Towards Speeding Up Graph-Relational Queries in RDBMSs

Angelos Christos Anadiotis, François Goasdoué, Mhd Yamen Haddad, Ioana Manolescu

Towards Speeding Up Graph-Relational Queries in RDBMSs

Angelos Christos Anadiotis*
Oracle
Zurich, Switzerland
angelos.anadiotis@oracle.com

François Goasdoué
Université de Rennes, Irisa
Rennes, France
fg@irisa.fr

Mhd Yamen Haddad
Inria & Institut Polytechnique de Paris
Palaiseau, France
mhd-yamen.haddad@inria.fr

Ioana Manolescu
Inria & Institut Polytechnique de Paris
Palaiseau, France
ioana.manolescu@inria.fr

ABSTRACT

Graph data is generally stored and processed using two main approaches: (*i*) extending existing relational database management systems (RDBMSs) with graph capabilities, and (*ii*) through native graph database management systems (GDBMSs). The advantage of leveraging RDBMSs is to benefit from the maturity of their query optimization and execution. Conversely, native GDBMSs treat complex graph structure as a first-class citizen, which may make them more efficient on complex structural queries.

In this work, we consider the processing of *graph-relational queries*, that is, queries mixing graph and relational operators, on graph data. We take a purely relational approach, reorganizing the graph connectivity information using a novel CSR Optimised Schema (COS). Based on our storage model, incoming queries are reformulated to take into account the COS data organization, and can then be optimized and executed by an RDBMS. We have implemented our approach on top of PostgreSQL and we demonstrate that COS improves the performance for many graph-relational queries of the popular Social Network Benchmark [5].

KEYWORDS

Graph data management, query execution, indexing, query optimization

1 INTRODUCTION

Graphs are ubiquitous in modern applications. They are used to represent financial data, social networks, etc. There are many ways to model and represent graphs, with the most popular being: the RDF model, adopted in the Semantic Web movement, and Property Graphs, increasingly studied and promoted by systems such as Neo4j [1], TigerGraph [4] and Graphflow [12].

In our study, we focus on Property Graph Data Model [20]. In this model, edges and vertices have labels and a set of properties. For example, a simple property graph contains vertices of labels "Person, University" and edges with "Follows, StudyAt" labels. Graph vertices and edges may also have properties, e.g., "Person" vertices may have (name, age, birthday). Thus, a graph database needs to store:

(1) The graph topology, representing the connectivity between nodes, typically stored as adjacency lists [10]. Such lists allow accessing the neighbors of a vertex in constant time.
(2) Vertices and edges properties.

The more regular the data is, the more suitable a relational model is to representing and storing the data [18]. On the contrary, if the graph is irregular, a key-value store may be used. In this paper, we consider the common case where the properties of nodes having a certain label are well-structured, thus they can be easily stored in a table. For example, all the properties of "Person" nodes might be stored in one table.

When storing a graph in an RDBMS, the topology is also stored in relations, allowing graph queries to be expressed in (possibly recursive) SQL. However, the performance of path traversals may be poor, because they translate into many join operators and thus significant random data access. Native GDBMSs aim at solving this issue by storing the graph in a way that entails fewer or no random accesses when traversing paths [15]. However, reorganizing the data in this way may not be favorable to the performance of select-project-join style queries.

In this work, we study ways to efficiently process both relational-style and traversal queries while storing the data in a relational format. Specifically, we investigate the impact of storing the topology of the graph in a form of "relational adjacency list", inspired by the CSR data structure previously proposed in [21]. We call the resulting storage model, separating node properties from the graph edges, **CSR Optimized Schema**, or COS, in short.

This paper is organized as follows. Section 2 introduces a motivating example, then Section 3 discusses graph storage models in RDBMSs, following which Section 4 describes our COS proposal. Section 5 describes the SQL query rewriting method we need in order to adapt incoming queries to our storage. We then discuss some optimization issues involved in processing the queries resulting from our approach in Section 6, before presenting our experimental evaluation in Section 7 and concluding.

