GraphGen: Adaptive Graph Processing using Relational Databases

Konstantinos Xirogiannopoulos
University of Maryland, College Park
kostasx@cs.umd.edu

Virinchi Srinivas
University of Maryland, College Park
virinchi@cs.umd.edu

Amol Deshpande
University of Maryland, College Park
amol@cs.umd.edu

ABSTRACT

Graph querying and analytics are becoming an increasingly important component of the arsenal of tools for extracting different kinds of insights from data. Despite an immense amount of work on those topics, graphs are largely still handled in an ad hoc manner, in part because most data continues to reside in relational-like data management systems, and because graph analytics/querying typically forms a small portion of the overall analysis pipelines. In this paper we describe an end-to-end graph analysis framework, called GraphGen, that sits atop an RDBMS, and supports graph querying/analytics through: (a) defining graphs as transformations over underlying relational datasets (as *Graph-Views*) and (b) specifying queries or analytics on those graphs using either a high-level language or Java programs against a simple graph API. Although conceptually simple, GraphGen acts as an abstraction/independence layer that opens up many opportunities for adaptively optimizing graph analysis workflows, since the system can decide where to execute tasks on a per-task basis (in database or outside), how much of the graph to *materialize* in memory, and what types of in-memory representations to use (especially critical when the graphs are larger than the input datasets, as is often the case). At the same time, by providing the ability to write arbitrary programs against the graphs, GraphGen removes a major expressivity limitation of many existing graph analysis systems, which only support limited programming frameworks. We describe the GraphGen DSL, loosely based on Datalog, that includes both graph specification and in-line analysis capabilities. We then discuss many optimization challenges in building GraphGen, that we are currently working on addressing.

ACM Reference format:
Konstantinos Xirogiannopoulos, Virinchi Srinivas, and Amol Deshpande. 2017. GraphGen: Adaptive Graph Processing using Relational Databases. In *Proceedings of GRADES'17, Chicago, IL, USA, May 19, 2017,* 7 pages.
DOI: http://dx.doi.org/10.1145/3078447.3078456

1 INTRODUCTION

Analyzing the interconnection structure, i.e., *graph structure*, among the underlying entities or objects in a dataset can provide significant insights and value in many application domains such as social media, finance, health, and the sciences. This has led to an increasing interest in executing a wide variety of graph analysis tasks and graph algorithms, e.g., community detection, influence propagation, network evolution, anomaly detection, centrality analysis, etc. Many specialized graph databases (e.g., Neo4j, Titan, OrientDB), and graph execution engines (e.g., Giraph, GraphLab, Ligra, Galois, GraphX) have been developed in recent years to address these needs. Although such specialized graph data management systems have made significant advances in storing and analyzing graph-structured data, a large fraction of the data of interest initially resides in relational database systems (or similar structured storage systems like key-value stores, with some sort of schema); this will likely continue to be the case for a variety of reasons including the maturity of RDBMSs, their support for transactions and SQL queries, and to some degree, inertia. Relational databases typically include many useful relationships between entities, and can contain many hidden, interesting graphs, and thus it often makes sense to extract and analyze those graphs. At the same time, given their maturity, relational databases are also often used as a backend to store graph data.

There has thus been much work at the boundary of graph and relational databases, which is often hard to reconcile since the underlying environmental assumptions and the query workloads tend to be quite different. We also often see confusion about what we may call "data independence issues" where implementation and abstraction are not clearly separated. As one may surmise, different design points are better for different types of data and different types of query workloads; unfortunately such conclusions are often missing from much of the work on this topic.

In this paper, we propose an end-to-end graph analysis framework, called GraphGen, that subsumes the different design points where relational or graph data models or engines are combined. GraphGen is intended as a layer on top of an extant relational database, and although it can simulate the different design points, it does not, as of now, offer solutions to all of the optimization challenges that arise in the process. GraphGen considers graph analytics or querying as a combination of: (1) specifying graphs of interest against the data in the underlying database as *GraphViews*, and (2) specifying an analysis task or a query (possibly at a later time) against those graphs. We describe a unified language, loosely based on Datalog, to both define such VIEWs (GraphGenDL) and to write queries or analytics against those VIEWs (GraphGenQL). GraphGenDL supports defining a *collection* of graphs, which enables rich functionality like temporal analytics, ego-centric analysis, or analysis of induced graphs satisfying different properties. GraphGen also supports writing arbitrary graph algorithms against the defined graphs as Java programs. GraphGen is "adaptive" in the sense that it can automatically make different choices regarding the optimal way to both define a Graph View *and* execute some analysis/query over that graph, depending on a variety of factors including the query itself and the properties of the defined graphs.

