An approach to efficiently storing property graphs in relational databases

Work in progress paper

Matthias Schmid
University of Passau
Germany
Matthias.Schmid@uni-passau.de

ABSTRACT

Graph structured data can be found in an increasing amount of use-cases. While there exists a considerable amount of solutions to store graphs in NoSQL databases, the combined storage of relationally stored data with graph structured data within the same system is not well researched. We present a relational approach to storing and processing huge property graphs, which is optimized for read-only queries. This way, all the advantages of full-fledged relational database systems can be used and the seamless integration of classical relational data with graph-structured data is possible.

1. INTRODUCTION

The increasing appearance of data graphs reinforce the need for adequate graph data management. To meet this requirement of property graph storage, several graph databases engines have been developed. The most popular[1] examples of databases with graph storage capability include Neo4j[2], Microsoft Azure Cosmos DB[3] and OrientDB[4]. While these graph databases offer good performance as long as the data fits into main memory, one can not always assume that this requirement can be fulfilled. In addition the seamless integration of those solutions with relational-based information system remains a problem.

Motivated by our own use-case, in which we need to store huge graphs that represent buildings, *RDF* data that semantically describes those buildings and relational data into a single information system, we were looking for a relational-based solution that offers efficient read-focused performance. In *SQLGraph:An Efficient Relational-Based Property Graph Store* [9] Sun et al. introduce an efficient approach to store property graph data in relational databases named *SQLGraph*. While the original approach seems to offer good performance and promises to also perform well on datasets that do not fit into main memory, we were able to increase its performance for read-only queries by adapting the adjacency list concept.

The contributions of this paper are:

1. We propose a new adaption of the relational schema presented in [9],
2. we show that the adapted schema performs better on read-only queries
3. and we show that the new schema requires less disk space.

2. RELATED WORK

In this work we focus on the relational-based storage of property graph data. Bornea et. al first proposed the approach of shredding graph edges into adjacency tables in [1]. Sun et al. base their work in [9] on the previously mentioned work of Bornea et. al and outlined a novel schema layout for storage of property graph data in relational databases, which is generalized from the approach to store RDF data in relational databases. They combine the shredding of edges into adjacency tables with the use of *JSON*-based attribute storage to overcome the limitations of fixed columns. To make the retrieval of data more convenient, they propose a translation mechanism that converts *Gremlin* [8] queries to *SQL* queries, which can be run on the proposed relational schema.

In the field of *RDF* there exist numerous benchmarking efforts. But to the best of our knowledge the benchmarks provided by the *Linked Data Benchmark Council (LDBC)* are the only benchmarks that directly address the problem of benchmarking property graph stores. We do not consider graph processing frameworks like *Apache Giraph*[5] in our work, since we focus on the storage and retrieval of graph data, not its parallel processing. The *LDBC* is an EU project with the goal to develop benchmarks for graph structured data. They strife to find the acceptance benchmarks like the TPC [7] have achieved. The *Linked Data Benchmark Council - Social Network Benchmark (LDBC-SNB)* is one approach being developed by the *LDBC* that uses a generated social network graph as its data set and represents the

[1] from https://db-engines.com/de/ranking/graph+dbms, if not explicitly stated, all websites were accessed in February 2019
[2] https://neo4j.com/
[3] https://azure.microsoft.com/de-de/services/cosmos-db/
[4] https://orientdb.com/
[5] http://giraph.apache.org/

31^{st} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 11.06.2019 - 14.06.2019, Saarburg, Germany.

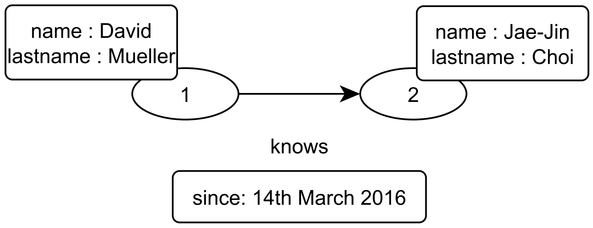

Figure 1: A property graph example

data as a property graph. The benchmark suite also provides a data set generator that can create test data with different scale factors. An overview of different scale factors and the corresponding number of vertices and edges is depicted in Table 1. Works on the *Social Network Benchmark* are not finished yet, but the *Interactive Workload* has been released in draft stage [4].

