Cytosm: Declarative Property Graph Queries Without Data Migration

Benjamin A. Steer
Queen Mary University of London
b.a.steer@qmul.ac.uk*

Alhamza Alnaimi
Hewlett Packard Labs
alhamza.alnaimi@hpe.com

Marco A. B. F. G. Lotz
Hewlett Packard Labs
lotz@hpe.com

Felix Cuadrado
Queen Mary University of London
felix.cuadrado@qmul.ac.uk

Luis M. Vaquero
Hewlett Packard Labs
luis.vaquero@hpe.com

Joan Varvenne
Hewlett Packard Labs
joan.varvenne@hpe.com

ABSTRACT

The property graph model has recently gained significant popularity, combining great expressiveness with powerful declarative graph query languages. However, in order to take advantage of these features, data must be loaded into a specialised graph database. Additionally, property graphs are often schema-free, complicating efficient query execution. In this paper we present Cytosm, a middleware application which enables the execution of property graph queries, on non-graph databases, without data migration. Cytosm relies on gTop, a schema containing an abstract property graph topology, and its mapping to specific database backends. Cytosm uses gTop to efficiently execute OpenCypher queries, exploiting schema information to optimise the query plan, and mapping query concepts to the relational backend. Our experiments show that Cytosm achieves competitive query execution times on relational backends, when compared to leading graph databases.

ACM Reference format:
Benjamin A. Steer, Alhamza Alnaimi, Marco A. B. F. G. Lotz, Felix Cuadrado, Luis M. Vaquero, and Joan Varvenne. 2017. Cytosm: Declarative Property Graph Queries Without Data Migration. In *Proceedings of GRADES'17, Chicago, IL, USA, May 19, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3078447.3078451

1 INTRODUCTION

Property Graphs provide a simple method to represent complex information as a network of entities (nodes) interconnected via relationships (edges). This alongside domain-specific declarative query languages, such as OpenCypher, provides a powerful new way to store, process and understand data. Unfortunately, especially in larger companies, data is frequently stored in non graph-specific databases. When managing up to Petabytes of information, the advantages gained from newer graph systems seldom outweigh the extra storage, complexity and Extract Transform Load (ETL) processing required to update their infrastructure.

GRADES'17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3078447.3078451

In this paper we present Cytosm (Cypher to sql mapping): a middleware application which enables the execution of graph queries, on non graph databases, without data migration. Furthermore, we present gTop, a schema-like format which captures the structure of property graphs, providing a flexible mapping between Graph Query Languages (GQL's) and a variety of underlying database systems. The most significant parts of Cytosm have now been released as open source[1].

Our experimentation shows that OpenCypher queries translated via Cytosm have a similar execution time to manually tailored SQL queries and, perhaps more surprisingly, have times comparable to the same queries executing on leading dedicated graph databases.

The rest of this paper is organised as follows: Section 2 discusses the components of a graph topology (*gTop*) file, how the mapping is implemented and methods of *gTop* generation; Section 3 explains the full translation pipeline, showing how an OpenCypher query is broken down and converted into an efficient SQL alternative; Section 4 evaluates the performance of Cytosm compared to a dedicated graph database and other state-of-the-art solutions; Section 5 looks at related works within this area; Section 6 highlights the main findings of this work.

2 GTOP: MAPPING PROPERTY GRAPHS

The Property Graph model presents powerful semantics for information representation, with the application of declarative query languages, such as OpenCypher and Gremlin, becoming increasingly popular amongst the community. Our motivation with gTop was to propose a flexible way to decouple these Property Graph concepts from domain-specific Online Transaction Processing (OLTP) systems, providing the community with a wider range of storage options when graph queries are to be supported.

