Extracting and Analyzing Hidden Graphs from Relational Databases

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

ABSTRACT

Analyzing interconnection structures among underlying entities or objects in a dataset through the use of graph analytics has been shown to provide tremendous value in many application domains. However, graphs are not the primary representation choice for storing most data today, and in order to have access to these analyses, users are forced to *manually extract* data from their data stores, *construct* the requisite graphs, and then *load* them into some graph engine in order to execute their graph analysis task. Moreover, in many cases (especially when the graphs are dense), these graphs can be *significantly larger* than the initial input stored in the database, making it infeasible to construct or analyze such graphs in memory. In this paper we address both of these challenges by building a system that enables users to *declaratively* specify graph extraction tasks over a relational database schema and then execute graph algorithms on the extracted graphs. We propose a declarative domain specific language for this purpose, and pair it up with a novel *condensed*, in-memory representation that significantly reduces the memory footprint of these graphs, permitting analysis of larger-than-memory graphs. We present a general algorithm for creating such a condensed representation for a large class of graph extraction queries against arbitrary schemas. We observe that the condensed representation suffers from a *duplication* issue, that results in inaccuracies for most graph algorithms. We then present a suite of in-memory representations that handle this duplication in different ways and allow *trading off* the memory required and the computational cost for executing different graph algorithms. We also introduce several novel *de-duplication* algorithms for removing this duplication in the graph, which are of independent interest for graph compression, and provide a comprehensive experimental evaluation over several real-world and synthetic datasets illustrating these trade-offs.

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, sciences, and many others. 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.) on graph-structured data. Many specialized graph databases (e.g., Neo4j [1], Titan [3], OrientDB [2], etc.), and graph execution engines (e.g., Giraph [8], GraphLab [26], Ligra [34], Galois [29], GraphX [16]) 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, as their name suggests, often include various interesting relationships between entities within them, and can contain many hidden, interesting graphs. 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 create 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 (which may indicate a true collaboration), or (5) if they attended the same conference, 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; however, many of these graphs may result in different types of interesting insights. It is also often interesting to juxtapose and compare graphs constructed over different time periods (i.e., *temporal graph analytics*). There are many other graphs that are possibly of interest here, e.g., the bipartite author-publication or author-conference graphs – identifying potentially interesting graphs itself may be difficult for large schemas with 100s of tables.

Currently a user who wants to explore such structures in an existing database is forced to: (a) manually formulate the right SQL queries to extract relevant data (which may not complete because of the space explosion discussed below), (b) write scripts to convert the results into the format required by some graph database system or computation framework, (c) load the data into it, and then (d) 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 leveraging graph analytics on these datasets. This is especially a problem given the large numbers of entity types present in most real-world datasets and a myriad of potential graphs that could be defined over those.

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[1], and execute graph analysis tasks or algorithms over them in memory. GraphGen supports an expressive Domain Specific Language (DSL), based on *Datalog* [6], allowing users to specify a single graph or a collection of graphs to be extracted from the relational database (in essence, as *views* on the database tables). GraphGen uses a translation layer to

[1]Although GraphGen (name anonymized for submission) currently only supports PostgreSQL, it requires only basic SQL support from the underlying storage engine, and could simply *scan* the tables if needed.

Graph	Representation	Edges	Extraction Latency (s)
DBLP	Condensed	17,147,302	105.552
	Full Graph	86,190,578	> 1200.000
IMDB	Condensed	8,437,792	108.647
	Full Graph	33,066,098	687.223
TPCH	Condensed	52,850	15.520
	Full Graph	99,990,000	> 1200.000
UNIV	Condensed	60,000	0.033
	Full Graph	3,592,176	82.042

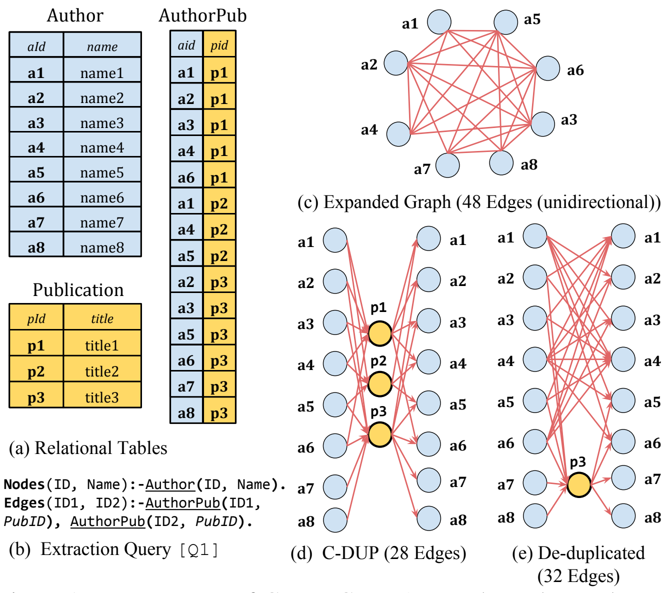
Table 1: Extracting graphs in GRAPHGEN using our *condensed* representation vs extracting the full graph. IMDB: Co-actors graph (on a subset of data), DBLP: Co-authors graph, TPCH: Connect customers who buy the same product, UNIV: Connect students who have taken the same course (synthetic, from *http://db-book.com*)

generate the appropriate 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. GRAPHGEN supports a general-purpose Java Graph API as well as the standard *vertex-centric API* for specifying analysis tasks like PageRank. Figure 1 shows a toy DBLP-like dataset, and the query that specifies a “co-authors” graph to be constructed on that dataset. Figure 1c shows the requested co-authors graph.

