Incremental view maintenance in graph databases: A case study in Neo4j

Márton Elekes, Gábor Szárnyas
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, Hungary
Email: {elekes, szarnyas}@mit.bme.hu

***Abstract*—With the increasing amount of densely connected data sets, graph analysis has become an integral part of data processing pipelines. Therefore, the last decade saw the emergence of numerous dedicated graph analytical systems along with specialized graph database management systems. Traditionally, graph analytical tools targeted global fixed-point computations, while graph databases focused on simpler transactional read operations such as retrieving the neighbours of a node. However, recent applications of graph processing (such as financial fraud detection and serving personalized recommendations) often necessitate a mix of the two workload profiles. Following this trend, the 2018 Transformation Tool Contest (an annual competition for graph transformation tools) presented a case study that requires participants to compute complex graph queries defined on a continuously changing social network graph. The solutions are assessed based on their scalability and query reevaluation time, therefore, solutions are encouraged to incrementalize their implementations. This paper demonstrates a solution in the popular Neo4j graph database using several incrementalization techniques and compares them against the reference implementation of the case study.**

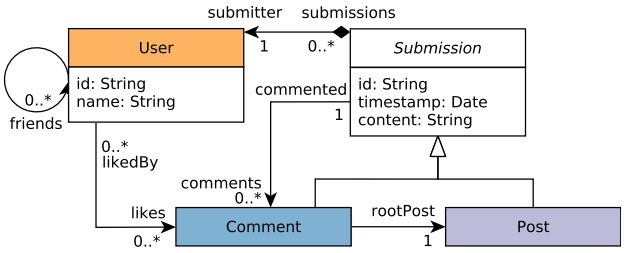


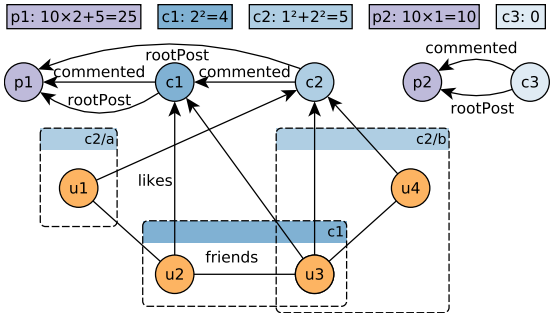
Fig. 1: Graph schema of the case study.

I. Introduction

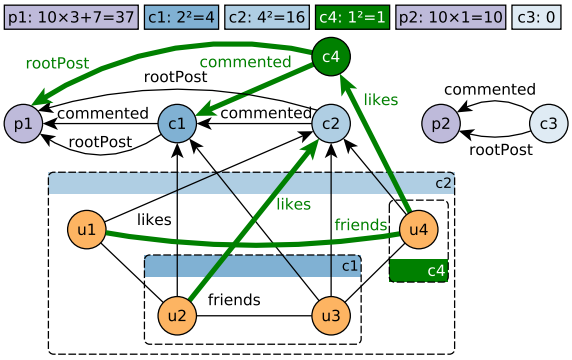
We start by describing the "Social Media" case study of the 2018 Transformation Tool Contest [5]. This case study is defined using a familiar social network-like data model (Fig. 1) consisting of Users and their Submissions. These submissions form a tree where the root node is a Post and the rest of the nodes are Comments. Users can like Comments and form friends relations with each other. Additionally, Comments have a direct pointer rootPost to the root Post to allow quick lookups. Fig. 2a shows an example graph with two Posts (`p1`, `p2`), three Comments (`c1`, `c2`, `c3`) and four Users (`u1`, ..., `u4`). Solutions are required to compute two queries:

Q1: influential posts. Assign a score to each Post, defined as 10 times the number of their (direct or indirect) Comments plus the number of Users liking those Comments. Return the top 3 Posts according to their score.

Q2: influential comments. Assign a score to each Comment, based on the friendships of the Users who like that Comment. Based on the graph formed by the User nodes and their friends edges, for every comment we define an induced subgraph which contains the Users who like the Comment and their friends edges. The subgraph contains connected components,



(a) Initial graph 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. $1^2 + 2^2 = 5$.



(b) Graph after performing an update that inserted six 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 rootPost edge to Post `p1`, (5) an outgoing commented edge to Comment `c1`, and (6) an incoming likes edge from User `u4`. The changes have increased the score of Post `p1` and resulted in Comment `c2` having a single component of size 4, therefore receiving a score of $4^2 = 16$. Comment `c4` getting a score of $1^2 = 1$.

Fig. 2: Example graphs: initial and updated versions.