A *gTop* file is composed of two core sections: an *Abstract* Property Graph model and a mapping to a specific storage *implementation*. The *abstract* section specifies the inherent structure of the Property Graph, defining the available vertex and edge types, together with the properties they contain. Graph entity instances of the same type will then share matching property keys, but may have different corresponding values. This is comparable to the relational schema popular within the RDBMS domain. A Property Graph model example can be seen in Figure 1. Figure 2 takes one relation from this to demonstrate how it would be mapped in a gTop file. The top layer of Figure 2 contains a node of type Person connected

[1] https://github.com/cytosm/cytosm

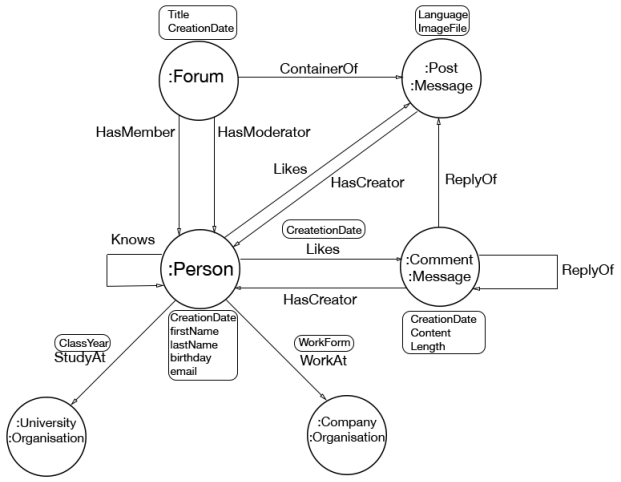

Figure 1: Abstract *gTop* model representing a subset of the Linked Data Benchmark Council (LDBC) Social Network data-schema. Edge and node properties are inside boxes.

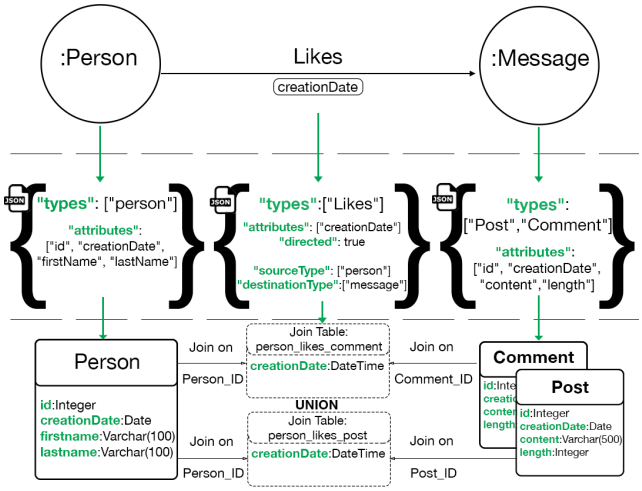

Figure 2: *gTop* modelling from LDBC data.

through an outgoing edge of type Likes to a node of type Message. This is serialised into a *gTop* Abstract (stored in JSON) containing high level information such as entity type and attributes.

The *gTop* implementation section then defines how these abstract entities map to a specific storage technology, such as a traditional relational database. For a relational backend, nodes will be mapped to rows within tables, whilst edges may be represented as either fields within these rows or as a sequence of table-join operations. This is demonstrated in the bottom layer of Figure 2. Here node type *Person* is mapped 1:1 with a database table of the same name. The *Message* type is actually two tables in the underlying database (*Post* and *Comment*) which would be joined via a union. Finally, the edge *Likes* is also split between two tables (*Person_Likes_Post* and *Person_Likes_Comment*). These are, however, not permanent tables, but instead join tables connecting tuples in *Person* to those in *Post/Comment* based on the foreign keys they contain.

gTop was designed to support automatic generation via database analysis techniques, such as DDL analysis for normalised tables (similar to [1, 14]), or more refined functional dependency analysis based on FDEP [3]. These techniques have been utilised to create *gTop* files for a variety of datasets, including Microsoft's Northwind[9] and the full LDBC-SNB graph[5]. Cytosm's documentation contains full details on the generation/mapping process, along with these example *gTop* files[2].