GRADES'17, Chicago, IL, USA
© 2017 ACM. 978-1-4503-5038-9/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3078447.3078456

2 PRIOR WORK

Graph+Relational Design Points: Including the work on building XML and RDF databases, there has been much work at the boundary of graph and relational-like databases which can roughly be classified into four categories.

1. Graph Frontend, Graph Backend: There has been much work on building native XML and RDF databases, but somewhat less so on *Property Graph* databases, with Neo4j and OrientDB being the primary examples of the last. These systems typically start from scratch to focus on graph queries and workloads, and use specialized representations and indexes towards their optimization. In addition to supporting query languages like SPARQL, Cypher or PGQL that are similar to SQL, some of these systems also support graph APIs to directly operate on the stored graphs, which is often a necessity for complex graph analytics. Using such a database of course requires a complete buy-in, which is often not an option since most enterprises also need to support non-graph workloads. These systems are also not as mature or scalable as RDBMSs/key-value stores, and often perform poorly on queries common to both.

2. Graph Frontend, Relational Backend: A common design for graph databases is to use a thin layer on top of an RDBMS that "shreds" graph data into the underlying database (Figure 1(i)) , and converts graph queries into queries against it (possibly with some post-processing). Shanmugasundaram et al. [12] did the early work exploring this option for XML databases, and there has been much work on building RDF databases in this fashion. More recently, Sun et al. proposed SQLGraph [15], which supports the Property Graph model on top of an RDBMS. The Titan graph database is also similar since it uses a key-value store at the backend to store the data. A major challenge for these systems is designing good schemas (and appropriate indexes) for storing the data in the underlying database, since that dictates the performance to a large degree. Vertexica [5] and Grail [2] use a similar design but focus primarily on supporting batch analytics over very large graphs.

3. Graph+Relational Frontend, Relational Backend: The above option can be seen as starting with the graph schema and figuring out an appropriate relational schema to use for the underlying database; and although it is technically possible to query the underlying database system (using SQL), the schema of that database is usually not meaningful. However, in most cases, enterprises typically have existing relational databases (and schemas) and often the goal is to analyze the graphs that are hidden within them. Analogous to the above case, such functionality can be supported by a thin layer on top, that again supports a graph data model on top of a relational database; however, unlike the above option, the layer is largely *independent*, does not have any freedom to rearrange the schema or even the ability to build new indexes. The layer also has to contend with updates happening to the underlying database through other interfaces (Figure 1(ii)).

Another major challenge is that the *mapping* between the graph nodes/edges and the relational tables may not be straightforward. For example, consider the familiar TPC-H schema (see Appendix A). We may be interested in analyzing the *bi-partite* graph between customers and products, where a customer is connected to a product if they bought it. Here, the *nodes* in the graph correspond directly to the customers and the products (i.e., there is a *direct* mapping), but

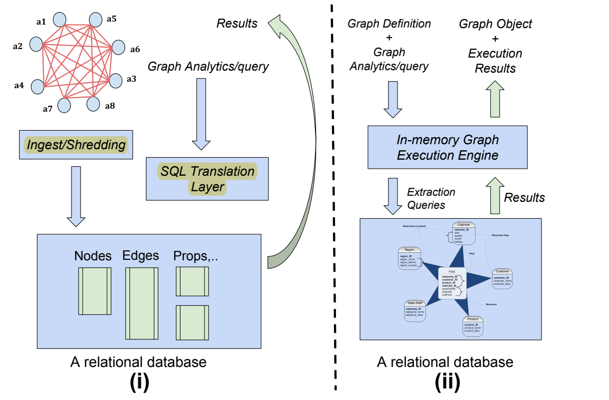


Figure 1: While similar in many ways, Approaches 2 (left) & 3 (right) make fundamentally different assumptions

the *edges* in the graph require us to do a join between the *LineItem* and *Orders* tables, followed by a `DISTINCT` to remove duplicate connections. The more troublesome mappings are ones that require us to do a self-join. For example, consider a DBLP-like database with three tables: Authors, Publications, and AuthorPublication (see Appendix A). A natural graph to analyze here is the `CoAuthors` graph, where two authors are connected if they have had a publication together. Creating the edges here requires doing a self-join on the AuthorPublication table followed by a `DISTINCT`. A key challenge here is that the size of the self-join is often much larger than the size of the original AuthorPublication table [16].

