolff, B.: Panel discussion: proposals for improving OCL. In: Proceedings of the 14th International Workshop on OCL and Textual Modelling, CEUR Workshop Proceedings, vol. 1285, pp. 83–99 (2014)

20. Buluç, A., Mattson, T., McMillan, S., Moreira, J.E., Yang, C.: Design of the GraphBLAS API for C. In: GABB at IPDPS, pp. 643–652. IEEE Computer Society (2017). https://doi.org/10.1109/IPDPSW.2017.117
21. Burckhardt, S., Leijen, D., Sadowski, C., Yi, J., Ball, T.: Two for the price of one: a model for parallel and incremental computation. SIGPLAN Not. **46**(10), 427–444 (2011). https://doi.org/10.1145/2076021.2048101
22. Calvar, T.L., Chhel, F., Jouault, F., Saubion, F.: Using process algebra to statically analyze incremental propagation graphs. In: Hebig, R., Berger, T. (eds.) Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018, CEUR Workshop Proceedings, vol. 2245, pp. 160–173. CEUR-WS.org (2018)
23. Calvar, T.L., Jouault, F., Chhel, F., Clavreul, M.: Efficient ATL incremental transformations. J. Object Technol. **18**(3), 2:1-17 (2019). https://doi.org/10.5381/jot.2019.18.3.a2
24. Chirkova, R., Yang, J.: Materialized views. Found. Trends Databases **4**(4), 295–405 (2012). https://doi.org/10.1561/1900000020
25. Davis, T.A.: Algorithm 1000: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra. ACM Trans. Math. Softw. **45**(4), 44:1-44:25 (2019). https://doi.org/10.1145/3322125
26. Dong, G., Su, J.: Incremental maintenance of recursive views using relational calculus/SQL. SIGMOD Rec. **29**(1), 44–51 (2000). https://doi.org/10.1145/344788.344808
27. Elekes, M., Antal, J.B., Szárnyas, G.: An analysis of the SIGMOD 2014 programming contest: complex queries on the LDBC social network graph. CoRR arXiv:2010.12243 (2020)
28. Elekes, M., Szárnyas, G.: An incremental GraphBLAS solution for the 2018 TTC Social Media case study. In: GrAPL at IPDPS, pp. 203–206. IEEE (2020). https://doi.org/10.1109/IPDPSW50202.2020.00045
29. Elekes, M., Szárnyas, G.: Incremental view maintenance in graph databases: a case study in Neo4j. In: Proceedings of the 27th PhD mini-symposium. Budapest University of Technology and Economics, Department of Measurement and Information Systems (2020). http://docs.inf.mit.bme.hu/paper-minisy20-elekes/elekes.pdf
30. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat-Pérez, A., Pham, M., Boncz, P.A.: The LDBC social network benchmark: interactive workload. In: SIGMOD, pp. 619–630 (2015). https://doi.org/10.1145/2723372.2742786
31. Fan, W., Hu, C., Tian, C.: Incremental graph computations: doable and undoable. In: SIGMOD, pp. 155–169. ACM (2017). https://doi.org/10.1145/3035918.3035944
32. Forgy, C.L.: Rete: a fast algorithm for the many pattern/many object pattern match problem. Artif. Intell. **19**(1), 17–37 (1982)
33. Francis, N., et al.: Cypher: an evolving query language for property graphs. In: SIGMOD, pp. 1433–1445. ACM (2018). https://doi.org/10.1145/3183713.3190657
34. Garcia-Dominguez, A.: Hawk solutions to the TTC 2018 Social Media Case. In: Garcia-Dominguez, A., Hinkel, G., Krikava, F. (eds.) Proceedings of the 11th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018) Federation of Conferences, CEUR Workshop Proceedings. CEUR-WS.org (2018)