The main scalability challenge in extracting graphs from relational tables is that: the graph that the user is interested in analyzing may be too large to extract and represent in memory, even if the underlying relational data is small. There is a space explosion because of the types of high-output[2] joins that are often needed when constructing these graphs. Table 1 shows several examples of this phenomenon. On the DBLP dataset restricted to journals and conferences, there are approximately 1.6 million authors, 3 million publications, and 8.6 million author-publication relationships; the co-authors graph on that dataset contained 86 million edges, and required more than half an hour to extract on a laptop. The *condensed* representation that we advocate in this paper is much more efficient both in terms of the memory requirements and the extraction times. The DBLP dataset is, in some sense, a best-case scenario since the average number of authors per publication is relatively small. Constructing the *co-actors* graph from the IMDB dataset results in a similar space explosion. Constructing a graph connecting pairs of customers who bought the same item in a small TPCH dataset results in much larger graph than the input dataset. Even on the DBLP dataset, a graph that connects authors who have papers at the same conference contains 1.8B edges, compared to 15M edges in the condensed representation.

In this paper, we address the problem of analyzing such large graphs by storing and operating upon them using a novel *condensed* representation. The relational model already provides a natural such condensed representation, obtained by omitting some of the high-output joins from the query required for graph extraction. Figure 1(d) shows an example of such a condensed representation for the *co-authors* graph, where we create explicit nodes for the pubs, in addition to the nodes for the authors; for two authors, u and v, there is an edge $u \rightarrow v$, iff there is a directed path from u to v in this representation. This representation generalizes the idea of using cliques and bicliques for graph compression [10, 22]; however, the key challenge for us is not generating the representation, but rather dealing with *duplicate paths* between two nodes.

In Figure 1, we can see such a duplication for the edge $a1 \rightarrow a4$ since they are connected through both $p1$ and $p2$. Such duplication prevents us from operating on this condensed representation directly. We develop a suite of different in-memory representations for this condensed graph that paired with a series of “de-


Author
AuthorPub
Publication
Nodes(ID, Name):-Author(ID, Name).
Edges(ID1, ID2):-AuthorPub(ID1, PubID), AuthorPub(ID2, PubID).
a1
a2
a3
a4
a5
a6
a7
a8
p1
p2
p3

Author

aId	name
a1	name1
a2	name2
a3	name3
a4	name4
a5	name5
a6	name6
a7	name7
a8	name8

AuthorPub

aid	pid
a1	p1
a2	p1
a3	p1
a4	p1
a6	p1
a1	p2
a4	p2
a5	p2
a2	p3
a3	p3
a5	p3
a6	p3
a7	p3
a8	p3

Publication

pId	title
p1	title1
p2	title2
p3	title3

(a) Relational Tables

```
Nodes(ID, Name):-Author(ID, Name).
Edges(ID1, ID2):-AuthorPub(ID1,
PubID), AuthorPub(ID2, PubID).
```

(b) Extraction Query [Q1]

(c) Expanded Graph (48 Edges (unidirectional))

(d) C-DUP (28 Edges)

(e) De-duplicated (32 Edges)

Figure 1: Key concepts of GRAPHGEN. (Note: the author nodes here are being shown twice for the sake of simplicity, they are *not* being stored twice)

duplication” algorithms, leverage a variety of techniques for dealing with the problem of duplicate edges and ensure only *a single edge* between any pair of vertices. We discuss the pros and cons of each representation and present a formal analysis comparing their trade-offs. We also present an extensive experimental evaluation, using several real and synthetic datasets.

Key contributions of this paper include:

- A high-level declarative DSL based on Datalog for intuitively specifying graph extraction queries.
- A general framework for extracting a condensed representation (with duplicates) for a large class of extraction queries over arbitrary relational schemas.
- A suite of in-memory representations to handle the duplication in the condensed representation.
- A systematic analysis of the benefits and trade-offs offered by extracting and operating on these representations.
- Novel techniques for “de-duplicating” these condensed graphs.
- The first end-to-end system for enabling analytics on graphs that exist within purely relational datasets, efficiently, and without requiring complex ETL.

Outline: We begin with a brief discussion of how related work has leveraged relational databases for graph analytics in the past (Section 2). We then present an overview of GRAPHGEN, briefly describe the graph extraction DSL, and discuss how GRAPHGEN decides when the condensed representation should be extracted instead of the full graph (Section 3). We then discuss the different in-memory representations (Section 4) and present a series of deduplication algorithms (Section 5). Finally, we present a comprehensive experimental evaluation (Section 6).

2. RELATED WORK

There has been much work on graph data management in recent years, most of it orthogonal to the work we present. Superficially the most closely related work is the recent work on leveraging relational databases for graph analytics, whose aim is to show that specialized graph databases or analytics engines may be unnecessary. Vertexica [21, 20] and GRAIL [13] show how to nor-

[2]We use this term instead of “selectivity" terms to avoid confusion.

malize and store a graph dataset as a collection of tables in an RDBMS (i.e., how to "shred" graph data), and how to map a subset of graph analysis tasks to relational operations on those tables. EmptyHeaded [5] is a relational database engine focused on efficient graph processing by implementation of worst-case optimal join algorithms to provide fast query processing on graphs that are *stored as relational tables*. Aster Graph Analytics [35] and SAP HANA Graph Engine [32] also supports specifying graphs within an SQL query, 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. Aster also only supports the vertex-centric API for writing graph algorithms. SQLGraph [36] addresses the challenges in storing property graphs (a common graph data model) in an RDBMS, focusing on the storage and layout issues; they also show how to answer Gremlin queries, which is a graph traversal-based DSL for querying property graphs, by translating them to SQL.

GraphGen has different goals compared to that prior work and faces fundamentally different challenges (Figure 2). Those systems do not consider the problem of *extracting* graphs from existing relational datasets, and can only execute analysis tasks that can be written using the vertex-centric programming framework or can be mapped to SQL. GraphGen, on the other hand, focuses on datasets that are *not* stored in a graph format. While it pushes some computation to the relational engine, most of the complex graph algorithms are executed on a graph representation of the data in memory through a full-fledged native graph API. This makes GraphGen also suitable for more complex analysis tasks that are often cannot be succintly expressed and computed using the vertex-centric framework. This native graph API enables access to analyses like community detection, dense subgraph detection/ matching, etc., provided the extracted condensed graph fits in memory.