Although this approach (Approach 3) is likely to be the most attractive in practice, there hasn't been much systematic work on understanding it; one of our goals here is to systematically explore it and discuss the key challenges that come up. Aster Graph Analytics [14] and SAP HANA Graph Engine support specifying graphs within an SQL query as transformations on underlying relational tables, and applying graph algorithms on those graphs. However, the interface for specifying which graphs to extract is not very intuitive and limits the types of graphs that can be extracted, while Aster only supports the vertex-centric API for writing graph algorithms. Ringo [9] also provides operators for converting from an in-memory relational table representation to a graph representation, but is not intended as a layer on top of an RDBMS. Finally, in our prior work on GraphGen, we focused on the problem of exploring hidden graphs within relational databases [17] and dealing with large intermediate results that get generated in the process [16].

4. Relational Backend, Relational Frontend, Graph Engine: Here the data is resident in a relational database, and a layer is built on top that utilizes a graph engine to process SQL queries efficiently [6, 7]. If the abstraction layer is still Relational/SQL, this is more akin to developing better SQL query processing and optimization algorithms, which are better suited for specific types of workloads. We consider this line of work somewhat orthogonal and discuss it here primarily for completeness, since those techniques could be implemented in other (existing) query processing engines and could be invoked as needed for specific types of queries.

Views: Our proposed approach is fundamentally based on looking at graphs as VIEWs over the underlying relational tables, and thus the work on incremental view maintenance and view materialization is closely related [1]. Similar to that work, we need to make

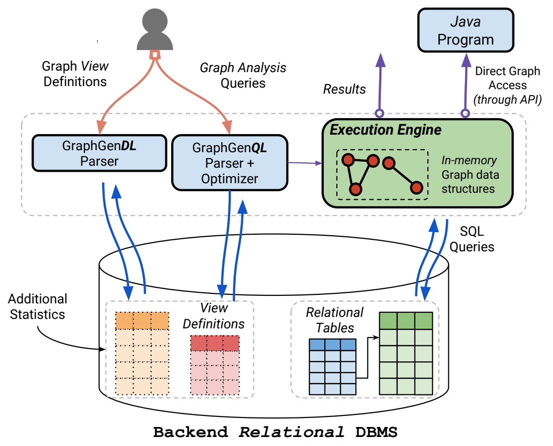


Figure 2: High-level Overview of GraphGen

decisions about: (a) converting operations over VIEWs into operations over underlying tables, (b) choosing VIEWs to materialize (the search space of possibilities is larger since we may want to materialize VIEWs in memory and on persistent storage), and (c) incrementally maintaining views. We plan to build upon the rich history of work in this space, focusing primarily on VIEWs involving self-joins, DISTINCT clauses, and aggregates as we saw in the examples above that are not as well-studied in the literature.

3 GRAPHGEN OVERVIEW

Figure 2 shows a high-level overview of our proposed GraphGen framework. The cornerstone of the system is an abstraction layer that sits atop an RDBMS, serving two main functions. First, it accepts GraphView definitions from the users, specified using a Datalog-like DSL called **GraphGenDL**, where the user specifies how to construct the nodes and the edges of the graph (in essence, as *named VIEWs* over the underlying tables). We chose to base our DSL on Datalog, which has been increasingly used for expressing data analytics workflows and graph analysis tasks [3, 4, 11], due to its elegance in naturally expressing recursive queries, but also in its overall intuitive and simple syntax choices. However, due to a natural need for features like defining a *collection* of graphs, our DSL differs significantly from standard Datalog beyond the basics.

Second, GraphGen accepts queries or analysis tasks against one or more of those GraphViews and executes them by either pushing the computation into the database, or extracting the requisite information from the database into memory, or a combination of the two. The queries are specified using a Datalog-like DSL, called GraphGenQL, which is based on subgraph pattern matching and is somewhat similar to SPARQL, Cypher, and PGQL. However, unlike those languages, GraphGenQL can also be used to write a limited set of analysis tasks. GraphGenQL supports the vertex-centric programming framework through UDFs, which mitigates that to some extent. We chose to design our own language primarily to maintain uniformity with GraphGenDL. Finally, analysis tasks can also be written using a Java API that allows direct access to the graph, thus allowing a user to write complex graph algorithms (e.g., community detection, bipartite matching, max-flow, etc.). This requires loading GraphGen as a library; we omit the details due to lack of space.

Regarding graph *updates*, we assume the data is typically updated through direct access to the database (i.e., using SQL); although it may be possible to update the underlying data through updates to the Graph-Views, we postpone supporting that feature given the semantic confusions it may introduce.