2 MOTIVATING EXAMPLE

We consider graph data expressed using the *property graph data model*. In this model, a *graph* G is noted $G = (V, E)$ and consists of a set V of vertices and a set E of edges between these vertices. Furthermore, vertices and edges are *labeled* (or typed); a given label indicates with which set of properties (or attributes) a vertex or edge is described. A vertex is at least described with an id, while

*Work done while at Ecole Polytechnique.

BDA '22, October 24–27, 2022, Clermont-Ferrand, France
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

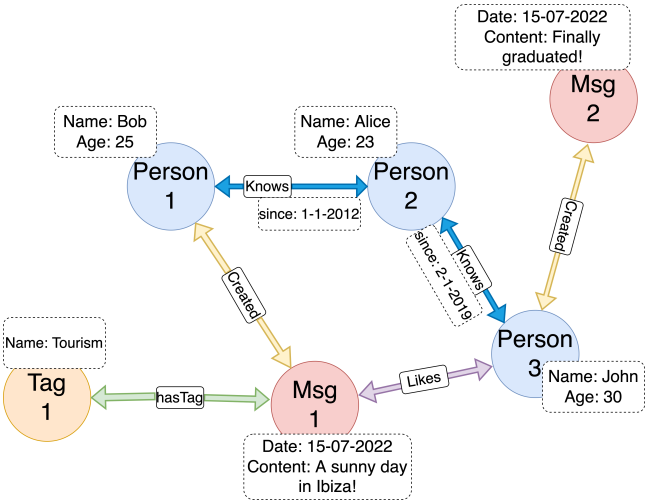

Figure 1: Sample property graph, representing a social network.

an edge is at least described with an id plus the ids of the vertices it connects.

Figure 1 displays a small social network represented as a property graph. Vertices (circles) are labeled with Person, Message (Msg, in short), and Tag, while edges (arrows) are labeled with Created, Knows, Likes, and hasTag. Same-label vertices are described using the same properties, e.g., each Person vertex is described with an id, a name, and an age. Similarly, edges sharing the same label are described using the same set of properties, e.g., each Knows edge is described by the ids of the two vertices it connects, and since when they know each other. We will use p_1, p_2, p_3 to refer to the Person nodes with the IDs 1, 2 and 3, respectively, and similarly use m_1 for the Message with ID 1, k_1 for the first Knows edge, etc. This graph states, for instance, that the p_1, whose name is Bob and age is 25, knows since 2012-01-01 the person p_2, whose name is Alice and age is 23. In addition, the person p_1 authored on 2022-07-15 the message m_1, whose content is "A sunny day in Ibiza!", etc.

Storing Vertex and Edge Properties A popular way to store node and edge attributes coming from a property graph, in a relational schema, consists of creating one relation for each vertex label, and each edge label. For instance, for the graph shown in Figure 1, Person vertices are stored in a Person(id, name, age) relation, Knows edges are stored in a Knows(id, src-id, dest-id, since) relation, etc, as illustrated in Figure 2. The Knows edges are bidirectional in our example; accordingly, each of them is represented by two directed edges in the storage.

Sample Query An example of *graph-relational query* that we might encounter in such graphs is:

- Given a **start person**, denote their *friends, and friends of friends* (excluding the start person) as **other person**.
- Find messages that any person from *otherPerson* has posted after a given **min date**. For those messages, count the number of each tag attached to the message.

The query consists of two components:

Person

id	name	age
p_1	Bob	25
p_2	Alice	23
p_3	John	30

Knows

id	src-id	dest-id	since
k_1	p_1	p_2	1-1-2012
k_2	p_2	p_1	1-1-2012
k_3	p_2	p_3	2-1-2019
k_4	p_3	p_2	2-1-2019

Figure 2: Part of the relational schema storing vertex and edge attributes for the social network example in Figure 1.