		EnumPaths (very slow)	EnumPaths Subgraph △	Reachability Subgraph (incompatible)	**MatComp FixedPoint** □	MatComp PathOp ⧄	**GraphAlgo** ⊠	GraphAlgo EdgeFilter ×	MatComp GraphAlgo ⊠
initial evaluation	enumerate paths	●	●			●			
	materialize subgraph (edges/labels)		●	↯	●	●			
	reachability (shortestPath)			↯					
	reachability+dynamic labels (APOC lib.)				●				
	fixed-point calculation (APOC library)				●				
	materialize subgraph components				●	●			●
	Graph Algorithms library						●	●	●
update	reevaluation						●	●	
	maintain component	n/a		n/a	●	●			●
	dirty flag for Comments		●						

TABLE I: Comparison of strategies for Q2 regarding the techniques (language features and libraries) used in each strategy. Strategies in **bold** are described in more detail. Notation – ●: the technique is used by the strategy, ↯: incompatible strategies. "MatComp" strategies are denoted with □, while "GraphAlgo" strategies are denoted with ×.

i.e. groups of users who know each other directly or via friends. The score is defined as the sum of squared component sizes. Finally, the top 3 Comments should be returned.

Updating the graph. After the query execution on the initial graph the case study requires solutions to perform a number of updates on the graph while maintaining the results of the queries. Fig. 2 shows the initial graph and the updated graph with the scores for Q1 and Q2.

Neo4j. 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 [2] which offers both read and update constructs [3]. While the main focus of Neo4j is to run graph queries, it also supports graph analytical algorithms with the recently released Graph Algorithms library [7].

II. Approach

Q1 Batch. Q1 can be expressed with the Cypher query in Listing 1. The Cypher language uses node labels (e.g. Post, Comment, User), edge types (e.g. ROOT_POST, LIKES), node and edges properties to express graph patterns. The query matches every node with label Post, then all its comments via the rootPost edges, then the Users via the likes edges. `OPTIONAL MATCH` denotes an optional pattern, where variables are filled with NULL values if there is no match. `RETURN` can be also used to group and aggregate. The results are grouped by the id and timestamp properties of the Posts, aggregated, then the top 3 scores are returned. The aggregation counts the likes using the number of Users (a User can like more Comments), and counts the number of Comments (`DISTINCT` is used to remove duplicate Comments).

```
MATCH (p:Post)
OPTIONAL MATCH (p)<-[:ROOT_POST]-(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 1: Q1 Batch.

Q1 Incremental. To incrementally evaluate Q1, we initially compute the score for each Post as previously and store it in score property (Listing 2). Based on this property the current top 3 scores can be computed using Listing 3. The score property is indexed to improve lookup times.

For every batch of updates Alg. 1 is executed to insert new elements and update the score property, then Listing 3 is used to get the top 3.

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

Listing 2: Q1 Incremental – initial evaluation.

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

Listing 3: Q1 Incremental – get top 3 results.

Algorithm 1 Maintaining scores for Q1

```
1:  procedure UpdateQ1(updates)
2:      for all update ∈ updates do
3:          AddNewElement(update)                    ▷ insert into the graph
4:          if update is Post then
5:              update.score ← 0                     ▷ init score for new Post
6:          else if update is ⟨Comment, Post⟩ then   ▷ new Comment node
7:              ⟨_, rp⟩ ← update                     ▷ get the root Post
8:              rp.score ← rp.score + 10             ▷ update score
9:          else if update is ⟨User, Comment⟩ then   ▷ new likes edge
10:             ⟨_, c⟩ ← update                      ▷ get Comment vertex of new edge
11:             rp ← RootPost(c)                     ▷ navigate via rootPost
12:             rp.score ← rp.score + 1              ▷ update score
13:         end if
14:     end for
15: end procedure
```

Q2 Strategies. Table I compares the different strategies used for Q2 to find connected components in a subgraph and handle updates. Fig. 3 shows a comparison of their runtime during the initial evaluation and after the updates. The *EnumPaths* strategy finds the connected components by enumerating all paths of friends edges between Users who like a comment

and filtering to ensure that every User on the path also likes the Comment. The complexity of this operation is intractable, therefore this strategy is not feasible on larger graphs. To only enumerate valid paths in the subgraph, the edges of the subgraph can be materialized for each subgraph, i.e. for each Comment (stored as an edge property). The combination of these techniques (*EnumPaths Subgraph* △) also exhibits poor scalability (later measurements show that it reaches the time limit after graph size 8). This limitation could be resolved by using a Cypher reachability query, but unfortunately such queries cannot perform filtering on properties (which is necessary to ensure we only enumerate valid paths).

To use reachability queries and avoid the costly enumeration of paths, *MatComp FixedPoint* □ strategy uses Neo4j's APOC library[1]. It materializes subgraph components by converting $\text{User} \xrightarrow{\text{likes}} \text{Comment}$ edges to $\text{Comment} \xrightarrow{\text{component}} \text{Component}$ and $\text{Component} \xrightarrow{\text{user}} \text{User}$ edges where the Component node connects every User who knows each other directly or via friends. The conversion is executed for each component one by one using the fixed-point query execution mechanism of APOC. This strategy has the best performance, which is caused by the use of reachability function and incremental maintenance after updates. Component materialization can be expressed with pure Cypher queries using path operations (*MatComp PathOp* ⧋, which also has limited scalability due to the use of path enumeration.

GraphAlgo strategies (×) use functions provided by the Neo4j Graph Algorithms library[2] [7] to find connected components in a subgraph conveniently. The library loads the subgraph into an in-memory projected subgraph before running the computations. The initial runtimes of these solutions are worse than the runtime of *MatComp FixedPoint*. This can be attributed to the load phase, which is run for every subgraph, i.e. for every Comment.

Strategies can differ in the way they handle the updates: they can fully reevaluate the queries or incrementally maintain the results depending on the update. Repeated reevaluations take a significant amount of time, which causes the execution to time out. *EnumPaths Subgraph* △ strategy uses Comment-level incrementalization with dirty flags, but using path enumerations limits its scalability. The incremental evaluation of materialized components (□) is implemented by merging the components and maintaining their sizes and the scores. These updates have the best performance. (The *MatComp* strategies with worse initial runtime reaches the time limit.)

Q2 Batch. Listing 4 shows the *GraphAlgo* solution for Q2 using Neo4j Graph Algorithms library. The `algo.unionFind.stream` function is used to find connected components of the subgraph given by the node labels and edge types or Cypher queries. For each Comment, the first Cypher query in Lines 3–7 select Users who like the Comment, the second query selects all friend edges as pairs

[1]https://neo4j.com/labs/apoc/
[2]https://neo4j.com/docs/graph-algorithms/