The abstractions supported by our system enable many different options regarding the implementation itself, which the system can choose from by analyzing the query and update workloads. In many cases, we may be able to push the entire graph query/analysis task into the database, which may be appropriate if the GraphViews are simple and the graph processing tasks are infrequent. However, for complex VIEWs involving self-joins, it may be better to load the data from the relations in memory and use optimized processing techniques that avoid executing the joins. We also have the option of materializing a graph (or portion of it) into memory if the rate of queries is higher than the rate of updates, with the downside being the need to do incremental view maintenance. For tasks written using the Java API, the graph must be loaded into memory, although many optimizations are possible there as we discuss later on.

GraphGen is designed as a centralized system, and although it can utilize multiple cores, we do not attempt to execute the tasks in a distributed fashion, primarily because of its goal to support complex graph analysis tasks that require random access to the graph [8]. We expect this to cover almost all the use cases for a system like this, since the vast majority of graphs are expected to fit in large multi-TB-memory machines that are easily available today. This is also the approach taken by Ringo [9] and most high-performance graph analysis systems (e.g., [13]). If required, GraphGen can output the graph into a format ingestible by distributed graph frameworks.

The GraphGen DSL naturally generates *directed* graphs, and undirected graphs are represented using *bidirectional* edges. The typical workflow for a user when writing an extraction query would be to initially inspect the database schema, figure out which relations are relevant to the graph(s) they are interested in exploring, and then choose which attributes in those relations would connect the defined entities in the desired way. We generally assume that the user is knowledgeable about the different entities and relationships existent in the database, and is able to formulate such queries. We have also built a visualization tool that allows users to discover and extract potential graphs in an interactive manner [17].

4 LANGUAGE DEFINITION

4.1 GraphGenDL: Specifying Graph Views

At a high level, specifying a graph requires specifying what constitutes the Nodes and the Edges of the graph. We begin with describing how a single graph definition is expressed in GraphGenDL, which is largely similar to standard Datalog. We then discuss how collections of graphs are specified succinctly.

Defining a Single Graph: The simplest graph to extract is a homogeneous graph, requiring at least one Nodes definition and one Edges definition, both specified as Datalog rules. The Nodes rule must have at least one "ID" attribute, to be used to uniquely identify nodes, and may have an arbitrary number of properties. The variable names in the *goal* of the rule are used as the labels of the respective properties (the first variable in the goal must be "ID").

```
CREATE GRAPHVIEW GV1 AS
   Nodes(ID,P1,P2,...) :- R(ID,P1,P2,...).
   Edges(ID1,ID2) :- E1(ID1, A),E2(A,B),...,En(X, ID2).
```

In the simplest case ("direct mapping"), the edges are already materialized in the database (i.e., the edges rule refers to only one database table); in general, however, we may have to do a sequence of joins in order to construct the edges. Edge properties can also be specified similarly to the node properties; however for non-direct mappings, those may require care since the computational cost of computing and materializing them may be high.

More generally, a user may use multiple Nodes rules and multiple Edges rules, to define more complex and heterogeneous graphs. The example below shows a graph specification over a `University` dataset (see Appendix A), containing three different types of nodes and two different types of edges. The second Edges rule defines a direct mapping to a relation, but the first Edges rule requires a non-key-foreign key join that may produce a large output.

```
CREATE GRAPHVIEW UnivGraph AS
  Nodes(I_ID, Name) :- Instructor(I_ID, Name).
  Nodes(S_ID, Name) :- Student(S_ID, Name).
  Nodes(SC_ID, Name, Address) :- School(SC_ID, Name, Address).
  Edges(I_ID, S_ID) :- TaughtCourse(I_ID, courseId), TookCourse(
        S_ID, courseId).
  Edges(I_ID,SC_ID) :- TaughtCourse(I_ID,_,SC_ID).
```

Properties may also be defined through aggregates over underlying attributes in the base relations. Since there is no standard syntax for aggregates in Datalog, we use the syntax advocated by Socialite [11]. The following annotates the edges of the *CoAuthors* graph with the number of times two authors have collaborated.

```
CREATE GRAPHVIEW CoAuthorsWeighted AS
  Nodes(ID, name) :- Author(ID, name).
  Edges(ID1, ID2, wt=$COUNT(pub)) :- AuthorPub(ID1, pub),
        AuthorPub(ID2, pub).
```

Specifying Multi-Graph Views: A key innovation of our DSL is the ability to support the common use case where a collection of different graphs need to be defined, indexed (identified) by some data-specific values. As we discuss later, abstracting this functionality at the language level enables many optimizations. First, we show how *ego networks*, corresponding to immediate connections of different entities in the database may be specified. Specifically, the example below creates a graph for each Author in a publications database, capturing the connections between its collaborators.

```
CREATE GRAPHVIEW AuthorEgoNetworks(X) WHERE Author(X) AS
      Nodes(X, name) :- Author(X, name).
      Nodes(ID, name) :- AuthorPub(X,pub), AuthorPub(ID,pub),
            Author(ID, name).
      Edges(ID1, ID2) :- AuthorPub(ID1, pub), AuthorPub(ID2, pub).
```