(1) Graph component, in this example, the traversal getting friends, and friends of friends;
(2) Relational component, specifically, the relational operators such as filtering and aggregation.

We do not currently handle unbounded-length (recursive) path traversals. Their translation into recursive SQL queries may lead to inefficient execution; to mitigate this, we should probably integrate into our approach some form of structural index, e.g., [6, 13, 16, 17], that can simplify the query and make it easier to optimize and evaluate. This is part of our future work.

3 GRAPH DATA LAYOUT

Various data structures have been used in GDBMSs to store the graph topology, on one hand, and the attributes of vertices and edges, on the other hand. We describe here the main data structures.

3.1 Adjacency Matrix

An adjacency matrix M is a two dimensional matrix of size $|V| \times |V|$ where $M[i][j]$ has a value of 1 if and only if there exists an edge between vertex i and vertex j; otherwise, it is 0. The neighbors of a vertex i are obtained by iterating over all the values stored in the i^{th} row of M, i.e., in $O(V)$. Adjacency matrices are particularly useful for queries that require fast checks of whether two vertices are connected since this operation is performed in $O(1)$. However, as its storage requirements are quadratic to the size of the vertex set, the adjacency matrix is generally not considered efficient for storing *sparse* graphs, that is, those such that $|E| \ll |V| \times |V|$.

3.2 Adjacency List

Adjacency lists [19] reduce the storage costs of adjacency matrices by storing only the neighbors of every vertex in the graph. Specifically, an adjacency list is a linked list that contains the neighbors of every vertex, with storage requirements of $O(|V| + |E|)$. However, the improvement in storage w.r.t. adjacency matrices comes at the price of increased cost for getting the neighbors of a given vertex i, since the linked list traversal incurs random accesses. Cache-friendly solutions [7, 22] have been proposed to amortize the linked list traversal problem by replacing the linked list of individual values with a linked list of fixed-sized arrays of values.

3.3 Compressed Sparse Row

The compressed sparse row (CSR, in short) [9] is a data structure used to store the topological information for sparse graphs. In a

nutshell, CSR stores the adjacency matrix of a graph in a compressed fashion. It consists of three one-dimensional arrays *offset*, *destination* and *edge* defined as follows:

- the *offset* array stores for each source vertex v the position of its first neighbor in the destination array,
- the *destination* array stores the destination vertices contiguously for every source vertex in the offset array, in the order of the source vertices in the offset array,
- the *edge* array stores the edge id that connects vertex v from the offset array with vertex d from the destination array.

When the data is in the CSR format, the neighbors of a vertex v are obtained by first probing the offset array to know where the neighbors of v are in the destination array *start*. The neighbors of v *end* at the offset of the vertex that follows v in the offset array. Then, retrieving the neighbors of v takes a scan of the values in the *destination* array in the range between $[start, end]$. CSR requires $O(|V| + |E|)$ for storage and it is very efficient in getting the neighbors of a specific vertex (scanning a continuous range in an array).

4 CSR OPTIMISED SCHEMA (COS)

This section describes how we optimize the standard relational schema for property graphs in a CSR-like fashion, to support graph operators efficiently. The objectives of our design are as follows:

- The new design should be general and works on any RDBMS without the need for a system-specific modification and customization. This means that our approach should be easy to deploy in any RDBMS without any changes to the system's internals.
- System users should not have to be aware of the underlying reorganization of the data, in particular, they should be able to query the graph as if vertex and edge attributes were stored in dedicated tables.

To reach our first objective, we replace each L(eid, src-id, dest-id, ep1,..., epn) edge relation that stores all the edges with label L (recall Section 2) into the two L-id-to-offset and L-destination tables defined as follows.

- L-id-to-offset(src-id, offset-start, length) stores every source vertex (src-id), as well as the offset at which starts its neighbours in the L-destination table (offset-start) and how many neighbours it has (length).
- L-destination(vid, eid, ep1,..., epm, dest-id) stores for every offset the L-labeled edge (eid) together with its properties (ep1,..., epn), as well as its destination vertex (dest-id).