35. García-Domínguez, A., Hinkel, G., Krikava, F. (eds.): Proceedings of the 11th Transformation Tool Contest, Co-located with the 2018 Software Technologies: Applications and Foundations, TTC@STAF 2018, Toulouse, France, June 29, 2018, CEUR Workshop Proceedings, vol. 2310. CEUR-WS.org (2019). http://ceur-ws.org/Vol-2310
36. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Softw. Syst. Model. **8**(1), 21–43 (2009). https://doi.org/10.1007/s10270-008-0089-9
37. Green, A., et al.: Updating graph databases with Cypher. PVLDB (2019). https://doi.org/10.14778/3352063.3352139. http://www.vldb.org/pvldb/vol12/p2242-green.pdf
38. Griffin, T., Kumar, B.: Algebraic change propagation for semijoin and outerjoin queries. SIGMOD Rec. **27**(3), 22–27 (1998). https://doi.org/10.1145/290593.290597
39. Gupta, A., Mumick, I.S., et al.: Maintenance of materialized views: problems, techniques, and applications. IEEE Data Eng. Bull. **18**(2), 3–18 (1995)
40. Hammer, M.A., Dunfield, J., Headley, K., Labich, N., Foster, J.S., Hicks, M., Van Horn, D.: Incremental computation with names. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 748–766. ACM (2015)
41. Hammer, M.A., Phang, K.Y., Hicks, M., Foster, J.S.: Adapton: composable, demand-driven incremental computation. SIGPLAN Not. **49**(6), 156–166 (2014). https://doi.org/10.1145/2666356.2594324
42. Hartmann, T., Fouquet, F., Jimenez, M., Rouvoy, R., Traon, Y.L.: Analyzing complex data in motion at scale with temporal graphs, pp. 596–601 (2017). https://doi.org/10.18293/SEKE2017-048. http://ksiresearchorg.ipage.com/seke/seke17paper/seke17paper_48.pdf
43. Hedin, G.: Reference attributed grammars. Informatica **24**(3), 301 (2000)
44. Hedin, G., Magnusson, E.: JastAdd: an aspect-oriented compiler construction system. Sci. Comput. Program. **47**, 37–58 (2003)
45. Hinkel, G.: NMF: A Modeling Framework for the .NET Platform. Technical report, Karlsruhe Institute of Technology, Karlsruhe (2016). http://nbn-resolving.org/urn:nbn:de:swb:90-537082
46. Hinkel, G.: Implicit incremental model analyses and transformations. Ph.D. thesis, Karlsruhe Institute of Technology (2017)
47. Hinkel, G.: An NMF solution to the Smart Grid case at the TTC 2017. In: García-Domínguez, A., Hinkel, G., Krikava, F. (eds.) Proceedings of the 10th Transformation Tool Contest (TTC 2017), Co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017, CEUR Workshop Proceedings, vol. 2026, pp. 13–17. CEUR-WS.org (2017). http://ceur-ws.org/Vol-2026/paper5.pdf
48. Hinkel, G.: The TTC 2017 outage system case for incremental model views. In: Garcia-Dominguez, A., Hinkel, G., Krikava, F. (eds.) Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) Federation of Conferences, CEUR Workshop Proceedings. CEUR-WS.org (2017)
49. Hinkel, G.: An NMF solution to the TTC 2018 Social Media Case. In: Garcia-Dominguez, A., Hinkel, G., Krikava, F. (eds.) Proceedings of the 11th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018) Federation of Conferences, CEUR Workshop Proceedings. CEUR-WS.org (2018)
50. Hinkel, G.: NMF: a multi-platform modeling framework. In: Rensink, A., Cuadrado, J.S. (eds.) Theory and Practice of Model Transformations: 11th International Conference, ICMT 2018, Held as Part of STAF 2018, Toulouse, France, June 25–29, 2018. Proceedings. Springer (2018)
51. Hinkel, G.: The TTC 2018 social media case. In: Garcia-Dominguez, A., Hinkel, G., Krikava, F. (eds.) Proceedings of the 11th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018) Federation

