GraphGen: Exploring Interesting Graphs in Relational Data

Konstantinos Xirogiannopoulos Udayan Khurana Amol Deshpande
University of Maryland, College Park; {kostasx | udayan | amol}@cs.umd.edu

ABSTRACT

Analyzing interconnection structures among the data through the use of *graph algorithms* and *graph analytics* has been shown to provide tremendous value in many application domains. However, graphs are not the primary choice for how most data is currently stored, and users who want to employ graph analytics are forced to extract data from their data stores, construct the requisite graphs, and then use a specialized engine to write and execute their graph analysis tasks. This cumbersome and costly process not only raises barriers in using graph analytics, but also makes it hard to *explore* and *identify* hidden or implicit graphs in the data. Here we demonstrate a system, called GRAPHGEN, that enables users to declaratively specify graph extraction tasks over relational databases, visually explore the extracted graphs, and write and execute graph algorithms over them, either directly or using existing graph libraries like the widely used *NetworkX* Python library. We also demonstrate how unifying the extraction tasks and the graph algorithms enables significant optimizations that would not be possible otherwise.

1. INTRODUCTION

Analyzing the interconnection structure among the underlying entities or objects can provide significant insights and value in many application domains such as social networks, communication networks, finance, health, and many others. There is 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.), on such graph-structured data. This has led to the development of many specialized graph databases (e.g., Neo4j, Titan, OrientDB, etc.), and graph execution engines (e.g., Apache Giraph, GraphLab, Ligra, Galois, GraphX, XStream, to name a few). Recently several researchers have also investigated the possibility of executing graph analysis tasks using a relational database system (e.g., Vertexica [5], GRAIL [2], Aster Graph Analytics [9]).

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 resides in relational database systems; and it will likely to continue to be for a variety of reasons including inertia, the maturity of RDBMSs, and need to support transactions and SQL. Many interesting graphs are hidden in those relational databases, and extracting those graphs and analyzing them could provide significant value. Currently a user who wants to explore such graphs is forced to manually formulate the right SQL queries to extract relevant data, write scripts to convert the results into the format required by some graph database system, load the data into it, and write and execute the graph algorithms on the loaded graphs. This is a costly, labor-intensive, and cumbersome process, and poses a high barrier to using graph analytics.

In addition, exploring different *potential* graphs among the entities of interest is difficult and time-consuming in such a scenario. For example, consider the familiar DBLP dataset, where a user may want to construct a graph with the *authors* as the nodes. However, there are many ways to define the edges between the authors; e.g., we may put an edge between two authors: (1) if they co-authored a paper, or (2) if they co-authored a paper recently, or (3) if they co-authored multiple papers together, or (4) if they co-authored a paper with very few additional authors (indicating a true collaboration), or (5) if they presented a paper in the same conference session (indicating overlap of research interests), and so on. Some of these graphs might be too sparse or too disconnected to yield useful insights, while others may exhibit high density or noise. There are also other graphs that are of interest here, e.g., the bipartite author-publication or author-conference graphs. In less familiar contexts, e.g., given a financial dataset with information about companies, trades, etc., it is not at all clear what might be the interesting graphs to extract and analyze. Extracting all possible different graphs and understanding them would likely not be feasible, especially given that some of the graphs might be too large to even extract.

We are building a system, called GRAPHGEN, with the goal to make it easy for users to extract a variety of different types of graphs from a relational database, and execute graph analysis tasks or algorithms over them in memory. GRAPHGEN supports an expressive Domain Specific Language (DSL), based on *Datalog*, to specify graph(s) to be extracted from the relational data. The user may specify a single graph, or a collection of graphs to be extracted. GraphGen uses a translation layer to generate SQL queries to be issued to the database, and creates an efficient in-memory representation of the graph that is handed off to the user program or analytics task; the extracted graph can also be analyzed using existing tools like the Python NetworkX library. Our prototype implementation uses PostgreSQL as the underlying relational engine, but other database engines can be supported with minor modifications. GRAPHGEN employs several optimizations to reduce the memory requirements, that are enabled by the unification of the graph extraction and analysis tasks. GRAPHGEN also features a frontend that allows visual exploration of the extracted graphs, and graph