We remark that vertex relations remain unchanged with our CSR optimized schema, as the topological information of a graph is stored in edge relations only. Vertex properties can be accessed knowing the node id by joining with the corresponding vertex relation.

Figure 3 shows how the Knows relation "left" that follows from Figure 1 is replaced by the corresponding two relations of the CSR optimized schema.

To reach our second design objective, we introduce a *query rewriting* procedure which takes as an input a user query unaware of the CSR-style reorganization, and produces a logically equivalent query over the COS, as we explain next.

CSR Optimized Schema

Knows relation

eid	src-id	dest-id
k_1	p_1	p_2
k_2	p_2	p_1
k_3	p_2	p_3
k_4	p_3	p_2

Knows-id-to-offset

nid	offset-start	length
p_1	0	1
p_2	1	2
p_3	3	1

Knows-destination

vid	eid	dest-id
1	k_1	p_2
2	k_2	p_1
3	k_3	p_3
4	k_4	p_2

Figure 3: Reorganising the knows table (left) by replacing it with the two tables of the CSR Optimised Schema (right)

5 QUERY REWRITING

We explain now how a SQL query expressed on the standard relational schema for property graphs is rewritten into a SQL query expressed on the CSR-optimised schema.

Every L(eid, src-id, dest-id, ep1,..., epn) edge relation in the FROM clause of a query is first replaced by the corresponding COS tables: L-id-to-offset(src-id, offset-start, length) and L-destination(vid, eid, ep1, ..., epm, dest-id). Furthermore, to relate the source vertices of L-edges with the appropriate destination vertices, we add the following condition in the WHERE clause: vid BETWEEN offset-start AND offset-start+length.

For instance, consider the property graph in Figure 1 and the following query asking for the name and age of every person known by a person named Bob:

```
SELECT p2.Name, p2.Age
FROM Knows k, Person p1, Person p2
WHERE k.src-id=p1.id AND p1.name='Bob' AND k.dest-id=p2.id
```

This query is rewritten as follows:

```
SELECT p2.Name, p2.Age
FROM Knows-id-to-offset kito, Knows-destination kd, Person p1,
Person p2
WHERE kito.src-id=p1.id AND p1.name='Bob' AND
kd.dest-id=p2.id AND kd.vid BETWEEN kito.offset-start
AND kito.offset-start+kito.length
```

The query then is sent to the PostgreSQL optimizer as any relational query, thus allowing it to be optimized and executed in the standard way.

As can be seen from our example, the positional encoding in COS leads to interval (inequality) joins, materialized by the BETWEEN predicate. This leads to specific optimization issues, as we discuss next.

6 COS AWARE CARDINALITY ESTIMATIONS

In this section, we explain an important challenge we had when running the set of graph-relational queries using the CSR index on top of PostgreSQL. As explained above, we replace each many-to-many edge table with the two corresponding CSR tables *id-to-offset, destination*, then join them together using a range filter in

the *WHERE* clause. In this way, we get the neighbors of a certain node using the *offset-start and length* columns to determine where to start scanning the adjacent nodes and for what length in the destination table. The main advantage, in this case, is that getting the neighbors of a node requires scanning only a continuous part of the destination table that have the neighbors without the need to scan the whole edge relation in the fully relational case.

However, when we implemented that in PostgreSQL, we noticed that running the queries using the CSR index tables yields slower execution time which was surprising at first (as scanning only the useful part of the edges relation should be faster than scanning the whole relation). When we investigated the query plans that PostgreSQL optimizer produced on the COS layout, we noticed the following:

- The optimizer chooses **hash join** to join some tables instead of using the **index nested loop join**, despite the fact that the index nested loop join is faster in execution. In order to know why the optimizer takes this decision, we investigated PostgreSQL cardinality estimators, which predict the number of output rows that each operator produces.
- It turned out that the cardinality estimator of PostgreSQL significantly overestimates the number of rows in the *destination* relation that satisfies the range filter to get the set of nodes in the range **[offset, offset+length]**. For example, if the range query asks for the rows in the destination relation that have *vid* values in the range [100,150], the actual number of returned rows is 51. However, the estimator predicts the output to be about 80.000 rows which is an overestimation of **1.600**×.