There also exist systems for migrating a relational database to a graph database by using the relational schema to reason about the graph structure [12]. Users however are typically not interested in completely migrating their data over to a graph database if they aren't strictly dealing with graph-centric workloads. Ringo [31] has somewhat similar goals to GraphGen and provides operators for converting from an in-memory relational table representation to a graph representation. It however does not consider expensive high-output joins that are often necessary for graph extraction, or the alternate in-memory representation optimizations that we discuss here, but instead assumes a powerful large-memory machine to deal with both issues. Ringo does include an extensive library of built-in graph algorithms in SNAP [25], and we do plan to support Ringo as a front-end analytics engine for GraphGen.

Table2Graph [24] is built towards extracting large graphs from relational databases using MapReduce jobs, while de-coupling the execution of the required join operations from the RDBMS. In Table2Graph users need to provide a set of descriptive XML files that specify the exact mappings for nodes, edges, properties and labels. Similarly, GraphBuilder [19] is a MapReduce-based framework for extracting graphs from unstructured data through user-defined Java functions for node and edge specifications. GLog [15] is a declarative graph analysis language based on Datalog which is evaluated into MapReduce code towards efficient graph analytics in a distributed setting. Again, the underlying data model they use (Relational-Graph tables) assumes that the complete graph to be analyzed *explicitly exists* as vertices and edges tables. It's important to note that none of the mentioned works are concerned with providing an intuitive interface or language for the mapping and extraction of hidden graphs from the relational schema.

There has been much work on large-scale, distributed graph an-

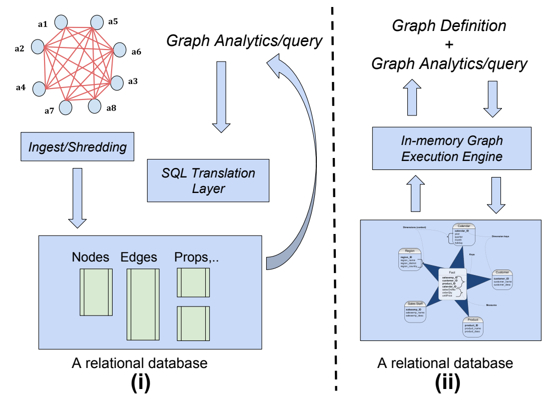


Figure 2: GraphGen (right) has fundamentally different goals than recent work on using relational databases for graph analytics (left)

alytics systems, some of which have adopted high-level declarative interfaces based on Datalog [37, 33, 15, 9]. Our use of Datalog is restricted to specifying which graphs to extract (in particular, we do not allow recursion), and we also do not consider distributed execution issues in this paper. Combining declarative graph extraction ala GraphGen, and high-level graph analytics frameworks proposed in that prior work, is a rich area for future work.

There has also been a significant amount of work on minimizing computation time in graph processing through the development of distributed graph processing engines. Some of these systems are built for in-memory computation [29, 34, 16, 28, 17], while [23] is disk-based and enable large graph processing through strategic data placement onto disk that enables more efficient scans from disk. The GraphGen prototype currently only works on a single, shared memory system, there is however potential for integrating GraphGen with some of this work in order to optimize the entire pipeline from graph definition and extraction all the way up to execution, which can be potentially be sped up by using some of these techniques.

Some very relevant work has also emerged recently on using the graph data model for general query processing on queries that involve a large number of joins, by loading relational data as a graph into a distributed graph processing engine [27].

An initial prototype of the GraphGen system was recently demonstrated [4], where the primary focus was on automatically proposing and extracting hidden graphs given a relational schema. The current submission provides an in-depth description of the techniques and algorithms for efficient graph extraction as well as a comprehensive experimental evaluation of the trade-offs therein.

3. SYSTEM OVERVIEW

We begin with a brief description of the key components of GraphGen, and how data flows through them. We then sketch our Datalog-based DSL for specifying graph extraction jobs, and APIs provided to the users after a graph has been loaded in memory.

3.1 System Architecture

The inner workings of GraphGen and the components that orchestrate its functionality are demonstrated in Figure 3. The cornerstone of the system is an abstraction layer that sits atop an RDBMS, accepts a graph extraction task, and constructs the queried graph(s) in memory, which can then be analyzed by a user program. The graph extraction task is expressed using a Datalog-like DSL, where the user specifies how to construct the nodes and the edges of the graph (in essence, as *views* over the underlying tables). This spec-

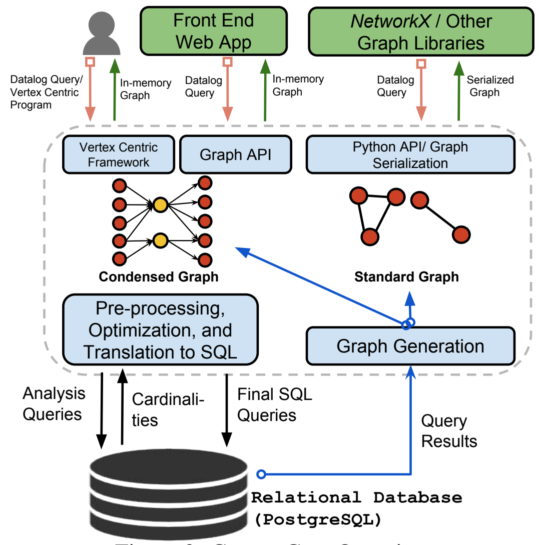


Figure 3: GRAPHGEN Overview

ification is parsed by a custom parser, which then analyzes the *selectivities* of the joins required to construct the graph by using the statistics in the system catalog. This analysis is used to decide whether to hand over the partial or complete edge creation task to the database, or to skip some of the joins and load the implicit edges in memory in a condensed representation (Section 4.2).