```
1 MATCH (c:Comment)
2   CALL algo.unionFind.stream(
3     'MATCH (c:Comment)<-[:LIKES]-(u:User)
4     WHERE id(c)=' + id(c) + '
5     RETURN id(u) as id',
6     'MATCH (u1:User)-[:FRIEND]->(u2:User)
7     RETURN id(u1) as source, id(u2) as target',
8     {graph: 'cypher'})
9   YIELD setId
10  WITH c, setId, count(setId) AS cSize
11  WITH c, cSize * cSize AS cSize_2
12  RETURN c.id AS id, sum(cSize_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 4: Q2 batch using the Neo4j Graph Algorithms library.

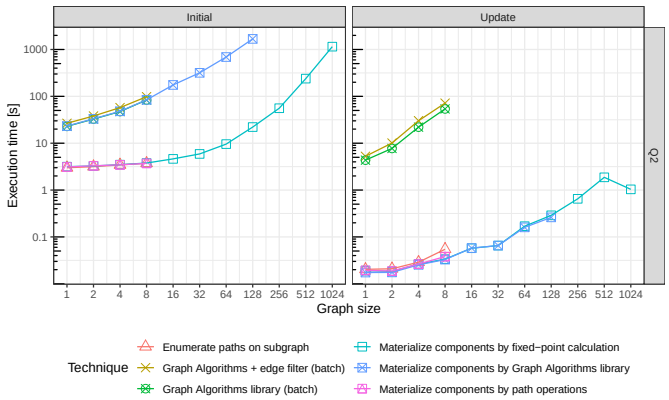


Fig. 3: Performance comparison of Q2 strategies. The symbols denoting techniques correspond to those shown in Table I.

of Users. The function returns the ID of the component containing the User node. Lines 10–14 calculate the squared sum of the component sizes and selects the top 3 scores. The function is invoked only for Comments with likes. The top 3 Comments without likes are enumerated by Lines 16–20.

Q2 Incremental. The incremental solution in this section (*MatComp FixedPoint*) materializes the components of the subgraph using fixed-point query execution of APOC library. To achieve this, the solution materializes subgraph edges as dynamically named labels (Listing 5) and finds reachable nodes using the APOC library (Listing 6). The incremental evaluation is performed by merging the components and maintaining their sizes and the scores (Listing 7).

III. Evaluation

To evaluate the performance and scalability of our solution, we have used the benchmark framework of the case study [5]. This executes the queries on graphs of increasing sizes as shown in Table II, then adds new elements to the graph, and maintains the result using the queries. (Similar queries can be formulated for element removal.) As a performance baseline,

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

Listing 5: Q2 – materializing subgraph by dynamic labelling of users liking the comment.

```
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(*)")
```