SF	Approx. #Vertices	Approx. #Edges
1	3,200,000	17,300,000
3	9,300,000	52,700,000
10	30,000,000	176,600,000
30	99,400,000	655,400,000

Table 1: Approximate number of vertices and edges by LDBC-SNB scale factor (SF)

3. SQLGRAPH SCHEMA ADAPTATION

For this paper we use the following definition of a property graph.

Definition 1. A **property graph** consists of a set of vertices and directed edges. Every edge has a label assigned to it. Each vertex or edge can additionaly have multiple key-value pairs assigned that serve as attributes.

Figure 1 shows an example of a simple property graph that describes two people, who are connected by an edge with the label *knows*.

3.1 Proposed Schema

Our approach utilizes the combined approach of relational and *JSON*-based storage developed in [9]. We adapted the schema of adjacency lists and as a result, our proposed schema reduces the amount of required tables from six (as in the original *SQLGraph* schema) to four tables.

The **vertex table** stores the internal vertex id and the attributes for each vertex. In our prototypical implementation the attributes are implemented as the *jsonb* data type provided by *PostgreSQL*.

VID	Attributes
1	**{"lastName": "Mueller", "firstName": "David"}**
2	**{"lastName": "Choi", "firstName": "Jae-Jin"}**

Table 2: The vertex attributes table

Our approach **differs** from the SQLGraph schema in respect to the definition of the **adjacency tables**. In their work Sun et al. [9] show that shredding vertex adjacency lists into a relational schema provides a significant advantage over other mechanisms, for example mechanisms that store all edge information in a single table. To this end a hash function has to be defined that matches edge labels to a respective column triple of the adjacency table. We directly apply the approach to compute a hash function through the use of coloring heuristics as described in [1]. Note that, since the LDBC-SNB data set includes a data model, it is possible to compute a conflict free hash function for this specific data set. In the original approach in [9] the edges are split into two tables: If only one edge of a label exists for the vertex, it's edge-id, label and target node-id are stored in the *outgoing primary adjacency (OPA)* and *incoming primary adjacency (IPA)* tables respectively. By studying available data sets and use cases (e.g. the *LDBC-SNB*) one can see that it is usually not the case that only one edge of a specific label exists for a single node. Therefore if the vertex has multiple outgoing edges of the same label, the edge-ids and target vertices are stored in the outgoing secondary adjacency (OSA) and incoming secondary adjacency (ISA) tables respectively, while the target vertex id of the *OPA* or *IPA* is set to an artificial value that serves as the join partner for the *OSA* and *ISA* table. A query to receive all outgoing or incoming neighbours can be written with the use of an outer join. If there exists only a single edge, the outer join will not find a corresponding partner in the secondary table and therefore use the data in the primary table. If more edges do exist, the join partners will be the resulting edges. This query works independently of the number of edges per label and the hash function. We will present an example of this query structure for our schema adaption, that omits the outer join, later in this section.

Nevertheless, this means every edge hop of a complex query requires an (outer) join-operation between the primary and secondary adjacency tables. This even is the case, if only a single edge of the required type is present, since the construction of the query should be independent of the hash function. Because the retrieval of direct neighbours of a vertex is one of the most important operations in any graph application, this join poses a major bottleneck for most queries.

In our approach we eliminate the need for a join-operation between OPA and OSA by storing all edges of one type in the corresponding columns using arrays. Table 3 shows an example of our adjacency tables: In the graph are two *knows*-edges that point from the vertex with id *1* to vertices *2* and *3*. Since in this example the *knows* and the *likes* label are hashed to the same column triple, a second row for vertex *1* has to be inserted. Note that this can be omitted with a better hash function or by providing more columns. An evaluation of optimal numbers of columns, if no hash function with a low amount of resulting conflicts for a given amount of columns can be found, has not been part of our research yet and will have to be addressed in the future.

Since we can provide a conflict free hash function for the *LDBC-SNB* data set, an outgoing neighbourhood query for a specific vertex can be answered by loading a single tuple from the database. By eliminating the need of an outer join for any edge hop, a major drawback of the original schema is overcome.