The system then builds one or more batches of SQL queries, where each batch defines one or more of the graphs that need to be extracted. We aim to ensure that the total size of the graphs constructed in a single batch (in our memory-efficient representation) is less than the total amount of memory available, so that the graphs can be analyzed in memory. The queries are executed in sequence, and the output graph object(s) is (are) handed to the user program. A major focus of this work is to enable analysis on *very large* graphs that would typically not fit in memory.

After extraction, users can (a) operate *directly* upon any portion of the graph using the Java Graph API, (b) define and run multi-threaded *vertex-centric* programs on it, (c) *visually* explore the graph through our front-end web application, or (d) *serialize* the graph onto disk (in its expanded representation) in a standardized format, so that it can be further analyzed using any specialized graph processing framework or graph library (e.g., NetworkX).

3.2 Datalog-Based DSL

Datalog has seen a revival in the recent years, and has been increasingly used for expressing data analytics workflows, and especially graph analysis tasks [18, 33, 15]. The main reason for its emergence lies in its elegance for naturally expressing recursive queries, but also in its overall intuitive and simple syntax choices. Our domain specific language (DSL) is based on a limited non-recursive subset of Datalog, augmented with looping and aggregation constructs; in essence, our DSL allows users to intuitively and succinctly specify nodes and edges of the *target graph* as *views* over the underlying database tables. We note that our goal here is **not** to specify a graph algorithm itself using Datalog (like Socialite [33]), although we plan to investigate that option in future work.

Our DSL uses three special keywords: `Nodes`, `Edges`, and `For`; the last keyword can be used for optionally expressing simultaneous extraction of multiple graphs. A graph specification comprises of at least one `Nodes` statement, followed by at least one `Edges` statement. Users may specify multiple `Nodes` and `Edges` statements in order to extract, say, *heterogeneous* graphs with multiple different types of vertices and edges.

```
[Q2] 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).

[Q3] Nodes(ID, Name) :- Customer(ID, Name).
     Edges(ID1, ID2) :- Orders(order_key1, ID1),LineItem(
          order_key1, part_key), Orders(order_key2, ID2),
          LineItem(order_key2,part_key).

[Q4] Nodes(ID, Name) :- Instructor(ID, Name).
     Nodes(ID, Name) :- Student(ID, Name).
     Edges(ID1, ID2) :- TaughtCourse(ID1, courseId),
          TookCourse(ID2, courseId)
```

Figure 4: Graph Extraction Query Examples (cf. Figure 1 for **Q1**)

The typical workflow for a user when writing a query in this DSL would be to initially inspect the database schema, figure out which relations are relevant to the graph they are interested in exploring, and then choose which attributes in those relations would connect the defined entities in the desired way. We assume here that the user is knowledgeable about the different entities and relationships existent in the database, and is able to formulate such queries; we have built a visualization tool that would allow users to discover potential graphs and construct such extraction queries in an interactive manner [4], however that is not the focus of this paper.

Figure 4 demonstrates several examples of extraction queries. In each one of these queries, a set of *common attributes* represents an equi-join between their respective relations. An extraction task can contain any number of joins; e.g. **`[Q1]`** in Figure 1, only requires a single join (in this case a self-join on the `AuthorPub` table), while **`[Q3]`** as shown in Figure 5a would require a total of 3 joins, some of which (in this case Orders(**order_key1**, ID1) $\bowtie$ LineItem(**order_key1**, partkey), and Orders(**order_key2**, ID2) $\bowtie$ LineItem(**order_key2**, partkey)) will be handed off to the database since they are highly selective key-foreign key joins. The extraction query **`[Q4]`** extracts a bi-partite (heterogeneous) directed graph between instructors and students who took their courses, shown in Figure 5b.

`[Q2]` shows how the `For` loop feature can be used to extract separate "ego-graphs" for every individual node. We have implemented a means for extracting these ego-graphs efficiently without the need for a series of independent SQL queries (through "tagging" returned rows), but omit the details due to lack of space.

3.3 Parsing and Translation

The first step towards communicating the user defined graph extraction to the system is the parsing of the Datalog query and translation into the appropriate SQL. We have built a custom parser for the DSL described above using the ANTLR [30] parser generator. The parser is then used to create the Abstract Syntax Tree (AST) of the query which is in turn used for translation into SQL. GRAPHGEN evaluates programs in our DSL and translates them into SQL, line at a time. Connections between the lines of code loosely exist (e.g., code below a `For` defines a multiple ego-graph query, and translation is done accordingly), and are maintained throughout the execution of the code from one statement to the next.

The translation itself requires a full walk of the AST, during which the system gathers information about the statement, loads the appropriate statistics for each involved relation from the database and creates a *translation plan* based on the information gathered. Lastly, the generation of the final SQL queries is actually triggered upon exiting the AST walk and is based on this translation plan. The specifics depend on the nature of the *Edges* statement(s).

Case 1: Each of the *Edges* statements corresponds to an acyclic, aggregation-free query. In that case, we may load a condensed representation of the graph into memory (Section 4.2).

Case 2: At least one Edges statement violates the above condition, in which case, we simply create a single SQL statement to construct the edges and execute that to load the graph into memory in an expanded fashion.

The rest of this paper focuses on Case 1 (which covers all the examples shown till now) and shows how to reduce memory requirements and execute graph algorithms more efficiently than loading the entire expanded graph in memory. In future work, we plan to generalize our ideas to other classes of queries in Case 2.

3.4 Analyzing the Extracted Graphs

The most efficient means to utilize GRAPHGEN is to directly operate on the graph either using our native Java Graph API, or through a *vertex-centric API* that we provide. Both of these have been implemented to operate on all the in-memory (condensed or otherwise) representations that we present in Section 4.