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

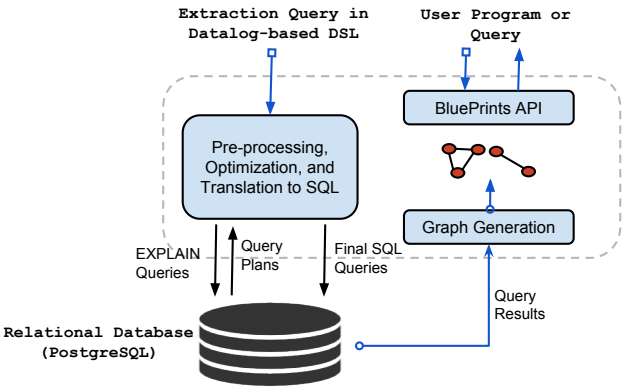

Figure 1: GraphGen Overview

discovery and exploration component that can be used to enumerate different graphs that could be created given a relational schema.

Note that, GRAPHGEN has fundamentally different goals than the recent work on graph analytics using relational databases (e.g., Vertexica [5], GRAIL [2], Aster Graph Analytics [9], SQL Server-based approach by Najork et al. [7]). In Vertexica and GRAIL, a graph with a given set of nodes and edges is normalized and stored in the relational database, and a subset of graph analysis tasks are mapped to relational operations. Those works do not consider the problem of extracting graphs from relational data, and can only execute analysis tasks that can be written using the vertex-centric programming framework. GRAPHGEN, on the other hand, pushes some computation to the relational engine, but most of the complex graph algorithms are executed in memory on a graph representation of the data. This allows GRAPHGEN to execute more complex analysis tasks like community detection, dense subgraph detection, matching, etc., as long as the extracted graph fits in memory (as we discuss later, we use a condensed in-memory representation that allows us to handle larger graphs than can fit in memory).

Aster Graph Analytics [9] also supports specifying graphs within an SQL query, and applying graph analytics algorithms on those graphs. However, the interface for specifying which graphs to extract is not intuitive and limits the types of graphs that can be extracted; Aster also only supports the vertex-centric API for writing graph algorithms. There also exist systems for migrating a relational database to a graph database by using the relational schema to reason about the graph structure [1]. Similarly, GraphBuilder [4] is a MapReduce-based framework for extracting graphs from unstructured data through user-defined Java functions for node and edge specifications. However, users are typically not interested in completely migrating their data over to a graph database.

Ringo [8] has somewhat similar goals to GraphGen and provides operators to convert from in-memory relational table representation to graph reprsentation; however it does not provide an expressive declarative DSL for graph extraction, and does not consider the optimizations that we discuss here. Ringo does support a large library of built-in graph algorithms, and we plan to support Ringo as a frontend analytics engine for our system.

2. GRAPHGEN OVERVIEW

We begin with a brief description of the key components of the GRAPHGEN system, and how data flows through them (Figure 1). We then sketch our Datalog-based DSL for specifying which graphs to extract, and briefly discuss some of the optimization techniques we employ. We also briefly discuss how we automatically enumerate a collection of graphs to explore given a relational schema.