This overestimation leads the optimizer to prefer using a hash join for joining the output of the filter with the other operators. However, using hash join, in this case, makes the performance worse, because it needs to build a hash table for one side of the join and start probing using the other side. In contrast, using an index nested loop join would only need a few index accesses (51 in our example), and thus lead to faster execution. Therefore, we had to investigate further *why* the PostgreSQL estimator significantly overestimates the number of rows resulting from the range query. We discuss this next.

6.1 PostgreSQL Estimation of Range Queries

Consider a range query in the following form:

```
SELECT * FROM R
WHERE R.c >A AND R.c<B
```

where **R.c** is a column of integer values and **A** and **B** are some integer values. The estimator distinguishes between two cases in order to estimate the number of rows that validate the range filter:

(1) If **A** and **B** are known to the optimiser *at query optimization time* (i.e **A** and **B** are constants written in the query), then the PostgreSQL cardinality estimator uses its materialised statistics available about the column **R.c** in the system catalogue, such as the histogram of **R.c**, the set of most common values of **R.c**, and their frequencies, in order to estimate the number of tuples that satisfy the range predicate [3].

(2) Otherwise, **A** and **B** may be unknown to the optimizer at query optimization time, in particular when **A** and **B** are *values that we get from the tables that we scan and possibly select, join, etc. in the query itself.* In this case, *at query optimization time*, the estimator is unable to proceed as above, and instead defaults to some *default selectivity constants* [2] as a last resort.

Unfortunately, our COS layout falls in the second case: the role of values **A** and **B** is played by the **offset-start** and **offset-start+length**; they are not known at query optimization time, but become known only at runtime when they are read from the CSR index table. Looking within the source code of PostgreSQL [2] uncovered a default selectivity of 0.11 in an example case we considered: the optimizer estimated 800.000 rows in the table, combined this with the default selectivity factor, and concluded that approximately 80.000 rows were going to be selected when the true value was closer to 50 (1.600× overestimation).

6.2 Adapting Estimations to Graph Queries

In order to solve the problem discussed, we had to choose a good replacement for the default selectivity the PostgreSQL estimator uses in order to fit our case. The goal of the new selectivity value is to be used in the graph scenario and to not overestimate the number of rows that validates the range query. We also want this value to be generic enough so that it still behaves reasonably when moving to a different graph.

As the expected number of rows that validates the range query is the same number of nodes that are neighbors of a node in the graph, we expect the average degree of nodes in the graph to be a good default estimator for the range queries in the destination relation. From that argument, we can recalculate the appropriate default selectivity for a given graph and replace the default selectivity value in PostgreSQL. We did that and it gives the desired behavior from the query optimizer which chooses index nested loop join in the cases where the number of scanned vertices is not big. Our experimental results (see below) validate the interest in this modification.

7 EXPERIMENTAL EVALUATION

This section presents the experimental results that we obtained by evaluating the performance of the CSR Optimised Schema compared to traditional ways of graph query processing such as RDBMSs or GDBMSs. We consider queries that include both a graph and a relational component, to (*i*) demonstrate that COS can be easily applied on top of any relational database, and (*ii*) evaluate the performance of COS in a mixed workload with queries stressing either the graph or the relational component.

In all experiments, we report the execution time of running the same graph-relational query in two deployment settings: (*i*) when the graph edges are stored as many-to-many relation, we call this scenario the **fully relational** setting; (*ii*) after introducing our COS approach to represent the edges in the graph. We call this scenario the **COS** setting.