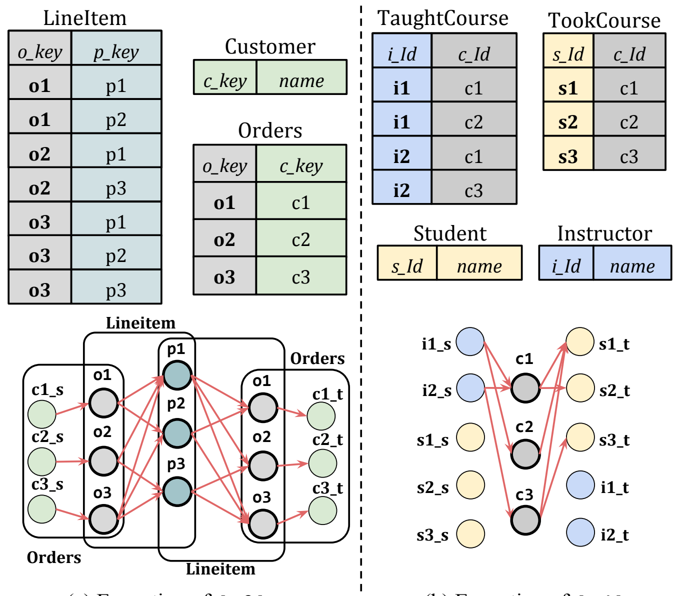
Basic Data Structure: The basic data structure we choose to store a graph in memory is a variant of Compressed Sparse Row (CSR) [11] format that allows for a higher degree of mutability (even the condensed representations uses this data structure underneath). Instead of maintaining single arrays for each node's incoming and outgoing edges like traditional CSR, we instead choose to maintain a set of two `ArrayLists`, for the in- and out-going edges of each vertex. We use Java `ArrayLists` instead of linked lists for space efficiency. Our custom graph data structure ensures the same graph traversal operation runtime complexities as CSR with the small space overhead of `ArrayLists`. On top of this, our data structure supports deletion of edges and vertices with the appropriate overheads those entail. Because we are using `ArrayLists`, the `deleteVertex` operation in particular requires rebuilding of the entire index of vertices. We therefore implement a lazy deletion mechanism where vertices are initially only removed from the index, thus logically removing them from the graph, and are then physically removed from the vertices list, in *batch*, at a later point in time. This way only a single re-building of the vertices index is required after a batch removal.

Java API: All of our in-memory representations implement a simple graph API, consisting of the following 7 operations:

- `getVertices()`: This function returns an iterator over all the vertices in the graph.
- `getNeighbors(v)`: For a vertex v, this function returns an iterator over the neighbors of v, which itself supports the standard `hasNext()` and `next()` functions. If a list of neighbors is desired (rather than an iterator), it can be retrieved using `getNeighbors(v).toList`.
- `existsEdge(v, u)`: Returns *true* if there is an edge between the two vertices.
- `addEdge(v, u), deleteEdge(v, u), addVertex(v), deleteVertex(v)`: These allow for manipulating the graphs by adding or removing edges or vertices.

The `Vertex` class also supports setting or retrieving properties associated with a vertex.

Vertex-centric API: The vertex-centric conceptual model has been extensively used in the past to express complex graph algorithms by following the "think-like-a-vertex" methodology in designing these algorithms. We have implemented a simple, multi-threaded variant of the *vertex-centric framework* in GRAPHGEN that allows users to implement a COMPUTE function and then execute that against the extracted graph regardless of its in-memory representation. The framework is based on a `VertexCentric` object which coordinates the multi-threaded execution of the `compute()` function for each job. The coordinator object splits the graph's nodes into chunks depending upon the number of cores in the machine, and distributes the load evenly across all cores. It also keeps track of the current superstep, monitors the execution and triggers a termination event when all vertices have voted to a halt. Users simply need to implement the `Executor` interface which contains a single method definition for `compute()`, instantiate their executor and call the `run()` method of the VertexCentric coordinator object with the Executor object as input. The implementation of message passing we've adopted is similar to the *gather-apply-scatter (GAS)* model used in GraphLab [**?**] in which nodes communicate by directly accessing their neighbors' data, thus avoiding the overhead of explicitly storing messages in some intermediary data structure.



LineItem

o_key	*p_key*
o1	p1
o1	p2
o2	p1
o2	p3
o3	p1
o3	p2
o3	p3

Customer

c_key	*name*

Orders

o_key	*c_key*
o1	c1
o2	c2
o3	c3

TaughtCourse

i_Id	*c_Id*
i1	c1
i1	c2
i2	c1
i2	c3

TookCourse

s_Id	*c_Id*
s1	c1
s2	c2
s3	c3

Student

s_Id	*name*

Instructor

i_Id	*name*

(a) Execution of `[Q3]`   (b) Execution of `[Q4]`

Figure 5: Extraction examples: (a) Multi-layered condensed representation, (b) extracting a heterogeneous bipartite graph

External Libraries: GRAPHGEN can also be used through a library called *higraphpy*, a Python wrapper over GRAPHGEN allowing users to run queries in our DSL through simple Python scripts and serialize the resulting graphs to disk in a standard graph format, thus opening up analysis to any graph computation framework or library. A similar workflow is used in the implementation of our front-end web application [4] through which users can get a *visual* cue of the kinds of graphs that exist within their relational schema, and decide whether they would be interesting to analyze. However, since those require loading the graph into an external application, the graph has to be in its expanded (non-condensed) format.

4. IN-MEMORY REPRESENTATION AND TASK EXECUTION

The key efficiency challenge with extracting graphs from relational databases is that: in most cases, because of the normalized nature of relational schemas, queries for extracting explicit relationships (i.e., *edges*) between entities from relational datasets (i.e., *nodes*) requires expensive non-key joins. Because of this, the extracted graph may be much larger than the input size itself (cf. Table 1). We propose instead maintaining and operating upon the extracted graph in a *condensed* fashion. We begin with describing a novel condensed representation that we use and discuss why it is ideally suited for this purpose; we also discuss the *duplication* issue with this representation. We then present an general algorithm for constructing such a condensed representation for a graph extraction query over an arbitrary schema, that guarantees that the condensed representation requires at most as much memory as loading all the underlying tables in the worst case. The condensed representation and the algorithm can both handle any non-recursive, acyclic graph extraction query over an arbitrary schema.