Intuitively, this statement specifies a graph for every possible X that matches the WHERE clause; for any specific X, the two Nodes clauses construct nodes for X and all of its collaborators, whereas the Edges clause only adds the appropriate edges required to construct the ego network. Although all of these ego networks are subgraphs of the overall CoAuthors graph defined above, if the query workload requires analyzing or querying only one or a small number of them, the ability to refer to them independently can result in orders of magnitude savings.

We can similarly define 2-hop neighborhoods around the Authors, a different CoAuthors graph for every "field" (assuming appropriate variables are present in the schema), or a different CoAuthors graph for each different conference simultaneously.

Another major use case for this functionality is *temporal analytics*, where a user may want to analyze the evolution of a graph over time, or perform queries over historical snapshots. The below shows an example where we specify temporal snapshots of the CoAuthors graph at the end of every year, assuming the Publication table contains the year of publication as the third attribute.

```
CREATE GRAPHVIEW CoAuthorsSnapshot(X)
WHERE X IN RANGE(1950, 2017, 1)
    Nodes(ID,name) :- Author(ID,name).
    Edges(ID1,ID2) :- AuthorPub(ID1, pub), AuthorPub(ID2, pub),
          Publication(pub, _, Y), Y <= X.
```

4.2 GraphGenQL: Querying Graph Views

Once the graphs are specified as VIEWs over the relational database, a user may issue different types of queries against one or more of those graphs. We plan to primarily support a subgraph pattern matching-based query language, similar to SPARQL etc., incorporating a Datalog syntax for naturally expressing graph traversal through recursive computation over the "Edges" VIEW. As an example, assuming the author table also has information about each author's field of study (as a third attribute), the query specified below finds all triangles in the CoAuthors graph where the three authors are in ML, Databases, and Algorithms respectively.

```
USING GRAPHVIEW CoAuthors
Triangle(X, Y, Z) :- Nodes(X, _, "ML"),Nodes(Y, _, "DB"),
         Nodes(Z, _, "AL"),Edges(X, Y),Edges(Y, Z),Edges(X, Z).
```

4.3 Specifying Analysis Tasks

There has been an intense amount of work on programming frameworks for specifying batch analysis tasks against large graphs. Although the vertex-centric framework is the most popular, it has fairly limited expressive power [8, 10] and cannot be used to easily write complex graph algorithms. We envision supporting several different ways to write analysis tasks against the graphs.

Using Datalog-based DSL: We plan to extend GraphGenQL by building upon the language proposed in Socialite [11], which also uses a Datalog-based DSL to specify analysis tasks like PageRank, Connected Components, and others. We omit further details for brevity but instead refer the reader to those papers.

Using a Java Program: To write arbitrary graph algorithms, GraphGen can be loaded as a library within a Java program. GraphGen provides functions for the program to create GraphViews or load existing GraphViews. The program then has access to the nodes and the edges of their specified graph through a GraphView object via a simple API that allows iterating through the vertices or the edges of a vertex. GraphGen also provides a vertex-centric programming model on top of this API to simplify writing analysis tasks. We refer the reader to [16] for further details.

Currently, GraphGen primarily supports the second option, and we are working on adding support for GraphGenQL through developing translation programs to convert those programs into SQL. We are also working on developing a new high-level programming framework to make it easy to specify analytics over a collection of graphs in a declarative fashion.

5 CHALLENGES AND OPPORTUNITIES

The framework we propose serves to unify different ways RDBMSs and graph querying/analytics have been combined together, but it also raises many computational challenges and research opportunities. In this section, we discuss some of these opportunities for optimization after briefly describing our experimental setup.

Experimental Setup: We set up four different relational databases, PostgreSQL, MySQL, and two commercial relational database systems (DBS1, and DBS2), and loaded several different datasets into those. Here we report numbers primarily for the *DBLP* dataset and the CoAuthors graph on that dataset; we show results for two subsets of the dataset, a *small* dataset with 100,000 tuples where the CoAuthors graph has 1,639 nodes and 55,436 edges, and a *large* one with 500,000 tuples where the CoAuthors graph has 15,741 nodes and 529,434 edges. We use these simple subsets of DBLP to showcase the potential for optimization in adaptively handling graph computation in our framework by trying various ways of re-writing the queries when pushing them into the database, and contrasting this with executing these queries in-memory. Graph processing can be done orders of magnitude faster on a graph structure versus using relational processing due to the fact that the neighbor-lists are immediately available for every vertex. In contrast, traversing a graph by continuously processing an "Edges" VIEW in the database can also have benefits in some situations where we can avoid materializing the "Edges" VIEW; these benefits start to become apparent even for small datasets like the ones we experiment with.