7.1 Experimental Setup

7.1.1 Software. A key COS advantage is that it can be trivially deployed on top of an already existing RDBMS. Accordingly, we

decided to use PostgreSQL v12.7 due to its maturity and its wide adoption.

7.1.2 Hardware. All experiments were executed on a machine equipped with two 10-core Intel Xeon E5-2640 v4 @ 2.40GHz CPUs and 256 GB of DRAM.

7.1.3 Benchmark. Following several works in the related literature, in our experiments, we used the dataset and the workload described in the **LDBC Social Network Benchmark** (SNB, in short) [5]. The SNB dataset represents a social network of people who can be friends with each other, join forums, and post messages as well as comments on the posts. The size of the graph is controlled by a scale factor. For instance, with scale factor 10, we generate about 36 million nodes and 124 million edges. Cocerning the workload, we consider the **Interactive Complex (IC)** queries that include both graph traversal and standard relational operators without changing the dataset. The queries access a large part of the database (often the two-step friendship neighborhood and associated messages). Nevertheless, their graph traversal component considers close proximity to a single node, that is from one to three hops away [8].

7.2 Experimental Results

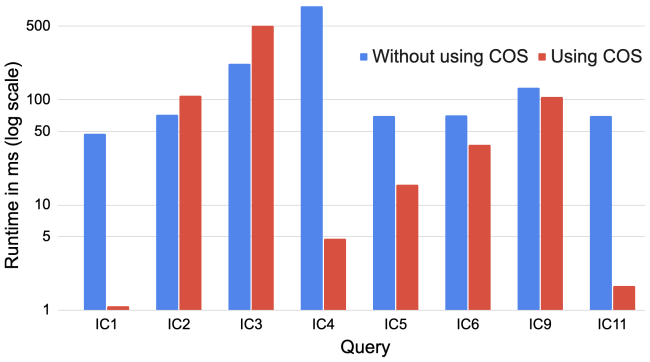

Figure 4: SNB-M 1-hop queries

We executed the set of Interactive Complex queries of SNB on both the fully relational scenario and the proposed COS scenario. We chose a subset of the SNB-IC queries where the execution varies between the two scenarios. This is the case for queries: 1 to 6, 9, and 11. The rest of the queries do not show any difference in the two scenarios and, therefore, we omit them from the experiments.

As our approach does not support recursive queries, we split the queries that traverse a range of hops into a set of queries with a fixed number of hops away from the source node. For example, IC1 asks for nodes of type person that are connected to starting person x through at most three *knows* edges. This query is *split into 3 queries*: IC1-1 which gives direct friends of a given node x, IC1-2 that gives friends of friends of x and IC1-3 which gives the set of nodes that are connected by exactly 3 knows edges to x. In the related literature [11] queries with this modification have been referred to as **SNB-Modified**. We study two cases described in the following sections.

7.2.1 1-hop Queries. We consider queries that explore the nodes in the graph that are 1-hop away from the starting node. This subset of queries are referred to in the SNB-M benchmark as **IC*-1**. Figure 4 shows the runtime of running SNB-M IC queries on PostgreSQL with and without using COS. We divide the queries according to the behaviour of the proposed approach into three main categories:
C1. Queries where COS improved the performance significantly such as IC1, IC4, and IC11 where the speedup compared to fully relational is 34×, 177×, and 41× respectively. Here, COS exploits the CSR access patterns to get the node neighbours efficiently, instead of doing the value based join used in traditional DBMSs.
C2. Queries where the COS performance improvement was smaller such as IC6, and IC9 where the speed up in execution was 1.9×, and 1.2× respectively. To explain why the speedup was smaller, we analyzed the execution plans produced by PostgreSQL for each one of the queries. We discovered that for queries IC6 and IC9, the graph traversal part of the query was executed very fast on the COS. However, the relational part of the query joins the nodes resulted from the graph traversal with large tables, hence dominating the execution time.
C3. Queries where COS underperformed such as IC2 and IC3 where the performance was slowed down by 1.5× and 2× respectively. By analyzing the execution plans, we discovered that most of the execution time is spent on getting the vertices' properties rather than in graph traversal. In order to get the properties, the query optimizer expects a small number of vertices and chooses an index nested loop join. For example, in IC3 there is an index nested loop join that accesses one table 310K times by index scan. A hash join would be a reasonable alternative. In fact, the optimizer underestimated the output of graph traversal in IC3 by a factor of 4700× which leads to so many index accesses to get the properties. Bad cardinality estimation has been shown to have a negative effect on the query plan [14].