We then propose a series of in-memory variations of the basic condensed representation that handle the duplication issue through uniquely characterized approaches. These representations are products of a single-run pre-processing phase on top of the condensed representation using an array of algorithms described in Section 5. We also expand on the natural trade-offs associated with storing and operating on each in-memory representation.

4.1 Condensed Representation & Duplication

The idea of compressing graphs through identifying specific types of structures has been around for a long time. Feder and Motwani [14] presented the first theoretical analysis of *clique-based compression*; there, one takes a *clique* C in the given (undirected) graph, adds a new *virtual* node corresponding to C (say v_C), removes all the edges amongst the nodes in C, and instead connects each of those nodes with v_C. Other structures (e.g., bi-cliques [10]) can be used instead as well.

Here, we propose a novel condensed representation, called C-DUP, that is better suited for our purposes. Given a graph extraction query, let $G(V, E)$ denote the output **expanded** graph; for clarity of exposition, we assume that G is a directed graph. We say $G_C(V', E')$ is an *equivalent C-DUP representation* if and only if:

(1) for every node $u \in V$, there are two nodes $u_s, u_t \in V'$ – the remaining nodes in V' are called *virtual nodes*;

(2) G_C is a directed *acylic* graph, i.e., it has no directed cycle;

(3) in G_C, there are no incoming edges to $u_s \forall u \in V$ and no outgoing edges from $u_t \forall u \in V$;

(4) for every edge $\langle u \rightarrow v\rangle \in E$, there is at least one directed path from u_s to v_t in G_C.

Figure 5 shows two examples of such condensed graphs. In the second case, where a heterogeneous bipartite graph is being extracted, there are no outgoing edges from $s[123]_s$ or incoming edges to $i[12]_t$, since the output graph itself only has edges from $i\_$ nodes to $s\_$ nodes.

Although we assume there are two copies of each real node in G_C here, the physical representation of G_C only requires one copy (with special-case code to handle incoming and outgoing edges).

Duplication Problem: The above definition allows for multiple paths between u_s and v_t, since that's the natural output of the extraction process below. *Any* graph algorithm whose correctness depends solely on the connectivity structure of the graph (we call these "duplicate-insensitive" algorithms), can be executed directly on top of this representation, with a potential for *speedup* (e.g., connected components or breadth-first search). However, this *duplication* causes correctness issues on all non duplicate-insensitive graph algorithms. The duplication problem entails that programmatically, when each real node tries to iterate over its neighbors, passing through its obligatory virtual neighbors, it may encounter the same neighbor *more than once*; this indicates a duplicate edge. The set of algorithms we propose in Section 5 are geared towards dealing with this duplication problem.

Single-layer vs Multi-layer Condensed Graphs: A condensed graph may have one or more layers of virtual nodes (formally, a condensed graph is called *multi-layer* if it contains a directed path of length > 2). In the majority of cases, most of the joins involved in extracting these graphs will be simple key-foreign key joins, and high-output joins (which require use of virtual nodes) occur relatively rarely. Although our system can handle arbitrary multi-layer graphs, we also develop special algorithms for the common case of single-layer condensed graphs.

4.2 Extracting a Condensed Graph

The key idea behind constructing a condensed graph is to postpone certain joins. Here we briefly sketch our algorithm for making those decisions, extracting the graph, and postprocessing it to reduce its size.

Step 1: First, we translate the *Nodes* statements into SQL queries, and execute those against the database to load the nodes in memory. In the following discussion, we assume that for every node u, we have two copies u_s (for *source*) and u_t (*target*); physically we only store one copy.

Step 2: We consider each *Edges* statement in turn. Recall that the output of each such statement is a set of 2-tuples (corresponding to a set of edges between real nodes), and further that we assume the statement is acyclic and aggregation-free (cf. Section 3.3, Case 1). Without loss of generality, we can represent the statement as:

$Edges(ID1, ID2) : -R_1(ID1, a_1), R_2(a_1, a_2), ..., R_n(a_{n-1}, ID2)$

(two different relations, R_i and R_j, may correspond to the same database table). Generalizations to allow multi-attribute joins and selection predicates are straightforward.

For each join $R_i(a_{i-1}, a_i) \bowtie_{a_i} R_{i+1}(a_i, a_{i+1})$, we retrieve the number of distinct values, d, for a_i (the join attribute) from the system catalog (e.g., `n_distinct` attribute in the `pg_stats` table in PostgreSQL). If $|R_i||R_{i+1}|/d > 2(|R_i| + |R_{i+1}|)$, then we consider this a *high-output* join and mark it so.

Step 3: We then consider each subsequence of the relations without a high-output join, construct an SQL query corresponding to it, and execute it against the database. Let $l, m, ..., u$ denote the join attributes which are marked as *high-output*. Then, the queries we execute correspond to:

$res_1(ID1, a_l) : -R_1(ID1, a_1), ..., R_l(a_{l-1}, a_l)$,
$res_2(a_l, a_m) : -R_{l+1}(a_l, a_{l+1}), ..., R_m(a_{m-1}, a_m)$, ..., and
$res_k(a_u, ID2) : -R_{u+1}(a_u, a_{u+1}), ..., R_n(a_{n-1}, ID2)$.

Step 4: For each join attribute $attr \in \{l, m, ..., u\}$, we create a set of virtual nodes corresponding to all possible values $attr$ takes.

Step 5: For $(x, y) \in res_1$, we add a directed edge from a real node to a virtual node: $x_s \rightarrow y$. For $(x, y) \in res_k$, we add a directed edge $x \rightarrow y_t$. For all other res_i, for $(x, y) \in res_i$, we add an edge between two virtual nodes: $x \rightarrow y$.