3 GRAPH QUERY CONVERSION

Pattern-based GQL's, such as OpenCypher and PGQL, try to find patterns in graph data, matching the inserted query to Regular Path Queries (RPQ's) [13]. As OpenCypher is schemaless, the usual approach in exploring these paths is brute force, querying all possible node/edge combinations within the RPQ's and filtering the results at the end. In this section we present how *gTop* models can be exploited to massively reduce the required search space and efficiently execute OpenCypher Queries (OCQ's) on any arbitrary database. This process consists of two modules: *Pathfinder* and the *OpenCypher to SQL* converter.

The split between *Pathfinder* and *OpenCypher to SQL* coincides with the two sections found within gTop. Firstly, *Pathfinder* takes the original OCQ and uses the *gTop's* abstract section to obtain a set of restricted OpenCypher Queries (rOCQ's). These rOCQ's contain no multi-hop edges or anonymous entities (as these cannot be mapped to SQL) and each specifies an explicit path from the original OCQ. *OpenCypher to SQL* then takes this set and uses the *gTop's* implementation section to map each rOCQ to an equivalent SQL query, which may be run on the underlying non-graph database. All the results are then unionised to return the information requested by the original OCQ.

3.1 Pathfinder

Enabling RPQ's for pattern-matching provides powerful expressiveness for application users. These do, however, require translating into paths contained within the graph data. *Pathfinder* is a query 'preplanner' which utilises the *gTop* abstract model to efficiently prune the search space from erroneous combinations and reduce unnecessary storage fetches.

Listing 1 shows an example OpenCypher Query. On line 1 and 2 the content after the 'Match' keywords are OpenCypher patterns, expressed via an RPQ. An OpenCypher pattern is built from a sequence of individual node and edge patterns, which in turn consist of labels and attribute keys representing at least one entity type defined in the *gTop* abstract model. Edge patterns can also contain additional complexity, defining a range of hops between node patterns and the direction of the edge.

```
1  Match (p: {"firstName": "John"})
2  Match (p) -[*1..2 ]-> (m:Message)
3  Return p, m.content;
```

Listing 1: OpenCypher Query example

[2]See https://github.com/cytosm/cytosm/blob/master/common/

Pathfinder takes this OCQ and the provided *gTop* abstract and inserts them into Algorithm 1. The first stage of this process is hint solving (lines 2-5). This examines node and edge patterns for information which can narrow the overall search scope. For example, the node referenced by variable *p* has the attribute "firstName". Taking the *gTop* presented in Figure 1 into consideration, this node must be of type Person[3].

Next, multi-hop edges are expanded in order to generate all possible paths (lines 6-9). As an example, the edge in Listing 1 between *p* and *m* contains an asterisk followed by '1..2'. This means the destination node can be either one or two nodes away from the starting point (with any node or edge type in-between). This step would, therefore, generate two paths which can be seen in Listing 2 below. A set is used to store these paths as duplicates are often generated.

```
1  (p:Person) ---> (m:Message)
2  (p:Person) ---> () --> (m:Message)
```

Listing 2: RPQ's generated from edge expansion

The final step is to 'solve' each path by filling in any anonymous edges and nodes. This is done by finding all the *gTop* paths that fit the RPQ pattern (lines 10-13). Each pattern is solved in parallel by running a Depth-First Search[4] on the gTop, starting from a typed node. *Pathfinder* then converts the returned set of explicit graph paths into rOCQ's (line 14) and passes them to *OpenCypher to SQL*. For example, RPQ 1 in Listing 2 has one possible edge type (*Likes*), with node type *Message* refering to either *Post* or *Comment*; this RPQ is, therefore, expanded into two explicit paths (Line 1 and 2 in Listing 3). Due to the number of anonymous entities, RPQ 2 is less straightforward, generating many non-intuitive paths such as Line 3 in Listing 3.

```
1  (p:Person)-[:Likes]->(m:Post)
2  (p:Person)-[:Likes]->(m:Comment)
3  (p:Person)-[:Knows]->(:Person)-[:Likes]->(m:Post)
```