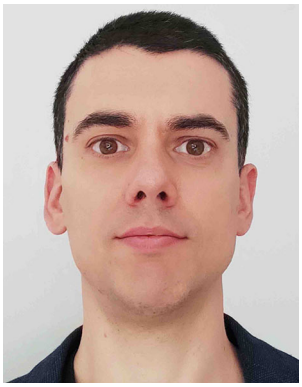
of Conferences, CEUR Workshop Proceedings. CEUR-WS.org (2018)
52. Hinkel, G., Happe, L.: An NMF solution to the TTC train benchmark case. In: Rose, L., Horn, T., Krikava, F. (eds.) Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) Federation of Conferences, CEUR Workshop Proceedings, vol. 1524, pp. 142–146. CEUR-WS.org (2015)
53. Jouault, F., Beaudoux, O.: On the use of active operations for incremental bidirectional evaluation of OCL. In: Proceedings of the 15th International Workshop on OCL and Textual Modeling, CEUR Workshop Proceedings, vol. 1512, pp. 35–45. Ottawa, Canada (2015)
54. Jouault, F., Beaudoux, O.: Efficient OCL-based incremental transformations. In: Proceedings of the 16th International Workshop in OCL and Textual Modeling, CEUR Workshop Proceedings, vol. 1756, pp. 121–136. Saint-Malo, France (2016)
55. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, vol. Satellite, pp. 128–138. Springer (2005)
56. Jouault, F., Tisi, M.: Towards incremental execution of atl transformations. In: Tratt, L., Gogolla, M. (eds.) Theory and Practice of Model Transformations, pp. 123–137. Springer, Berlin (2010)
57. Kepner, J., Aaltonen, P., Bader, D.A., Buluç, A., Franchetti, F., Gilbert, J.R., Hutchison, D., Kumar, M., Lumsdaine, A., Meyerhenke, H., McMillan, S., Yang, C., Owens, J.D., Zalewski, M., Mattson, T.G., Moreira, J.E.: Mathematical foundations of the GraphBLAS. In: HPEC, pp. 1–9 (2016). https://doi.org/10.1109/HPEC.2016.7761646
58. Kepner, J., Gilbert, J.R. (eds.): Graph Algorithms in the Language of Linear Algebra, Software, Environments, Tools, vol. 22. SIAM (2011). https://doi.org/10.1137/1.9780898719918
59. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press, San Francisco (2018)
60. Knuth, D.E.: Semantics of context-free languages. Math. Syst. Theory **2**(2), 127–145 (1968). https://doi.org/10.1007/BF01692511
61. Kolovos, D.S., Paige, R.F., Polack, F.: 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, pp. 128–142 (2006). https://doi.org/10.1007/11787044_11
62. Magnusson, E., Hedin, G.: Circular reference attributed grammars: their evaluation and applications. Sci. Comput. Program. **68**(1), 21–37 (2007)
63. Martínez, S., Tisi, M., Douence, R.: Reactive model transformation with atl. Sci. Compute. Program. **136**, 1–16 (2017). https://doi.org/10.1016/j.scico.2016.08.006
64. Mattson, T., Davis, T.A., Kumar, M., Buluç, A., McMillan, S., Moreira, J.E., Yang, C.: LAGraph: a community effort to collect graph algorithms built on top of the GraphBLAS. In: GrAPL at IPDPS, pp. 276–284 (2019). https://doi.org/10.1109/IPDPSW.2019.00053
65. McSherry, F., Murray, D.G., Isaacs, R., Isard, M.: Differential dataflow. In: CIDR (2013). http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf
66. Mey, J., Kühn, T., Schöne, R., Aßmann, U.: Reusing static analysis across different domain-specific languages using reference attribute grammars. Art Sci. Eng. Program. (2020). https://doi.org/10.22152/programming-journal.org/2020/4/15
67. Mey, J., Schöne, R., Hedin, G., Söderberg, E., Kühn, T., Fors, N., Öqvist, J., Aßman, U.: Continuous model validation using reference attribute grammars. In: Proceedings of the 11th International Conference on Software Language Engineering (2018)
68. Monge, A.E., Elkan, C.: An efficient domain-independent algorithm for detecting approximately duplicate database records. In: DMKD (1997)
69. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: SIGOPS, pp. 439–455. ACM (2013). https://doi.org/10.1145/2517349.2522738
70. Murray, D.G., McSherry, F., Isard, M., Isaacs, R., Barham, P., Abadi, M.: Incremental, iterative data processing with timely dataflow. Commun. ACM **59**(10), 75–83 (2016). https://doi.org/10.1145/2983551
71. Needham, M., Hodler, A.E.: Graph Algorithms: Practical Examples in Apache Spark and Neo4j. O'Reilly Media, Sebastopol (2019)
72. Peldszus, S., Bürger, J., Strüber, D.: Detecting and preventing power outages in a smart grid using emoflon. In: García-Domínguez, A., Hinkel, G., Krikava, F. (eds.) Proceedings of the 10th Transformation Tool Contest (TTC 2017), Co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017, CEUR Workshop Proceedings, vol. 2026, pp. 19–23. CEUR-WS.org (2017). http://ceur-ws.org/Vol-2026/paper17.pdf
73. Pugh, W., Teitelbaum, T.: Incremental computation via function caching. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 315–328. ACM (1989)
74. Raasveldt, M., Mühleisen, H.: DuckDB: an embeddable analytical database. In: SIGMOD, pp. 1981–1984. ACM (2019). https://doi.org/10.1145/3299869.3320212
75. Ramalingam, G., Reps, T.: A categorized bibliography on incremental computation. In: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 502–510. ACM (1993)
76. Reiss, S.P.: An approach to incremental compilation. In: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, SIGPLAN '84, pp. 144–156. ACM, New York, NY, USA (1984). https://doi.org/10.1145/502874.502889
77. Reps, T.: Optimal-time incremental semantic analysis for syntax-directed editors. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '82, pp. 169–176. ACM, New York, NY, USA (1982). https://doi.org/10.1145/582153.582172
78. Schöne, R., Mey, J.: A JastAdd-based solution to the TTC 2018 social media case. In: Garcia-Dominguez, A., Hinkel, G., Krikava, F. (eds.) Proceedings of the 11th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018) federation of conferences, CEUR Workshop Proceedings. CEUR-WS.org (2018)
79. Sebaa, A., Tari, A.: Materialized view maintenance: issues, classification, and open challenges. Int. J. Coop. Inf. Syst. **28**(1), 19300011–193000159 (2019). https://doi.org/10.1142/S0218843019300018
80. Sedgewick, R., Wayne, K.: Algorithms, 4th edn. Addison-Wesley, Boston (2011)
81. Szárnyas, G.: Query, analysis, and benchmarking techniques for evolving property graphs of software systems. Ph.D. dissertation, Budapest University of Technology and Economics (2019). http://hdl.handle.net/10890/13133
82. Szárnyas, G., Izsó, B., Ráth, I., Varró, D.: The train benchmark: cross-technology performance evaluation of continuous model queries. Softw. Syst. Model. (2017). https://doi.org/10.1007/s10270-016-0571-8
83. Szárnyas, G., Semeráth, O., Ráth, I., Varró, D.: The TTC 2015 train benchmark case for incremental model validation. In: TTC at STAF, pp. 129–141 (2015). http://ceur-ws.org/Vol-1524/paper2.pdf
84. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
85. Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z.: Road to a reactive and incremental model transfor-