2.1 System Architecture

The cornerstone of the system is an abstraction layer that sits on top of an RDBMS, accepts a graph extraction task and constructs one or more graphs in memory that can then be analyzed by the user program. The graph extraction task is specified using a Datalog-like DSL, where the user specifies how to construct the nodes and the edges of the graph(s). This specification is parsed by a Datalog parser, which then issues a set of EXPLAIN queries to the database engine to extract relevant statistics about the data. This helps us to estimate the size of the requested graph output, and to decide whether to use a regular or condensed graph representation. Although we currently use the optimizer-provided estimates directly for this purpose, given the known limitations of the selectivity estimation process (especially for self-joins), we plan to develop techniques to maintain additional information to improve those estimates. The system then constructs one or more batches of SQL queries, where each batch constructs one or more of the graphs that need to be constructed. We aim to ensure that the total size of the graphs constructed in a single batch is less than the total amount of memory available, so that the graphs can be constructed and analyzed in memory. The batches are then executed one after the other, with the control handed off to the user programs in between (i.e., after construction of the graphs in a batch).

GRAPHGEN also features a graph discovery and exploration component (not shown in the figure) that provides two main functionalities. First, it allows a user to specify a graph extraction query and to interactively explore the returned graphs (Section 3). Second, given a relational schema, it enumerates a collection of different graphs that could be created over a set of entities in that schema and allows the user to explore those in an interactive fashion (Section 2.3).

2.2 Query Language and API

Our proposed DSL for graph extraction over relational data is based on Datalog, which has been shown to be an effective centerpiece in enabling declarative specification in a range of domains including networking, data cleaning, machine learning, and social network analysis. We use the *property graph model* as the target graph data model, primarily because of the open-source ecosystem that has been built around it. In the property graph model, each vertex and each edge is associated with a set of properties (as key-value pairs), that may be different for different vertices/edges.

To extract a graph, a user needs to specify how to construct the vertices and the edges, and their required properties. Figure 2(i) shows a simple query that uses basic Datalog to specify a *co-authors* graph to be extracted from the DBLP dataset (relevant schema: *Author(ID, Name), Publication(PubID, Title, ConfID), AuthorPub(ID, PubID), PubConf(PubID, ConfID)*). Both `Nodes` and `Edges` are treated as special keywords, and the variable names used in the head of a `Nodes` or `Edges` rule specify the properties to be constructed. We currently support a basic subset of Datalog without recursion, but augmented with aggregates, to support the example graphs discussed earlier in Section 1.

Figure 2(ii) shows a multi-graph extraction query, where we extract 1-hop neighborhoods in the above *co-authors* graph. We introduce a special `For` construct for this purpose; that can also be used to specify, e.g., multiple historical snapshots of the graph to be extracted (corresponding to different time points) [6]. Heterogeneous k-partite graphs are specified using multiple `Nodes` rules.

GRAPHGEN can be imported by a user program as a Java library. The library provides a static `generateGraph()` method, that takes as input a graph extraction query in our DSL, and returns a collection of `Graph` objects that export the widely used *Blueprints* API. Blueprints is a generic graph Java API, that provides graph

```
(1) Nodes(ID, Name) :- Author(ID, Name)
    Edges(ID1, ID2) :- AuthorPub(ID1, PubID),
                       AuthorPub(ID2, PubID)
(2) For Author(X, _).
      Nodes(ID, Name) :- Author(ID, Name), ID = X.
      Nodes(ID, Name) :- AuthorPub(X,P), AuthorPub(ID,P),
                         Author(ID, Name)
      Edges(ID1, ID2) :- Nodes(ID1, _), Nodes(ID2, _),
                         AuthorPub(ID1, P), AuthorPub(ID2, P)
```

Figure 2: Graph Extraction Query Examples

access methods like `getVertices()`, `getEdges()`, etc., and is used by several graph processing and programming frameworks (including Gremlin, a popular graph query language). By supporting the Blueprints API, we immediately enable use of many of these already existing toolkits over extracted graphs.

In our current implementation, a returned `Graph` object may be a `TinkerGraph`, or a subclass that supports condensed representation (Section 2.4). `TinkerGraph` is an efficient, in-memory implementation of the property graph model, and is part of the open-source TinkerPop stack (`http://www.tinkerpop.com/`). The GRAPHGEN library also provides a functional interface to apply a function to the generated graphs and return only the results.