Listing 3: Subset of explicit graph paths

3.2 OpenCypher To SQL

The set of rOCQ's created by *Pathfinder* is passed to the *OpenCypher to SQL* converter to translate and execute on the underlying database. Each rOCQ can be processed independently (enabling parallelism), through a four stages process. These stages are illustrated below via the example rOCQ in Listing 4, derived from the OCQ discussed in the previous section.

```
1  Match (p:Person { "firstName":"John" })
2  Match (p:Person) -[:Likes]-> (m:Post)
3  Return p, m.content;
```

Listing 4: rOCQ built from explicit path

Building a Basic AST Tree

When a rOCQ is initially parsed, an Abstract Syntax Tree (AST) is generated. In the first pass over the query, this is analyzed using the visitor pattern in order to build an intermediate SQL representation tree. This tree is made of items that resemble SQL statements and is later used to render the final SQL query. In this form, OpenCypher information can be represented as an outer SELECT containing multiple nested "WITH SELECT" items, one for each MATCH. This can be seen in the top layer of Figure 3 where the variables returned by Listing 4 are contained in the Outer SELECT, and each MATCH is converted into an internal WITH SELECT clause.

Algorithm 1: Pathfinder

Input : *ocq*, an OpenCypher Query
gTop, a gTop file
Output : *rocqs*, set of restricted OpenCypher queries matching the *gTop*

```
1  rocqs ⟵ ∅;
2  hints ⟵ ocq.scanForHints(gTop);
3  foreach hint h ∈ hints do
4  |  ocq ⟵ ocq.replaceHint(h);
5  end
6  paths ⟵ ∅;
7  foreach rpq r ∈ ocq do
8  |  paths.add(rpq.expandMultiHopEdges())
9  end
10 erpqs ⟵ ∅;
11 foreach path p ∈ paths do
12 |  erpqs.add(p.solve(gTop));
13 end
14 rocqs ⟵ erpqs.buildRocqs();
```

Variable Scope and Dependencies

On a second pass through the intermediate SQL tree, variable scope is analysed to discover which variables require information from the database, and if there are any dependencies between the SELECT items. For Listing 4 this would discover that variable *p* and *m* refer to information in the database, and that variable *p* in the second WITH SELECT clause is defined in the first; thus creating a dependency between them. Furthermore, the Outer SELECT will be dependent on all the clauses it contains as it returns data from them.

Mapping Nodes to RDBMS

The third pass utilises the *gTop's* implementation layer to map nodes to the correct underlying database table. For example, *p* may be mapped to a table storing information on people, as in Figure 2. However, it is important to reiterate that an entity type may not always have a 1:1 mapping. For instance, in a heavily normalised database, a persons email and phone number may be stored in separate tables. This can be seen in the Third layer of Figure 3 where SELECT(*p*) has been converted into UNION(*p*); containing inner SELECTS to return data from all tables involved.

Mapping Traversals

Finally, once all nodes are fully mapped, the last pass resolves the graph edges. The implementation layer of *gTop* provides table and column JOIN information in order to traverse from one node type to another; allowing new JOIN items to be inserted into the tree.

[3]Hints are not always this clean cut and can return multiple possible entity types, especially for common attributes such as 'id'. In this case the different possibilities are recorded and used in the final solving stage as alternative starting points

[4]Since a *gTop* abstract model contains the graph topology rather than the graph data itself, there is usually only a small number of node and edge types, even when the graph data it models contains billions of node and edge instances.

This can be seen in the final layer of Figure 3, where the tables are joined together to form the required *Likes* edge. Once this process is finished, the intermediate SQL tree is ready to be rendered into SQL statements. These can then be executed on the database, and the results from all rOCQ's can be joined together and returned to the user.