mation platform: three generations of the VIATRA framework. Softw. Syst. Model. **15**(3), 609–629 (2016). https://doi.org/10.1007/s10270-016-0530-4
86. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher order attribute grammars. In: PLDI '89. ACM, New York, NY, USA (1989). https://doi.org/10.1145/73141.74830
87. Waudby, J., Steer, B.A., Prat-Pérez, A., Szárnyas, G.: Supporting dynamic graphs and temporal entity deletions in the LDBC Social Network Benchmark's data generator. In: GRADES-NDA at SIGMOD, pp. 8:1–8:8. ACM (2020). https://doi.org/10.1145/3398682.3399165
88. Zhang, Y., Azad, A., Hu, Z.: FastSV: a distributed-memory connected component algorithm with fast convergence. In: PPSC, pp. 46–57. SIAM (2020). https://doi.org/10.1137/1.9781611976137.5
89. Zhao, K., Yu, J.X.: All-in-one: graph processing in RDBMSs revisited. In: SIGMOD, pp. 1165–1180. ACM (2017). https://doi.org/10.1145/3035918.3035943

Georg Hinkel received his B.Sc. and M.Sc. degrees in Computer Science from the Karlsruhe Institute of Technology (KIT), in 2011 and 2014, respectively, and the B.Sc. degree in math in 2012. In 2017, he received his Ph.D. degree on implicit incremental model analyses and transformations from the KIT. Currently, he is a Senior Software Technology Engineer at Tecan Software Competence Center GmbH. His research interest covers model-driven engineering, incrementality and medical robotics. He has organized several international workshops and is a reviewer for multiple international journals. He is the lead developer of NMF and has (co)-authored more than 30 peer-reviewed publications.

Dr. Antonio Garcia-Dominguez is a Lecturer in Computer Science at Aston University (United Kingdom). His main research interests are model-driven engineering and software testing, with an interest on the use of non-relational database technologies and AI approaches to deal with increasingly more complex systems. In addition to over 10 papers in peer-reviewed journals and over 40 papers in conferences and workshops, Antonio is a core contributor in several related open source projects. Some of these projects include the Eclipse Epsilon model management languages and tools, the MuBPEL mutation testing framework for WS-BPEL, or the Eclipse Hawk model indexing framework.