Step 6 (Pre-processing): For a virtual node, let in and out denote the number of incoming and outgoing edges respectively; if $in \times out \leq 6$, we "expand" this node, i.e., we remove it and add direct edges from its in-neighbors to its out-neighbors. This pre-processing step can have a significant impact on memory consumption. We have implemented a multi-threaded version of this to exploit multi-core machines, which resulted in several non-trivial concurrency issues. We omit a detailed discussion for lack of space.

If the query contains multiple *Edges* statements, the final constructed graph would be the union of the graphs constructed for each of them.

It is easy to show that the constructed graph satisfies all the required properties listed above, that it is equivalent to the output graph, and it occupies no more memory than loading all the input tables into memory (under standard independence assumptions).

In the example shown in Figure 5a, the graph specified in query `[Q3]` that is extracted assumes that all three of the joins involved portray low selectivity, and so we choose not to hand any of them to the database, but extract the condensed representation by instead projecting the tables in memory and creating intermediate virtual nodes for each unique value of each join condition.

4.3 In-Memory Representations

Next, we propose a series of in-memory graph representations that can be utilized to store the condensed representation mentioned above in its *de-duplicated* state. We focus on the implementation of the *getNeighbors()* iterator, which underlies most graph algorithms.

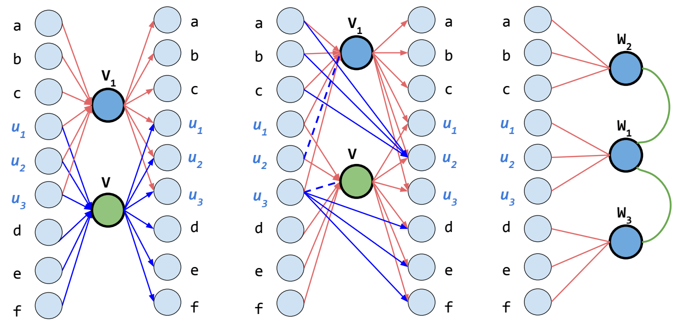
C-DUP: Condensed Duplicated Representation: This is the representation that we initially extract from the relational database, which suffers from the edge duplication problem. We can utilize this representation as-is by employing a naive solution to de-duplication, i.e., by doing de-duplication *on the fly* as algorithms are being executed. Specifically, when we call *getNeighbors(u)*, it starts a depth-first traversal from u_s and returns all the real nodes ($\_t$ nodes) reachable from u_s; it also keeps track of which neighbors have already been seen (in a *hashset*) and skips over them if the neighbor is seen again.

This is typically the most memory-efficient representation, does not require any pre-processing overhead, and is a good option for graph algorithms that access a small fraction of the graph (e.g., if we were looking for information about a small number of specific nodes). On the other hand, due to the required hash computations at every call, the execution penalty for this representation is high, especially for multi-layer graphs; it also suffers from memory and garbage collection bottlenecks for algorithms that require processing all the nodes in the graph. Operations like `deleteEdge()` are also quite involved in this representation, as deletion of a logical edge may require non-trivial modifications to the virtual nodes.

EXP: Fully Expanded Graph: On the other end of the spectrum, we can choose to expand the graph in memory, i.e., create all direct edges between all the real nodes in the graph and remove the virtual nodes. The expanded graph typically has a much larger memory footprint than the other representations due to the large number of edges. It is nevertheless, naturally, the most efficient representation for operating on, since iteration only requires a sequential scan over one's direct neighbors. The expanded graph is the baseline that we use to compare the performance of all other representations in terms of trading off memory with operational complexity.

DEDUP-1: Condensed De-duplicated Representation: This representation is similar to C-DUP and uses virtual nodes, but the major difference that it does not suffer from duplicate paths. This representation typically sits in the middle of the spectrum between EXP and C-DUP in terms of both memory efficiency and iteration performance; it usually results in a larger number of edges than C-DUP, but does not require on-the-fly de-duplication, thus significantly decreasing the overhead of neighbor iteration. The trade-offs here also include the one-time cost of removing duplication; de-duplicating a graph while minimizing the number of edges added can be shown to be NP-Hard. Unlike the other representations discussed below, this representation maintains the simplicity of C-DUP and can easily be serialized and used by other systems which need to simply implement a proper iterator.

BITMAP: De-duplication using Bitmaps: This representation results from applying a different kind of pre-processing based on maintaining *bitmaps*, for *filtering* out duplicate paths between nodes. Specifically, a vritual node V may be associated with a set of bitmaps, indexed by the IDs of the real nodes; the size of each bitmap is equal to the number of outgoing edges from V. Consider a depth-first traversal starting at u_s that reaches V. We check to see if there



(a) **C-DUP** (24 Edges) (b) **DEDUP1** (32 Edges) (c) **DEDUP2** (22 Edges)

Figure 6: The resulting graph after the addition of virtual node V. (c) shows the resulting graph for if we added edges *between* virtual nodes (we omit $\_s$ and $\_t$ subscripts since they are clear from the context).

is a bitmap corresponding to u_s; if not, we traverse each of the outgoing edges in turn. However, if there is indeed a bitmap corresponding to u_s, then we consult the bitmap to decide which of the outgoing edges to skip (i.e., if the corresponding bit is set to 1, we traverse the edge). In other words, the bitmaps are used to eliminate the possibility of reaching the same neighbor twice.