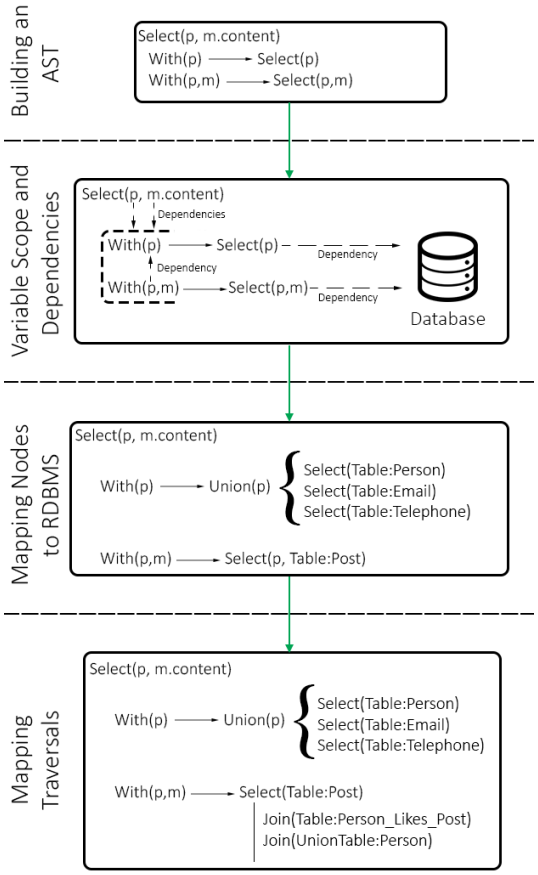

Figure 3: *OpenCypher to SQL conversion steps*

4 EXPERIMENTATION

In this section we show how the gTop model and query planning contributions perform on a variety of Property Graph Queries using data generated from the Linked Data Benchmark Council (LDBC) [5]. LDBC is a EU project to define industry-strength benchmarks for graph systems. LDBC version 0.2.2 includes a standard set of queries for each benchmarked platform, implemented by the respective vendors. Aiming on reproducibility of results, a subset of the provided Cypher (Neo4j) queries were utilised. Furthermore, we examined how well Cytosm scales, testing the framework on 1, 3, 10, 30 and 100 Gigabyte datasets.

4.1 Evaluation of Pathfinder

Our first experiment was to independently evaluate the effectiveness of *Pathfinder*. To do this we took several LDBC queries and executed them on Neo4j v2.3.4/3.0.7 before and after *Pathfinder* processing. As *Pathfinder* is a preprocessing step, the transformed queries could be run on the same Neo4j engine as their vanilla equivalent, thus allowing comparison via Neo4j's 'database hits' metric, defined as an abstract unit of storage engine work.

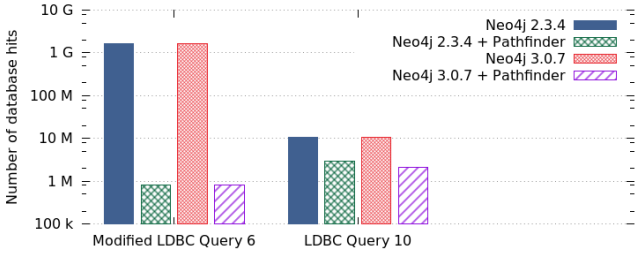

Figure 4: Database hits (on log scale) on Neo4j with and without the *Pathfinder* module.

Figure 4 plots the results for LDBC queries 6 and 10. Note that Query 6 was slightly modified for this test, increasing the range of hops in the *Knows* edge from two to four. This was done to see how well the system performed on longer multi-hop anonymous queries. Without *Pathfinder* preprocessing the vanilla Neo4j data engine first retrieved all stored nodes and then sequentially performed *expand* and *filter* operations. Where expansion explores all connected edges to a node and filter removes those not relevant to the query. When *Pathfinder* was utilised, the query plan appeared to have much higher parallelism[5] in addition to a massive reduction in database hits. This translated directly into faster retrieval times and less stress on the database server. These results show that the use of a schema (*gTop* abstract), together with a preplanner (*Pathfinder*) can also benefit native graph storage systems.