Listing 6: Q2 – grouping components by selecting a single user (u1) per comment and their reachable friends (u2) in the subgraph, then replacing LIKES edges with a Component node and COMPONENT and USER edges until reaching a fixed point where all LIKES edges are replaced.

```
WITH $friendEdge AS friendEdge
MATCH (cp1:Component)-[:USER]->(u1:User)
  -[friendEdge]->(u2:User)<-[:USER]-(cp2:Component)
  <-[:COMPONENT]-(c:Comment)-[:COMPONENT]->(cp1)
WITH c, cp1, cp2,
     cp1.size AS cp1Size, cp2.size AS cp2Size,
     cp1.size + cp2.size AS newCompSize
CALL apoc.refactor.mergeNodes([cp1, cp2],
  {mergeRels: true}) YIELD node AS newComp
SET newComp.size = newCompSize,
    c.score = c.score - cp1Size*cp1Size
    - cp2Size*cp2Size + newCompSize*newCompSize
```

Listing 7: Q2 – for every FRIEND edge inserted where the two users belonged to separate components, merge the components and maintain the size and scores accordingly. (A similar query exists for new LIKES edges.)

we used the reference implementation of the case study, written in the .NET Modeling Framework [4] (NMF Batch) and its incremental version (NMF Incremental). We executed the benchmark on a cloud machine with a 24-core Intel® Xeon® Platinum 8167M CPU with Hyper Threading at 2.00 GHz, 320 GB RAM, and HDD storage. The execution times are shown in Fig. 4. The results show that the batch variant does not scale, as it takes more than 20 minutes for graph size 8. However, the incremental Neo4j variant is able to scale for all graph sizes. Neither of them is competitive against the incremental NMF solution which achieves sub-second reevaluation times for both queries.

IV. Conclusion and Future Work

This paper presented a Neo4j-based solution for the "Social Media" case study of the 2018 Transformation Tool Contest. We discussed a number of techniques that allow incremental

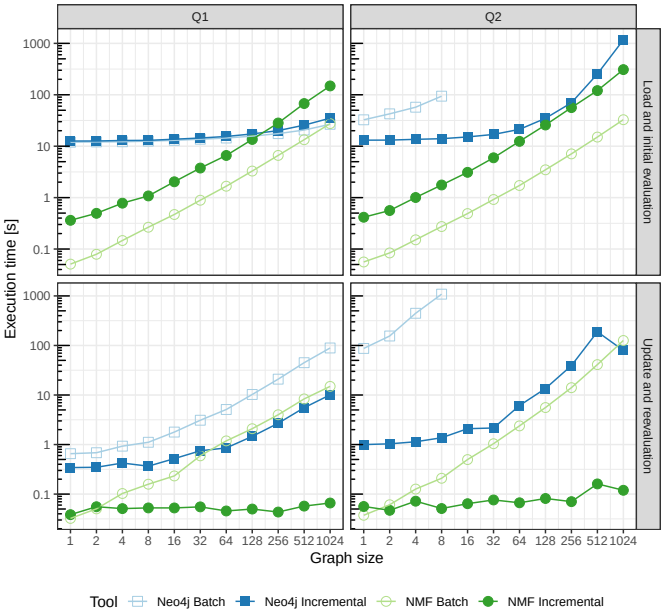


Fig. 4: Execution times of the queries with respect to the graph sizes in the "load and initial evaluation", and the "update and reevaluation" phases. Both the x axis (graph size) and the y axis (execution time) are logarithmic.

	1	2	4	8	16	32	64	128	256	512	1024
#nodes	1274	2071	4350	7530	15k	30k	58k	115k	225k	443k	859k
#edges	2533	4207	9118	18k	35k	71k	143k	287k	568k	1.1M	2.3M

TABLE II: Graph sizes w.r.t. to the scale factor.

evaluation during updates. Initial results show that incrementalization techniques provide significant performance benefits and allow the solution to scale for graph sizes orders of magnitude larger than batch solutions. In the future, we plan to perform a detailed performance evaluation of our solution against other query and transformation tools, including traditional relational databases (such as PostgreSQL), EMF-based model query engines (such as VIATRA [8]), and differential dataflow engines [6]. Our Neo4j solution is also subject to further optimizations such as using an incremental connected components algorithm [1].

References

[1] D. Ediger et al. Tracking structure of streaming social networks. In *IPDPS*, pages 1691–1699. IEEE, 2011.

[2] N. Francis et al. Cypher: An evolving query language for property graphs. In *SIGMOD*, pages 1433–1445. ACM, 2018.

[3] A. Green et al. Updating graph databases with Cypher. *PVLDB*, 2019.

[4] G. Hinkel. NMF: A multi-platform modeling framework. In *ICMT*, 2018.

[5] G. Hinkel. The TTC 2018 Social Media case. In *TTC at STAF*, 2018.

[6] F. McSherry et al. Differential dataflow. In *CIDR*, 2013.

[7] M. Needham and A. E. Hodler. *Graph Algorithms: Practical Examples in Apache Spark and Neo4j*. O'Reilly Media, 2019.

[8] D. Varró et al. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Softw. Syst. Model.*, 2016.