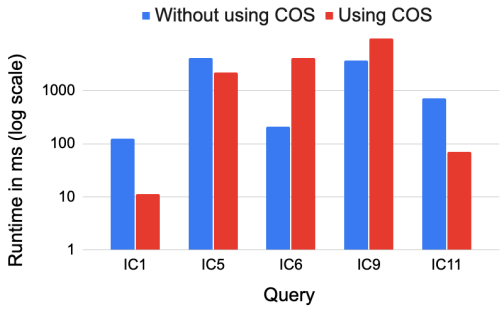

Figure 5: SNB-M 2-hop queries

7.2.2 2-hop Queries. We consider the queries that explore the nodes in the graph that are 2-hop away from the starting node. This subset of queries is referred to in the SNB-M benchmark as **IC*-2**. We selected a subset of the SNB-IC queries that explores more than 2 hops away from a starting node and compared their runtimes with/out using COS in Figure 5. The figure shows that using COS leads to performance improvements on queries IC1, IC5, and IC11, due to the COS's ability to efficiently get the neighbors of

a node in the graph. However, performance was hurt in IC6 and IC9. This is due to a bad cardinality estimation of the query optimizer on the output of the graph traversal part of the query, again resulting in an index nested loop join instead of hash join to get the vertex properties. The problem is more obvious in 2-hops queries as the number of vertices resulted from graph traversal is significantly larger than a 1-hop query. Therefore, more index scans occur which leads to slower execution.

Overall, we observe that COS enhanced the performance in most of the SNB graph-relational queries, whereas the graph traversal part is always improved. The relational part of the query slows down the execution in some queries due to suboptimal query plans. We can avoid the slowing down in the relational part of some queries by having better cardinality estimations which can lead to better plans.

7.3 Scalability Experiments

We investigate how the COS approach scales with the size of the graph (in terms of the number of nodes and edges). We chose a subset of the SNB-M IC queries and we report the runtime of this query with different scale factors. We executed the queries of IC*-2, which traverse 2-hops in the graph, with **SF3** (3GB dataset) and **SF10** (10GB dataset) and we report the results in Figure 6.

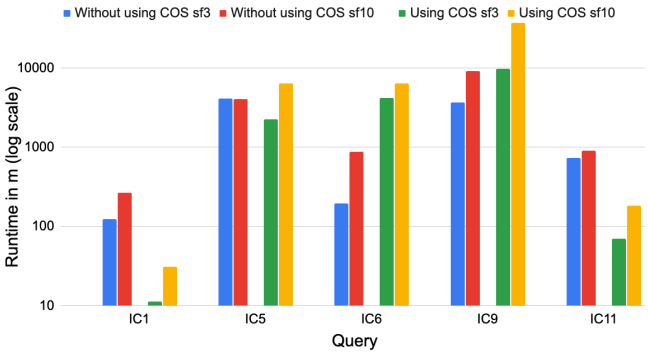

Figure 6: SNB-M 2-hop with SF3 and SF10

We can see in Figure 6 that executing the same query on SF10 has a larger runtime compared to SF3, with and without using COS, as expected. Another interesting observation is that COS scalability seems to be better correlated to the size of the graph compared to the fully relational model, that is: the running time for most queries at SF10 using COS is roughly 3× more than for SF3. In contrast, in the fully relational model, the running time did not change much between SF3 and SF10 in some queries like IC5, or it become 4× larger in some others like IC6. The reason for the constant execution time of IC5 is the fixed cost spent in building the hash table of the edges' relationship.