4.2 Evaluation of Cytosm Mapping

The second experiment aimed to evaluate the efficiency of Cytosm running on top of a relational database. This was done by mapping the LDBC dataset to Vertica and executing queries transformed via Cytosm. The results were then compared to SQLGraph [12], as well as a vanilla Neo4j v2.3.4 implementation (i.e. without *pathfinder*). LDBC queries 2 and 6 (modified) were used for this test as they cover two main query use cases: *Data-Fetch* (simple relational algebra operations, such as sorting or selections) and *Graph Traversal*.

To assess SQLGraph and Neo4j v2.3.4, the proposed data schema format was implemented on top of MySQL v5.8. Both systems could then be tested on the same physical machine (containing 8 Intel Xeon 4870E7 processors and 4TB of RAM). Unfortunately Vertica can only run in a distributed setting, so was tested on a cluster of three identical servers connected over a 1Gb link. Each server had 2 Intel Xeon X5650 processors and 96GB RAM. All systems were run on warm cache configuration, with the exception of MySQL, which had the query caching disabled. For the query experiments, 4 samples were taken after cache warmup. Figure 5 compares the performance of these systems on Data-Fetch queries (Query 2). By using projections, Vertica has the advantage over other databases,

[5]Neo4j provides a visualisation of its query plan structure; full breakdown of the plotted queries can be found at: https://github.com/cytosm/cytosm/tree/master/pathfinder

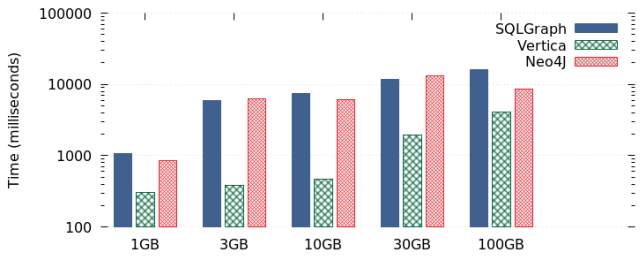

Figure 5: Query 2 (Data-fetch) time vs LDBC dataset size. 90*th* percentiles are given based on 4 samples after cache warm-up.

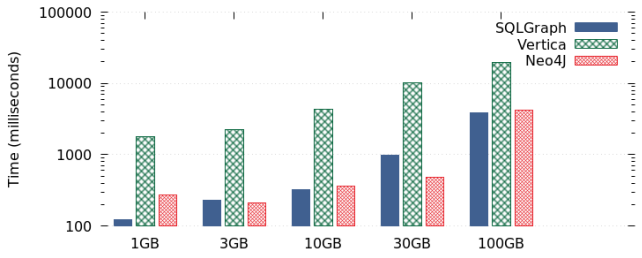

Figure 6: Query 6 (Graph Traversal) time vs LDBC dataset size. 90*th* percentiles are given, based on 4 samples after cache warm-up.

since it can optimise the data-layout to reduce the amount of intermediate results sent over the network and improve the degree of parallelism on the computation. Although the performance of Neo4j seems slower on 3GB than 10GB, the difference is in the same order of magnitude. The same is true for 30GB and 100GB[6].

Neo4j has the smallest query time on the Graph Traversal operations (Query 6) shown in Figure 6. Neo4j's query planner uses efficient space pruning, reducing the problem down to a subset of the graph and thus fetching less data. It also splits the query plan into several parallel branches. SQLGraph, on the other hand, is not optimised for graphs with many edge and node types. The schema requires searching for subtypes on a table during the traversal, and the query planner is not aware of the graph structure underneath it.

5 RELATED WORK

Unipop [10] maps Gremlin queries on top of RDBMS and NoSQL stores. The project defines a model (ontology) for representing Property Graph information and mapping it to SQL systems. Ontologies provide similar expressivity to gTop models, and Tinkerpop compatible systems can map Gremlin commands to Unipop-enabled components. The similarity between both models show our proposed query resolution contributions, such as *Pathfinder*, could be applied to Gremlin through the Unipop stack.