VID	EID_1	$Label_1$	$Targets_1$	...	EID_k	$Label_k$	$Targets_k$
1	[4, 5]	knows	[2, 3]		[11]	creator_of	[13]
1	[12]	likes	[7]		null	null	null
2	[6]	likes	[7]		null	null	null

Table 3: The adapted outgoing adjacency table

```
WITH unshred_edges AS (
  SELECT vertexid AS sourceid ,
    UNNEST(array[label_0 , ... , label_k]) AS label ,
    UNNEST(array[target_0 , ... , target_k]) AS tmp
  FROM outgoing_adjacency
  WHERE vertexid = <?>
),
gather_edges AS (
  SELECT sourceid , label ,
    array_elements(tmp) AS targetid
  FROM unshred_edges
  WHERE label = <?>
)
SELECT source_id , label , target_id
FROM gather_edges
```

Listing 1: Example neighbourhood query for a single node

An example for the general query structure implemented in the *PostgreSQL* dialect is depicted in Listing 1. The first common table expression *unshred_edges* converts the list of tuples belonging to one vertex, of which every row contains k edge types, into a list of rows that contains one edge type per row. The second common table expression *gather_edges* then splits those tuples into a list of edges that represents a well known edge structure.

Analogue to the storage of vertex attributes, we store **edge attributes**. As in the original schema, we do not only store attributes in this table, but also additional information about the edges. Namely the source vertex, the target vertex and the label of the edge are recorded. In [9] it is claimed, that queries that require to trace along an variable number of edges are performed much more efficiently when the edge table is used instead of the adjacency table. Our preliminary evaluations support this claim.

EID	SID	TID	Label	Attributes
4	1	2	knows	{"since" : "14.03.2016"}
5	1	3	knows	{"since" : "20.04.2016"}
6	2	7	likes	null

Table 4: The edge attributes table

3.2 Preliminary Schema Evaluation

In order to evaluate and test our concepts we have created a prototypical implementation of the concepts in *PostgreSQL*.

3.2.1 Methodology

Our goal was to compare the read-only query performance of our adaptation with the original *SQLGraph* schema. To this end we used part of the *Interactive Workload* of the *LDBC-SNB* [2, 4, 10].

We generated different data set sizes using the generator and performed tests with data sets that fit into the main memory of the system as well as data set sizes that are too huge to fit into main memory. To confirm the requirement of redundant representation of edges, we defined path queries with different fixed length and also path queries of variable lengths. After we confirmed the necessity of the edge tables, we also chose several queries provided by the *Interactive Workload* of the *LDBC-SNB* to evaluate the performance of our adjacency implementation against the original schema. We chose test queries applying the following criteria:

- The main pattern of the query uses paths of fixed length and use edges of known label. This criteria is desired, because preliminary evaluations (see [6]) have shown, that in other cases the use of the attribute table performs better than the adjacency tables.
- The set of queries requires the use of incoming, outgoing and undirected edges to cover all combinations of the adjacency tables.
- Different lengths of paths are contained in the set of queries to evaluate, if the difference in path length makes one of the two schema versions preferable.
- Queries with big intermediate result sets are part of the query set, since [9] states this type of query as a potential bottleneck of the original approach.

The parameters required by the queries were randomly chosen from the generated data set files and generated parameters. The same parameters were used for both approaches.

To make the performance of the two schemas as comparable as possible, we first implemented all queries for the adapted schema. Then we replaced only the necessary subqueries that concern the differences between the schemas, changing as little of the query as possible. We confirmed the correctness of our query implementations by comparing the results with results returned by a reference implementation provided for *Neo4j*, of which correctness has been validated.[6]

We evaluated the previously chosen queries on the same hardware using the same data set. The evaluation was conducted on a dedicated server with two Intel Xeon 2.4GHz CPUs (in total 16 cores), 64GB memory and a single 240GB SSD running 64-bit Linux. We used *PostgreSQL 10* on the aforementioned server, while we ran the client program of the benchmark on a standard laptop that connected to the database over LAN. All queries that were performed for performance evaluation purposes were proceeded by several warm-up queries as in most state-of-the-art benchmarks and also advised in [3].

All examples previously shown in this paper are a simplified version of the test data provided by the *LDBC-SNB* data generator.

[6]https://github.com/PlatformLab/ldbc-snb-impls/tree/master/snb-interactive-neo4j

3.2.2 *Redundant Edge Data*

Our findings support the need for redundant storage of edge data in the edge attributes table as well as the adjacency table as claimed by [1, 9]. We find a general tendency for queries to perform significantly better with the use of adjacency tables, if the queried path is of fixed length. While this is true for paths of fixed length, the opposite holds for paths of variable length. This type of queries requires recursive SQL queries. Recursive queries with the use of the edge attributes table outperform any recursive query that uses adjacency tables. In her Master's thesis Kornev [6] conducted an extensive evaluation, that shows when to use the edge attributes table in favor of adjacency lists.