7.4 Conclusion

The experiments demonstrated that CSR-organised schema provides performance improvements on some of the Interactive-Complex queries of SNB compared to storing the graph in a fully relational schema. We showed how some queries can have significant, *orders-of-magnitude* improvements in performance compared to the fully relational version. Some of them have a slightly better performance due to the large portion of the relational component, and some register a performance penalty due to bad cardinality estimation from the query optimizer. We also showed how running the queries on top of COS scales linearly with respect to the size of the graph. More experiments can be done on the cardinality estimation of the query optimizer to further improve the performance.

Acknowledgments. This work is funded by ANR CQFD (ANR-18-CE23-0003).

REFERENCES

[1] Neo4j. https://neo4j.com/, 2022. [accessed 10-June-2022].
[2] PostgreSQL cardinality estimation defaults. https://github.com/postgres/postgres/blob/master/src/include/utils/selfuncs.h, 2022. [accessed 10-June-2022].
[3] PostgreSQL cardinality estimation documentation. https://www.postgresql.org/docs/current/row-estimation-examples.html, 2022. [accessed 10-June-2022].
[4] TigerGraph. https://www.tigergraph.com/, 2022. [accessed 10-June-2022].
[5] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. The LDBC social network benchmark. *CoRR*, abs/2001.02299, 2020.
[6] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, and Javiel Rojas-Ledesma. Time- and space-efficient regular path queries on graphs. 2022.
[7] David Ediger, Jason Riedy, David Bader, and Henning Meyerhenke. Tracking structure of streaming social networks. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1691–1699, 2011.
[8] Orri Erling, Alex Averbuch, Josep-Lluis Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. The LDBC Social Network Benchmark: Interactive workload. In *SIGMOD*, pages 619–630, 2015.
[9] Soukaina Firmli, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, Chiadmi Dalila, Sungpack Hong, and Hassan Chafi. Csr++: A fast, scalable, update-friendly graph data structure. 12 2020.
[10] Alon Itai and Michael Rodeh. Representation of graphs. *Acta Inf.*, 17, 06 1982.
[11] Guodong Jin and Semih Salihoglu. Making rdbmss efficient on graph workloads through predefined joins. *Proc. VLDB Endow.*, 15(5):1011–1023, jan 2022.
[12] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1695–1698, New York, NY, USA, 2017. Association for Computing Machinery.
[13] Jochem Kuijpers, George Fletcher, Tobias Lindaaker, and Nikolay Yakovets. Path indexing in the cypher query pipeline. In *EDBT*, pages 582–587, 2021.
[14] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.
[15] Amine Mhedhbi, Pranjal Gupta, Shahid Khaliq, and Semih Salihoglu. A+ indexes: Lightweight and highly flexible adjacency lists for graph database management systems. *CoRR*, abs/2004.00130, 2020.
[16] Inju Na, Ilyeop Yi, Kyu-Young Whang, Yang-Sae Moon, and Soon J. Hyun. Regular path query evaluation sharing a reduced transitive closure based on graph reduction. *CoRR*, abs/2111.06918, 2021.
[17] You Peng, Xuemin Lin, Ying Zhang, Wenjie Zhang, and Lu Qin. Answering reachability and k-reach queries on large graphs with label constraints. *VLDB J.*, 31(1):101–127, 2022.
[18] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. Sqlgraph: An efficient relational-based property graph store. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.
[19] Wen Sun, Kavitha Srinivas, Achille Fokoue, Anastasios Kementsietsidis, Gang Hu, and GuoTong Xie. Sqlgraph: An efficient relational-based property graph store. In *SIGMOD*, 2015.
[20] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pgql: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, New York, NY, USA, 2016. Association for Computing Machinery.
[21] Brian Wheatman and Helen Xu. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, 2018.
[22] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 183–193, 2015.