Stardog [11] provides similar graph topology mapping features for RDF graphs. The mapping syntax can be used to convert RDBMS into a RDF representation, which is Stardog's native format for all vertices, edges and properties. Gaffer [8] provides a graph schema to specify edge/vertex properties in a JSON format, with persistence in an Accumulo store. Unlike gTop, the intermediate schema does not support edge and node typing (e.g. specifying that nodes in a given format are of type Person) or composed edges (with data coming from multiple tables in order to represent a single edge). Simpler approaches for mapping have also been employed. RapidGrapher [7] requests the user to manually enter SQL queries in order to map nodes and edges in a RDBMS. GraphX [4] requires manual mapping during ETL. [6] uses undirected graphs for their mapping.

Grail [2] defines a SQL schema which supports both storing graphs and executing full graph processing through a vertex-centric processing computation model, such as Pagerank, or Weakly Connected Components. The schema includes both permanent tables for graph information and intermediate tables representing computation state. Grail presents multiple optimisations that show the potential of a traditional SQL backend to support graph storage and analysis on a single node.

SQLGraph [12] proposes a relational schema which stores Gremlin Property Graph information in a relational structure designed for executing queries efficiently. Each Gremlin query pipe is translated to a SQL instruction, combining them all into a single SQL database query. SQLGraph schemas can be adopted as a gTop implementation, although this would require the data to be modelled as a Property Graph, rather than interpreting an existing SQL database as a graph.

6 CONCLUSION

In this paper we address the challenges of running declarative Property Graph queries on non-domain specific backends without performing ETL operations. Our solution is split into three contributions: (i) A mapping format, *gTop*, which enables the translation from the Property Graph domain to alternative storage; (ii) a parallel RPQ solver which uses graph topology information whilst searching for paths; and (iii) a module that translates OpenCypher queries to SQL instructions.

Our evaluation confirms that relational systems can still produce comparable performance against domain-specific technologies, depending heavily on the type of graph query. Furthermore, we show that providing schema-like information (via a mapping such as *gTop*) to a query planner can greatly influence system performance for both the proposed use case, and even current state of the art graph databases.

As a continuation to this work, we plan to investigate methods for inferring graph topologies from existing relational and NoSQL data stores, combining structural relational analysis and automated data classification techniques. This way, Cytosm can enable the use of property graph queries to explore the inherent relationships from non graph datasets.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. The work is supported by the Engineering and Physical Sciences Research Council (EPSRC) and Hewlett Packard Enterprise (HPE).

[6]The documentation for the full benchmarking process, along with the results for all LDBC queries can be found at https://github.com/Alnaimi-/database-benchmark

REFERENCES

[1] R. De Virgilio, A. Maccioni, and R. Torlone. Converting relational to graph databases. In *GRADES (2013)*.
[2] J. Fan, A. G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR (2015)*.
[3] P. A. Flach and I. Savnik. Database dependency discovery: A machine learning approach. *AI Commun. (1999)*.
[4] J. E. Gonzalez et al. Graphx: Graph processing in a distributed dataflow framework. In *OSDI (2014)*.
[5] A. Iosup et al. Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *VLDB (2016)*.
[6] U. Khurana, S. Parthasarathy, and D. Turaga. Graph-based exploration of non-graph datasets. *VLDB (2016)*.
[7] Graph Base. RapidGrapher for Agility. http://graphbase.net/GBaseRapidGrapher.htm.
[8] GCHQ: Gaffer. https://github.com/gchq/Gaffer.
[9] Microsoft. https://northwinddatabase.codeplex.com/.
[10] Unipop: Data Integration Graph. https://github.com/unipop-graph/unipop.
[11] Stardog 4: The Manual. http://docs.stardog.com/.
[12] W. Sun et al. Sqlgraph: An efficient relational-based property graph store. In *SIGMOD (2015)*.
[13] O. van Rest et al. Pgql: a property graph query language. In *GRADES (2016)*.
[14] R. D. Virgilio, A. Maccioni, and R. Torlone. R2G: a tool for migrating relations to graphs. In *EDBT (2014)*.