Where to Execute Queries/Tasks: The first major question is *where* should the *execution* take place, in GraphGen or in the database itself? As one may expect, this is entirely dependent on the workload, both the rate of updates to the underlying tables and the rate of queries themselves. Table 1 and Figure 3 show the results for two queries, *triangle matching* (i.e., finding all occurrences of a triangle where all three authors work in "ML"), and *triangle counting* (i.e., counting the number of triangles in the graph) for a variety of different options. Contrasting Table 1 with Figure 3, one can see the differences between execution over an in-memory graph versus in-database. While GraphGen in-memory execution is usually much faster, after accounting for the cost of extraction, it may be better to simply let the database handle certain pattern matching queries and avoid the extraction effort.

A key challenge, thus, is to develop accurate cost models, effective workload monitoring tools, and optimization techniques to adaptively decide which part of the overall task to perform where. These decisions are closely tied with pre-computation/materialization decisions. For graphs that are frequently queried, it may be appropriate to materialize them, fully or partially, at the GraphGen layer to reduce query latencies. This, however, necessitates developing incremental view maintenance techniques, which may itself prove challenging for the more complex graph-views.

Another challenge here is optimizing execution over collections of graphs. For instance, for a temporal analytics task that wants to compare and analyze all or a subset of the CoAuthorsSnapshot graphs, it may be ideal to load them in an overlapped fashion in memory to reduce the memory footprint, and use incremental computation techniques to reduce the execution times.

By hiding all of these implementation decisions under high-level abstraction layers, GraphGen enables us to consider each of these optimizations in an adaptive fashion, unlike prior work where most such decisions are set in stone.

Query Rewriting: If the execution is to be pushed inside the database, there are often different ways to construct equivalent SQL queries. Although it is tempting to leave these decisions to the query optimizer, the auto-generated SQL can be verbose and may involve many blocks, making query optimization challenging. Figure 3 shows the results for the two example queries for *five* different ways of do so (MySQL does not support the `WITH` clause). First, we may create explicit VIEWs in the database, and write the analysis queries in terms of those. There are also two different ways we can filter out unnecessary tuples here (corresponding to duplicate generation of the same pair of authors); we can either insert a `DISTINCT` clause in the Edges VIEW or `WITH` statement, or simply remove the duplicates at the end of the computation (which may be a better idea if the query is duplicate-insensitive). Second, we may create temporary tables using the `WITH` construct (with both ways of filtering mentioned above). Such common table expressions (CTEs) are implemented differently in different systems, with some materializing the CTE as a temporary table, and others seeing it as a VIEW; further, some systems (e.g., PostgreSQL) do not optimize *across* CTEs. Lastly, we can execute the query on the base `AuthorPub` table directly, and compute all the appropriate joins as needed. As we can see in Figure 3, performance shows significant variations for the different rewrites, with no clear pattern. We also see huge variability in performance depending on where we choose to filter out the duplicate edges; removing duplicates earlier (in the statement where the edges are defined) is always better for these queries, however, the costly duplicate removal may be unnecessary if the query does not care about them.

For the triangle matching query, there are even more choices in terms of pushing down the selection (on area), which we do not consider here. We saw similar variability for several other queries, but we omit a detailed discussion due to lack of space.

Overall, rewriting and optimizing the queries that are naturally generated during graph analytics appears to expose gaps in today's query optimizers that need to be studied further.

Dataset	Triangle Counting	Triangle Pattern	ETL
small	0.169	0.001	2.049
large	6.723	0.015	17.52

Table 1: Times (sec) for running the two queries in memory. "ETL" is the time required for Extraction and Loading of the graph into memory from PostgreSQL

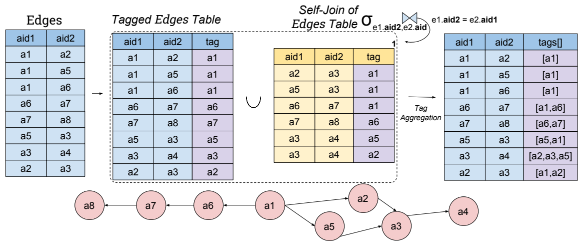
Triangle Counting (on *small*)