2.3 Graph Enumeration Framework

For a user interested in performing graph analysis on the entities in a relational dataset, constructing meaningful graphs requires careful examination of the relational schema, as well as the data itself. Seemingly natural interconnection structures can turn out to be too sparse or too dense or too disconnected to lead to meaningful insights. Manually trying out different possibilities is a time-consuming process, and may miss out on insightful graphs. To aid the user in this process, we are building a *graph enumeration framework* that performs automated discovery of graphs in a dataset using a collection of universal rules. We briefly sketch the framework here. In a normalized database, we consider all *primary keys* to be candidate node sets. For example, in the DBLP dataset, the set of *author-ids* is a candidate node set, and so is the set of *publication-ids*. The process of graph discovery hinges upon finding possible connections for a node set with another (including itself). These connections can be found through the data description constraints such as foreign keys, functional dependencies and attribute co-memberships for a table. More concretely, the problem of discovering homogeneous or bi-partite graphs (i.e., over one or two node sets) can be formalized as follows. Let $\mathcal{N}$ denote the set of primary keys, and $\mathcal{N}'$ denote the set of all remaining attributes in the schema. Let G denote a graph (called *schema graph*) over $\mathcal{N} \cup \mathcal{N}'$, where an edge indicates either a foreign key relationship between a primary key and another attribute in a different table, or a functional dependency between a primary key and another attribute in the same table. A potential homogeneous or bi-partite graph in this dataset can be described as a pair $(X, Y), X, Y \in \mathcal{N}$ (X and Y may be identical), and a path p connecting the two in the schema graph G (the path may traverse the same edge twice, and may visit a node multiple times). A larger set of graphs can be enumerated by adding rules that generate filtering conditions or aggregate operations using the schema graph. Our current implementation supports enumerating graphs without those, and we are working on developing a comprehensive framework for supporting those types of rules.

2.4 Implementation Details

We briefly discuss how GRAPHGEN is implemented, and some of the optimizations it features. GRAPHGEN has been implemented in Java, with PostgreSQL as the backend relational engine. The front-end (discussed in Section 3) uses JavaScript and D3.js for different components of the visualization and processing. Java Servlets are used for the webserver request processing, and all the client-server calls happen asynchronously through AJAX HttpRequests.

As discussed in Section 2.1, GRAPHGEN translates the declaratively specified extraction task into SQL queries to be issued to the underlying relational engine. There are several trade-offs here that need to be navigated carefully.

First, in the case of a single-graph extraction, it may be beneficial to construct the graph lazily by only pushing a part of the computation to the underlying relational engine and finishing the computation on demand during the execution of the graph algorithm. Consider, e.g., the construction of the *co-authors* graph on a bibliography dataset. For a publication with k authors, the *co-authors* graph contains $k(k-1)/2$ edges between those authors. For datasets that contain many publications with large numbers of authors, the resulting graph can be very dense and can require significant memory to load. By postponing the final self-join in the corresponding SQL query, we can significantly reduce the amount of data that is transferred between the relational engine and GRAPHGEN. Further, it allows us to represent the graph in a *condensed fashion* in memory, where a clique in the graph is replaced by a *virtual node* that is connected to all the vertices in the clique (in this example, each publication corresponds to a clique over its authors). Such condensed representations have been explored in past work [3], where algorithms to construct condensed representation are developed; here, however, we get the condensed representation for free. The downside of this approach is that it takes longer to analyze the resulting graph, especially when supporting generic APIs like BluePrints. On the other hand, some graph algorithms (e.g., breadth-first search, connected components, etc.) can be executed directly on the condensed representation. A challenge here is also to decide when to use the condensed representation. We propose using the query optimizer for this purpose, by asking it to generate a series of query plans and then analyzing the query plans to make the decisions in a cost-based manner. Our initial experiments show that using the condensed representation can reduce the memory requirements significantly, by a factor of 5 or more on relatively small graphs with 100k to 500k vertices (the benefits are expected to be higher for larger graphs).