René Schöne is a research assistant and Ph.D. student at the Chair of Software Technology at Technische Universität Dresden. His research and PhD focuses on the application of reference attribute grammars for models@run.time in self-adaptive systems currently within the domain of smart home. Challenges there include adequate modelling of domains, abstraction of and connection to real hardware devices, and efficient analyses in the presence of frequent model updates.



Artur Boronat is a lecturer at the School of Computing and Mathematical Sciences of the University of Leicester (UK). His research interests revolve around: model-driven engineering and agile software development, applications in healthcare and industry 4.0; application of AI technology and formal methods in the intersection of the two areas above. He obtained his Ph.D. degree from the Universitat Politénica de Valéncia (UPV, Spain) in 2007. He has been a visiting researcher at University of Illinois at Urbana-Champaign (UIUC, USA) and at Universitat Politécnica de Catalunya (UPC, Spain). He has organized several international workshops and is a reviewer for multiple international journals and funding bodies. He has (co)-authored more than 50 peer-reviewed publications and authored several MDE tools, such as YAMTL for model transformations and EMF-Syncer for agile model-driven engineering.

Massimo Tisi is an associate professor in the Department of Computer Science of the Institut Mines-Telecom Atlantique (IMT Atlantique, Nantes, France), and deputy leader of the NaoMod team, LS2N (UMR CNRS 6004). Since 2019 he coordinates the Lowcomote Marie Curie European Training Network. He has been visiting researcher at McGill University and the National Institute of Informatics (NII) in Japan, and post-doctoral fellow at Inria. He received his PhD degree in Information Engineering at Politecnico di Milano (Italy), where he was a member of the Database and Web Technologies group. His research interests revolve around software and system modeling, domain-specific languages and applied logic. He contributes to the design of the ATL model-transformation language and investigates the application of deductive verification techniques to model-driven engineering.

Théo Le Calvar is an associate professor at IMT-Atlantique (France). He received his PhD in Computer Sciences from the University of Angers in France in 2019. He worked as a postdoc in the ERIS team at ESEO in Angers, France and in the GEODES team at Université de Montréal in Canada. His current research interests include incremental model transformation, constraint solving and modeling on the web.

Frederic Jouault is a research associate at ESEO, France. He received his PhD from the University of Nantes before doing a postdoc at the University of Alabama at Birmingham. His research interests involve model engineering, transformation, synchronization, and execution, as well as their application to Domain-Specific Languages (DSLs) and model-based reverse engineering. Frédéric created ATL, a DSL for model-to-model transformation. He is now leading the development of ATL (language and toolkit) on Eclipse.org, and is in charge of the Eclipse modeling MMT project as well as a member of the modeling PMC.

József Marton is a lecturer in Database Theory at the Faculty of Electrical Engineering and Informatics of the Budapest University of Technology and Economics (Hungary). His research interest covers continuous query processing, continuous data integration from heterogeneous data sources and mapping non-relational data models and queries to relational model and SQL engines. Jozsef also contributes to related open source research and industrial projects.

Tamás Nyíri received his M.Sc. degree in computer science from the Budapest University of Technology and Economics in 2021. In his thesis he investigated incremental execution technologies for graph queries, especially the differential dataflow computational model. Currently, he is working at Cloudera as a software engineer.

János Benjamin Antal is a software engineer. He received his M.Sc. degree in computer science from the Budapest University of Technology and Economics in 2019. As part of his master thesis he contributed to the Linked Data Benchmark Council Social Network Benchmark. Currently, he is working on an in-memory graph database at Memgraph.

Márton Elekes is a Ph.D. student at the Budapest University of Technology and Economics. His research interests include graph databases and software testing. He is a contributor of the LDBC Social Network Benchmark.

Gábor Szárnyas is a postdoctoral researcher. He obtained his Ph.D. in software engineering in 2019, focusing on the intersection of object-oriented graph models and property graphs. He currently works on efficient graph processing techniques, including formulating graph algorithms in the language of linear algebra (GraphBLAS, LAGraph), implementing property graph query engines (openCypher, SQL/PGQ), and designing graph benchmarks. He serves on the steering committee of the Linked Data Benchmark Council.