Query	DBS1	DBS2	MySQL	PostgreSQL
With (at edges)	**1**	**1.62**	NA	**13.8**
With (at the end)	53.028	2.99	NA	37.8
View (at edges)	1.054	2.07	**3.01**	15.6
View (at the end)	51.92	77.13	538.19	35.991
On Base Table	46.45	74.878	678.87	36.160

Triangle Pattern Matching where area = "ML" (on *large*)

Query	DBS1	DBS2	MySQL	PostgreSQL
With (at edges)	14.765		NA	0.749
With (at the end)	4.59		NA	**0.704**
View (at edges)	15.557		**4.26**	2.193
View (at the end)	**4.25**		20	11.063
On Base Table	8.612		22.69	3.089

Figure 3: Running times (sec) for two queries, rewritten to either use the "`WITH`" clause or a VIEW. The parentheses next to the method used specify at which computation the initial filtering of tuples was done with `DISTINCT`


Edges
Tagged Edges Table
Self-Join of Edges Table $\sigma_{e1.aid2,e2.aid}$
$\bowtie$ e1.aid2 = e2.aid1
1
$\cup$
Tag Aggregation
a8
a7
a6
a1
a2
a5
a3
a4

Edges

aid1	aid2
a1	a2
a1	a5
a1	a6
a6	a7
a7	a8
a5	a3
a3	a4
a2	a3

Tagged Edges Table

aid1	aid2	tag
a1	a2	a1
a1	a5	a1
a1	a6	a1
a6	a7	a6
a7	a8	a7
a5	a3	a5
a3	a4	a3
a2	a3	a2

aid1	aid2	tag
a2	a3	a1
a5	a3	a1
a6	a7	a1
a7	a8	a6
a3	a4	a5
a3	a4	a2

aid1	aid2	tags[]
a1	a2	[a1]
a1	a5	[a1]
a1	a6	[a1]
a6	a7	[a1,a6]
a7	a8	[a6,a7]
a5	a3	[a5,a1]
a3	a4	[a2,a3,a5]
a2	a3	[a1,a2]

Figure 4: Tagging towards the efficient specification of ego-graphs with a single query

Selectivity Estimation: We also observed that the estimated selectivities for the more complex generated queries are often quite inaccurate. Here we mainly experimented with PostgreSQL, where we found that the selectivity estimates of the join result sizes were on point for simple queries. As we generated more complex graphs that required several self-joins, the estimates started diverging by orders of magnitude. For instance, on the DBLP database, consider a query that creates a graph where authors are connected to each other if they've published in the *same conference*. This query requires a join between `AuthorPub` and `Publication`, and then a self-join of the result with itself. The number of rows estimated by PostgreSQL in this case was 40x higher than the actual number.

Optimizing the Extraction of Multi-Graph Views: The naive way to extract a collection of graphs is to generate *a separate SQL query* for each distinct graph. However, by abstracting the functionality at the language level, GraphGen can employ several different optimizations that result in significant performance improvements.

As an example, consider the AuthorEgoNetworks MultiGraphView above where the user specifies a collection of ego networks, one for each Author. Instead of issuing a separate query for each Author to generate its ego network, we can instead use a technique we call *result-tagging*, where we extract information about which ego networks each edge is part of, all at once. A depiction of this process is demonstrated in Figure 4. We start with tagging each tuple of the edge VIEW of the entire graph with its source id, v (these tuples are the immediate neighbors of v) – let that be VIEW A. A self-join is then computed over the *Edges VIEW* (this is for computing the neighbors of neighbors of v), where each of the tuples in the result is *tagged* with the source vertex of the source id in the *left hand side* of the join; let this be VIEW B. A union of $A \cup B$ is used, and the final result is finally aggregated by each edge, thus creating a list of tags, each one signifying a subgraph each edge belongs to. By continuously repeating the above process on the result of each subsequent self-join this technique works for obtaining the ego-graph x-hops away from each vertex.

Thus, a key challenge for us, is to develop a systematic approach to optimizing the extraction of, and execution against, such MultiGraphViews.

Dealing with Large-Output Joins: It is very natural to specify graphs that require doing a non-key-foreign-key join for creating the edges, resulting in a space blowup. A primary example of that is the self-join that is typical in the extraction of nearly any homogeneous graph (like the CoAuthor or UnivGraph defined above). By "large-output", we refer to joins where the cardinality of the join attribute is low compared to the input table sizes , resulting in a large number of tuples in the join. As we showed in our prior work [16], operating on a *condensed* representation of the graph in-memory may be ideal in such cases. This representation is built to efficiently load the defined graph into memory *without* explicitly generating all of the edges. This representation can be very useful when one does not want to analyze the entire graph but *portions* of it at a time. With a trade-off in latency, we can also analyze graphs that normally wouldn't fit in memory using this representation.