Second, there are many possible ways to execute a multi-graph extraction query. A simple option is to execute a separate set of queries for each graph to be extracted (that may be batched together when sent to the relational engine to reduce the number of round-trips). However, in many cases, the graphs to be extracted have significant overlaps (e.g., when we are trying to extract historical snapshots, or node neighborhoods), and that approach may read the same data multiple times from the database. At the other extreme, we may issue a single "union" query that extracts a union of all the graphs, and then separates out the graphs in GRAPHGEN. That approach however requires a significant duplication of effort and does not utilize the capabilities of the database sufficiently. We instead employ a "tagging-based" approach where we combine the extraction of a small number of graphs into a single batch; for each batch, a small number of queries is issued (e.g., in the simplest case, one query each for extracting nodes and edges) such that the tuples in the result sets are tagged with the graphs they belong to. For a small subset of the DBLP dataset containing 2576 different authors, the total time to extract the neighborhoods of all nodes in the *co-authors* graph came down from 18s to approximately 2.2s using this optimization. We omit further details due to lack of space.

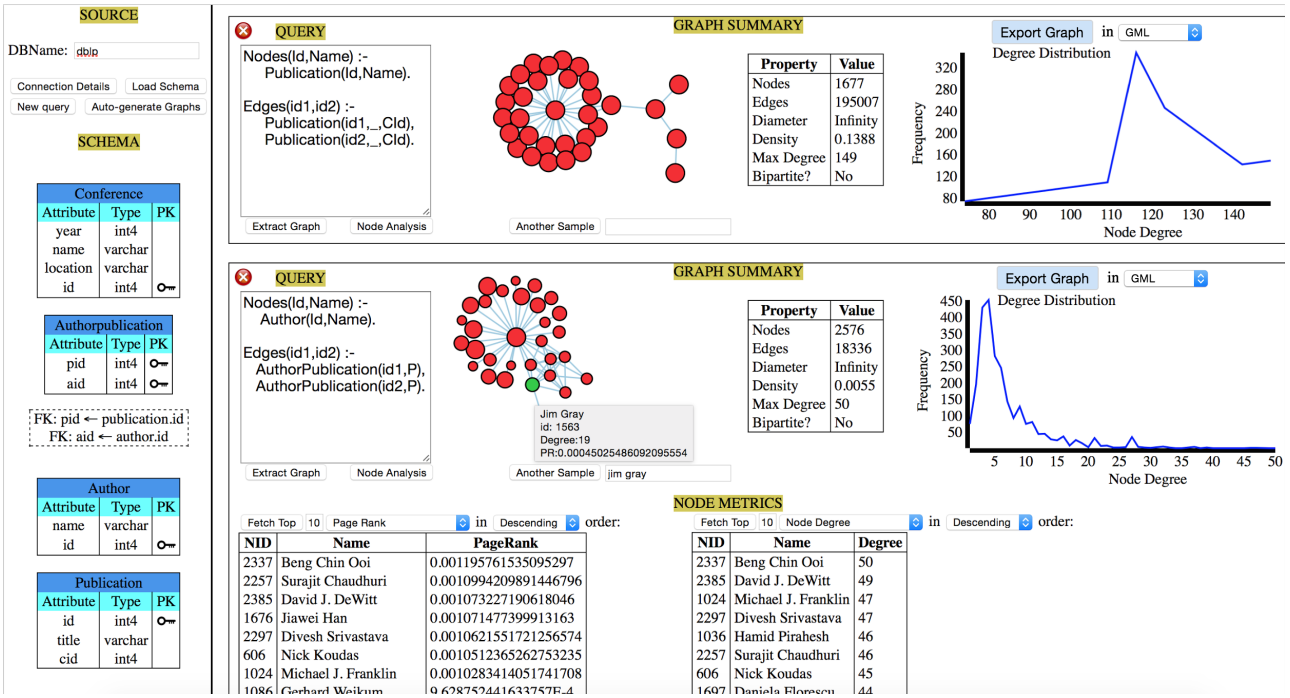

Figure 3: Graph Generator tool with two different graph extraction queries for a small DBLP dataset.