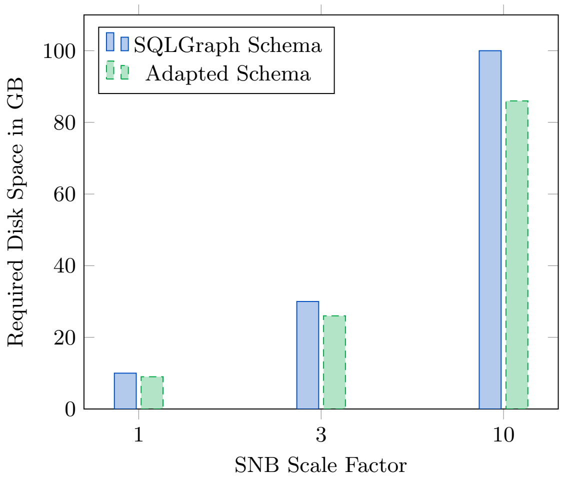

Figure 2: Required storage capacity

3.2.3 *Required Disk Space*

Since we reduced the number of tables, we expected a reduction in required storage capacity. To this end we imported different sizes of generated data sets provided by the *LDBC-SNB* data set generator. The generator creates data sets according to an input scale factor $(1, 3, 10, 30, \ldots)$. We then sampled the required storage capacity using *PostgreSQL* functions. The numbers shown in Figure 2 represent the complete graph schema, including all indexes and constraints. Due to hardware constraints we could not yet import data sets with a scale factor greater than 10. We will address higher scale factors in future work. Nevertheless our findings show, that our adaption of the schema reduces the required storage space by over 10%. This is mainly due to the reduced overhead by removing one table and the associated index structure that is needed to efficiently join the *OPA*/*IPA* and *OSA*/*ISA* tables.

3.2.4 *Query Performance*

We expected a reduction in query execution time for retrieval queries that use the adjacency lists. To confirm this we evaluated our approach with the use of queries defined by the *Interactive Workload*. First we chose four queries of the short query set. This type of query usually requires the system to evaluate a relatively low amount of vertices and edges to compute the answer, typically the neighbours of one entity of the data set [4]. The results shown here were performed against the generated data set with scale factor 10. This data set size does not fit in the main memory of the server.

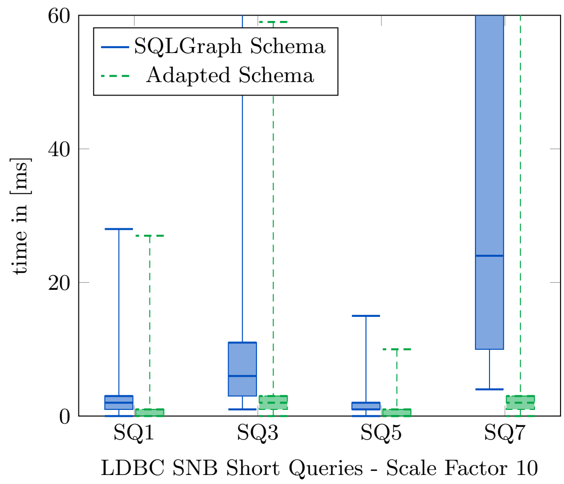

Figure 3: Runtime of the Interactive Workload Short Queries

Figure 3 shows the results of the conducted tests in a boxplot. The upper bound of the box shows the 75th percentile, the line within the box shows the 50th percentile and the lower bound of the box shows the 25th percentile, while the whiskers show the maximal and minimal execution time respectively. The results of the simpler queries confirm our expectation regarding execution time. The schema adaption improved query performance across the board for these queries. As we expected, the more edge hops a query performs, we can observe a bigger difference in execution time. Note that *short query 3 (SQ3)* uses an undirected edge. Since our schema does not directly support undirected edges, this query results in a *SQL* query that uses both the incoming adjacency and outgoing adjacency tables and therefore is similar to a 2-edge-hop.

We then additionally conducted experiments on four complex read-only queries of the *Interactive Workload*. Complex queries touch a significant amount of data and often include aggregation [4]. The chosen queries differ in the count of required edge-hops with *CQ 12* containing the highest amount of hops.