Developing similar condensed representations for more complex Graph-Views that avoid generation of any large intermediate results remains a rich area of future work. This is also closely tied with the work on *factorized representations of query results* [18] and *worst-case optimal joins*; we refer to [16] for a more in-depth discussion.

6 CONCLUSIONS

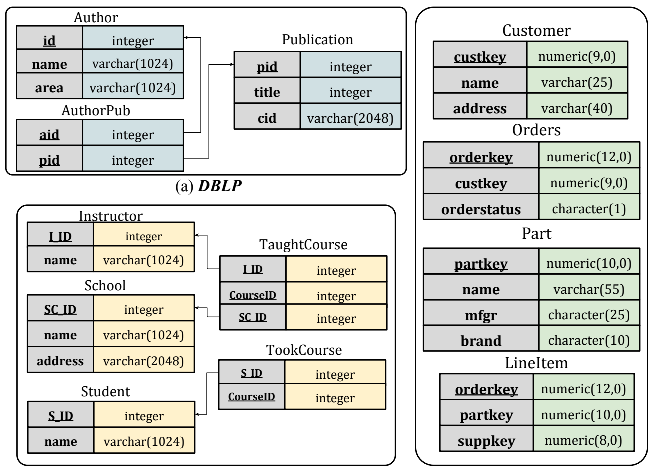
We presented a unified framework for extraction and analysis of graph-structured data stored in an RDBMS or similar structured storage engines. Apart from providing an intuitive means of specifying various graph structures within the relational data without compromising on the ability to write complex graph algorithms, our high-level abstractions enable a wide variety of adaptive optimizations for conducting these types of analyses. There are many interesting and difficult challenges here in terms of deciding where to execute graph queries/tasks, re-writing the SQL queries, making materialization decisions, and handling inaccuracies of the query optimizer and database statistics exposed by natural graph extraction and analysis tasks.

REFERENCES

[1] Rada Chirkova and Jun Yang. 2012. Materialized views. *Foundations and Trends in Databases* 4, 4 (2012), 295–405.
[2] J. Fan, G. Raj, and J. Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *CIDR*.
[3] Jun Gao, Jiashuai Zhou, Chang Zhou, and Jeffrey Xu Yu. 2014. GLog: A high level graph analysis system using MapReduce. In *ICDE*.
[4] Todd J Green, Molham Aref, and Grigoris Karvounarakis. 2012. *LogicBlox, platform and language: A tutorial*. Springer.
[5] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. 2014. VERTEXICA: Your Relational Friend for Graph Analytics! *PVLDB* (2014).
[6] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. 2016. Fast In-Memory SQL Analytics on Typed Graphs. *PVLDB* (2016).
[7] Hongbin Ma, Bin Shao, Yanghua Xiao, Liang Jeff Chen, and Haixun Wang. 2016. G-SQL: Fast Query Processing via Graph Exploration. *PVLDB* (2016).
[8] Frank McSherry, Michael Isard, and Derek G Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.
[9] Y. Perez, R. Sosič, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec. 2015. Ringo: Interactive Graph Analytics on Big-Memory Machines. In *SIGMOD*.
[10] Abdul Quamar, Amol Deshpande, and Jimmy Lin. 2016. NScale: neighborhood-centric large-scale graph analytics in the cloud. *VLDB Journal* (2016).
[11] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. 2013. Distributed socialite: a datalog-based language for large-scale graph analysis. *PVLDB* (2013).
[12] J. Shanmugasundaram et al. 1999. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*.
[13] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*.
[14] D. Simmen et al. 2014. Large-scale Graph Analytics in Aster 6: Bringing Context to Big Data Discovery. *PVLDB* (2014).
[15] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. 2015. SQLGraph: an efficient relational-based property graph store. In *SIGMOD*.
[16] Konstantinos Xirogiannopoulos and Amol Deshpande. 2017. Extracting and Analyzing Hidden Graphs from Relational Databases. In *SIGMOD*.
[17] K. Xirogiannopoulos, U. Khurana, and A. Deshpande. 2015. GraphGen: exploring interesting graphs in relational data (Demo proposal). *PVLDB* 8, 12 (2015).
[18] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. 2012. FDB: A query engine for factorised relational databases. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1232–1243.

A DATABASE SCHEMATA

Figure 5 shows the schemata of the different databases we used in our experiments, as well as in our examples when describing GraphGenQL and GraphGenDL.



(a) ***DBLP***

(b) ***University***

(c) ***TPC-H***

Figure 5