3. DEMONSTRATION

Finally, we briefly describe the interactive graph discovery and exploration front-end that we have developed, and discuss the demonstration plan. The front-end allows a user to: (a) connect to an existing relational database and view its schema, (b) write queries in our DSL to extract different graphs, (c) explore the graphs through node-link visualizations and various global and node-level metrics, and (d) compare graphs extracted using different queries. Figure 3 shows one such snapshot where the user connects to the DBLP database. On the top left, the database name and other connection details can be specified. `Load Schema` displays the list of tables, attribute information, and constraints such as primary and foreign keys. The `New Query` option creates a new pane on the right. Here, the user would write a graph extraction query using the schema details displayed on the left.

`Extract Graph` initiates the graph generation task at the backend, along with the computation of several global and node-level metrics. Upon its completion, a small subset of the extracted graph is displayed using a *force-directed layout*. It also displays graph statistics such as node count, density, diameter, etc., and a plot of the node degree distribution. The user can visualize specific portions of the graph through the `Another Sample` option by specifying a keyword in the text-box besides it. The system uses a keyword search on nodes' attributes and returns a subgraph around the node with the first occurrence. In case of a missing keyword or the hint being unusable, a random subgraph is presented instead. Using the `Node Analysis` option, a user can view and sort by different metrics for nodes, such as degree, betweenness centrality, PageRank, clustering coefficient, and others. Multiple query panes, launched through the `New Query` option, are aligned such that different queries and graphs are vertically juxtaposed for comparison. Moreover, by selecting `Export Graph`, the entire generated graph can be serialized to disk into one of the standard formats in the drop-down list. This gives the user the ability to load the graph into any graph library that supports these formats, and execute graph algorithms against it. Finally, if the user is unfamiliar with the dataset and wants to explore, she can use the `Auto-generate Graphs` option. Based upon the database schema, it automatically populates a few panes with valid extraction queries and resultant graphs.

Demonstration Plan: During the demonstration, the conference attendees will be able to use the front-end to write graph extraction queries over various pre-populated datasets, and visually explore the results. The conference attendees will also be encouraged to think about potential graphs among the entities in the dataset, and how those can be mapped to the proposed graph extraction DSL. Certain pre-selected queries will be used to demonstrate graph exploration and comparison. We will also demonstrate how users can effortlessly operate upon the extracted graphs using the Python NetworkX graph library and its built-in graph algorithms.

Acknowledgments: This work was supported by NSF under grant IIS-1319432, and by an IBM Faculty Award.

4. REFERENCES

[1] R. De Virgilio, A. Maccioni, and R. Torlone. Converting relational to graph databases. In *GRADES*, 2013.

[2] J. Fan, G. Raj, and J. Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[3] T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. *JCSS*, 1995.

[4] N. Jain, G. Liao, and T. Willke. GraphBuilder: A Scalable Graph ETL Framework. In *GRADES*, 2013.

[5] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. VERTEXICA: your relational friend for graph analytics! *PVLDB*, 7(13):1669–1672, 2014.

[6] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, 2013.

[7] M. Najork et al. Of hammers and nails: An empirical comparison of three paradigms for processing large graphs. In *WSDM*, 2012.

[8] Y. Perez et al. Ringo: Interactive graph analytics on big-memory machines. In *SIGMOD*, 2015.

[9] D. Simmen et al. Large-scale Graph Analytics in Aster 6: Bringing Context to Big Data Discovery. *PVLDB*, 7(13), 2014.