Once again our findings confirm an increase in read performance of our schema adaptation. Considering complex queries the difference in execution time becomes even more apparent, which confirms the findings in [9] that stated that big intermediate join results, which is a point of focus for the set of complex queries, can be a bottleneck for their approach. Our approach significantly increases query performance for these types of queries.

4. CONCLUSION

In this paper we have described our approach to store property graph data in a relational database. Our approach reduces the number of joins that are required for queries that contain paths of fixed length compared to earlier approaches. We have conducted a preliminary evaluation of our approach using part of a standardized benchmark for property graphs, namely the *LDBC-SNB*. The evaluation

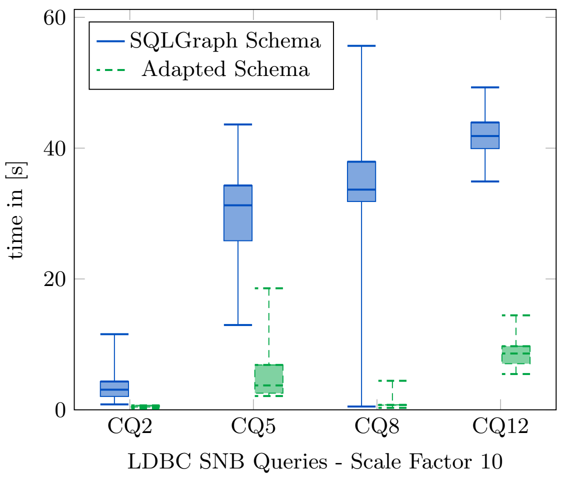

Figure 4: Runtime of the Interactive Workload Complex Queries

results show that the adaption is a more efficient version of the database schema in regards to read-only queries. Due to the reduction in the amount of required tables and therefore also index structures, we were also able to reduce the required disk space.

Therefore we have found an approach that enables us to efficiently store the property graph data from our use-case in a relational database and link it with the already existing relational data. In addition, [9] show an approach to store *RDF* data in a property graph model. Thus, we can efficiently integrate all data of our use-case in a single relational database.

5. FUTURE WORK

Future evaluations will have to show the validity of the approach compared to native database systems like *Neo4j*. Since most *NoSQL* and *graph database* systems focus on main memory computation, we expect those to perform comparable or better than our approach on data sets that can be handled in main memory. On the other hand we expect our approach do outperform non-relational based systems on data sets that require regular disk accesses due to their size. These points will be addressed in future evaluations.

Additionally we will evaluate the approach more extensively using the complete *LDBC-SNB* Interactive Workload on bigger data sets and more realistic server hardware. The workload also contains queries that insert data. We expect updates to also perform efficiently.

One major drawback of this approach is the complexity of the queries required to retrieve the data. To that end systems like *Neo4j* offer a very abstract and convenient way to query data with graph query languages like *Cypher* [5]. We propose a translation mechanism that converts *Cypher* queries to *SQL* queries that can be evaluated by our implementation on *PostgreSQL*. Kornev [6] has already shown that a translation of *Cypher* queries to *SQL* is possible. Unfortunately the translated queries do not in all cases achieve the efficiency that can be achieved with queries written by an expert. Therefore more optimization possibilities for the translation mechanism need to be explored. This will be part of our future research.

6. ACKNOWLEDGMENTS

This research was partially supported by the "Bayrische Staatsministerium für Wirtschaft, Energie und Technologie" in the context of "BaBeDo" (IUK568/001) which we conduct in partnership with *VEIT Energie GmbH*.

7. REFERENCES

[1] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. pages 121–132, 06 2013.

[2] P. A. Boncz. LDBC: benchmarks for graph and RDF data management. In *17th International Database Engineering & Applications Symposium, IDEAS '13, Barcelona, Spain - October 09 - 11, 2013*, pages 1–2, 2013.

[3] D. Dominguez-Sal, N. Martinez-Bazan, V. Muntes-Mulero, P. Baleta, and J. L. Larriba-Pey. A discussion on the design of graph database benchmarks. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 25–40. Springer, 2010.

[4] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 619–630, New York, NY, USA, 2015. ACM.

[5] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, New York, NY, USA, 2018. ACM.

[6] L. S. Kornev. Cypher für sqlgraph. Master's thesis, University of Passau, 2017.

[7] M. Poess and C. Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[8] M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, pages 1–10, New York, NY, USA, 2015. ACM.

[9] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1887–1901. ACM, 2015.

[10] G. Szárnyas, A. Prat-Pérez, A. Averbuch, J. Marton, M. Paradies, M. Kaufmann, O. Erling, P. A. Boncz, V. Haprian, and J. B. Antal. An early look at the LDBC social network benchmark's business intelligence workload. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Houston, TX, USA, June 10, 2018*, pages 9:1–9:11, 2018.

APPENDIX

A. EVALUATION QUERIES: INTERACTIVE WORKLOAD

Query	Cypher Representation
SQ 1	MATCH (n:Person {id:{id}})−[:IS_LOCATED_IN]−(p:Place) RETURN n.firstName AS firstName, n.lastName AS lastName, n.birthday AS birthday, n.locationIP AS locationIp, n.browserUsed AS browserUsed, n.gender AS gender, n.creationDate AS creationDate, p.id AS cityId
SQ 3	MATCH (n:Person {id:{id}})−[r:KNOWS]−(friend) RETURN friend.id AS personId, friend.firstName AS firstName, friend.lastName AS lastName, r.creationDate AS friendshipCreationDate ORDER BY friendshipCreationDate DESC, toInt(personId) ASC;
SQ 5	MATCH (m:Message {id:{id}})−[:HAS_CREATOR]−>(p:Person) RETURN p.id AS personId, p.firstName AS firstName, p.lastName AS lastName;
SQ 7	MATCH (m:Message {id:{id}})<−[:REPLY_OF]−(c:Comment) −[:HAS_CREATOR]−>(p:Person) OPTIONAL MATCH (m)−[:HAS_CREATOR]−>(a:Person)−[r:KNOWS]−(p) RETURN c.id AS commentId, c.content AS commentContent, c.creationDate AS commentCreationDate, p.id AS replyAuthorId, p.firstName AS replyAuthorFirstName, p.lastName AS replyAuthorLastName, CASE r WHEN null THEN false ELSE true END AS replyAuthorKnowsOriginalMessageAuthor ORDER BY commentCreationDate DESC, toInt(replyAuthorId) ASC;
CQ 2	MATCH (:Person {id:{1}})−[:KNOWS]−(friend:Person)<−[:HAS_CREATOR]−(message) WHERE message.creationDate <= {2} AND (message:Post OR message:Comment) RETURN friend.id AS personId, friend.firstName AS personFirstName, friend.lastName AS personLastName, message.id AS messageId, CASE exists(message.content) WHEN true THEN message.content ELSE message.imageFile END AS messageContent, message.creationDate AS messageDate ORDER BY messageDate DESC, toInt(messageId) ASC LIMIT {3};
CQ 5	MATCH (person:Person {id:{1}})−[:KNOWS∗1..2]−(friend:Person) <−[membership:HAS_MEMBER]−(forum:Forum) WHERE membership.joinDate>{2} AND not(person=friend) WITH DISTINCT friend, forum OPTIONAL MATCH (friend)<−[:HAS_CREATOR]−(post:Post)<−[:CONTAINER_OF]−(forum) WITH forum, count(post) AS postCount RETURN forum.title AS forumName, postCount ORDER BY postCount DESC, toInt(forum.id) ASC LIMIT {3};
CQ 8	MATCH (start:Person {id:{1}})<−[:HAS_CREATOR]−()<−[:REPLY_OF] −(comment:Comment)−[:HAS_CREATOR]−>(person:Person) RETURN person.id AS personId, person.firstName AS personFirstName, person.lastName AS personLastName, comment.creationDate, comment.id AS commentId, comment.content AS commentContent ORDER BY commentCreationDate DESC, toInt(commentId) ASC LIMIT {2};
CQ 12	MATCH (:Person {id:{1}})−[:KNOWS]−(friend:Person) OPTIONAL MATCH (friend)<−[:HAS_CREATOR]−(comment:Comment)−[:REPLY_OF]−>(:Post) −[:HAS_TAG]−>(tag:Tag), (tag)−[:HAS_TYPE]−>(tagClass:TagClass)−[:IS_SUBCLASS_OF∗0..] −>(baseTagClass:TagClass) WHERE tagClass.name = {2} OR baseTagClass.name = {2} RETURN friend.id AS friendId, friend.firstName AS friendFirstName, friend.lastName AS friendLastName, collect(DISTINCT tag.name), count(DISTINCT comment) AS count ORDER BY count DESC, toInt(friendId) ASC LIMIT {3};

Table 5: Chosen queries for preliminary evaluations of the adapted schema approach