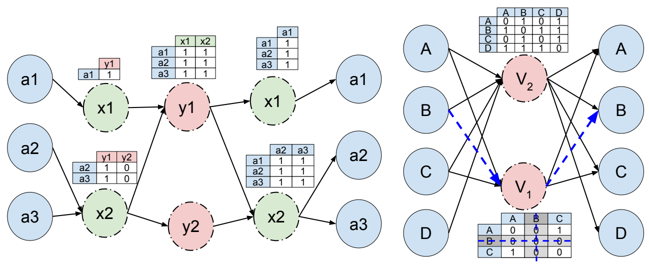
The main drawback of this representation is the memory overhead and complexity of storing these bitmaps, which also makes this representation less *portable* to systems outside GraphGen. The pre-processing required to set these bitmaps can also be quite involved as we discuss in the next section.

DEDUP-2: Optimization for Single-layer Symmetric Graphs: This optimized representation can significantly reduce the memory requirements for dense graphs, for the special case of a single-layer, *symmetric* condensed graph (i.e., $\langle u_s \rightarrow v_t \rangle \implies \langle v_s \rightarrow u_t \rangle$); many graphs satisfy these conditions. In such a case, for a virtual node V, if $u_s \rightarrow V$, then $V \rightarrow u_t$, and we can omit the $\_t$ nodes and associated edges. Figure 6 illustrates an example of the same graph if we were to use all three de-duplication representations. In C-DUP, we have two virtual nodes V_1 and V_2, that are both connected to a large number of real nodes. The optimal DEDUP-1 representation (Figure 6b) results in a substantial increase in the number of edges, because of the large number of duplicate paths. The DEDUP-2 representation (Figure 6c) uses *special* undirected edges between virtual nodes to handle such a scenario. A real node u is considered to be connected to all real nodes that it can reach through each of its direct neighboring virtual nodes v, *as well as* the virtual nodes directly connected to v (i.e. 1 hop away); e.g., node a is connected to b and c through W_2, and to u_1, u_2, u_3 through W_1 (which is connected to W_2), but not to d, e, f (since W_3 is not connected to W_2). This representation is required to be duplicate-free, i.e., there can be at most one such path between a pair of nodes. The DEDUP-2 representation here requires 11 undirected edges, which is just below the space requirements for C-DUP. However, for dense graphs, the benefits can be substantial as we discuss in Section 6.

Generating a good DEDUP-2 representation for a given C-DUP graph is much more intricate than generating a DEDUP-1 representation. Due to space constraints, we omit the details of the algorithm.

5. PRE-PROCESSING & DE-DUPLICATION

In this section, we discuss a series of pre-processing and de-duplication algorithms we have developed for constructing the different in-memory representations for a given query. The input to



(a) Multi-layer Graph (b) Single-layer Graph

Figure 7: Using *BITMAP*s to handle duplication.

all of these algorithms is the C-DUP representation, that has been extracted and instantiated in memory. We first present a general pre-processing algorithm for the BITMAP representation for multi-layer condensed graphs. We then discuss a series of optimizations for single-layer condensed graphs, including de-duplication algorithms that attempt to eliminate duplication (i.e., achieve DEDUP-1 representation).

We also describe the runtime complexity for each algorithm in which we refer to n_r as the number of real nodes, n_v as the number of virtual nodes, k as the number of layers of virtual nodes, and d as the maximum degree of any node (i.e., the maximum number of outgoing edges).

5.1 Pre-Processing for BITMAP

Recall that the goal of the pre-processing phase here is to associate and initilize bitmaps with the virtual nodes to avoid visiting the same real node twice when iterating over the out-neighbors of a given real node. We begin with presenting a simple, naive algorithm for setting the bitmaps; we then analyze the complexity of doing so optimally and present a set cover-based greedy algorithm.

5.1.1 BITMAP-1 Algorithm

This algorithm only associates bitmaps with the virtual nodes in the penultimate layer, i.e., with the virtual nodes that have outgoing edges to $\_t$ nodes. We iterate over all the real nodes in turn. For each such node u, we initiate a depth-first traversal from u_s, keeping track of all the real nodes visited during the process using a hashset, H_u. For each virtual node V visited, we check if it is in the penultimate layer; if yes, we add a bitmap of size equal to the number of outgoing edges from V. Then, for each outgoing edge $V \rightarrow v_t$, we check if $v_t \in H_u$. If yes, we set the corresponding bit to 0; else, we set it to 1 and add v_t to H_u.

This is the least computationally complex of all the algorithms, and in practice the fastest algorithm. It maintains the same number of edges as C-DUP, while adding the overhead of the bitmaps and the appropriate indexes associated with them for each virtual node. The traversal order in which we process each real node does not matter here since the end result will always have the same number of edges as C-DUP. Changing the processing order only changes the way the **set** bits are distributed among the bitmaps.

Complexity: The worst-case runtime complexity of this algorithm is $O(n_r * d^{k+1})$. Although this might seem high, we note that this is always lower than the cost of expanding the graph.

5.1.2 Formal Analysis

The above algorithm, while simple, tends to initialize and maintain a large number of bitmaps. This leads us to ask the question: how can we achieve the required deduplication while using the minimum number of bitmaps (or minimum total number of bits)? This seemingly simple problem unfortunately turns out to be NP-Hard, even for single-layer graphs. In a single-layer condensed graph, let u denote a real node, with edges to virtual nodes $V_1, ..., V_n$, and let $O(V_1)$ denote the set of real nodes to which V_1 has outgoing edges. Then, the problem of identifying a minimum set of bitmaps to maintain is equivalent to finding a *set cover* where our goal is to find a subset of $O(V_1), ..., O(V_n)$ that covers their union. Unfortunately, the set cover problem is not only NP-Hard, but is also known to be hard to approximate.

5.1.3 BITMAP-2 Algorithm

This algorithm is based on the standard *greedy algorithm* for set cover, which is known to achieve the best approximation ratio ($O(\log n)$) for the problem. We describe it using the terminology above for single-layer condensed graphs. The algorithm starts by picking the virtual node V_i with the largest $|O(V_i)|$. It adds a bitmap for u to V_i, and sets it to all 1s; all nodes in $O(V_i)$ are now considered to be *covered*. It then identifies the virtual node V_j with the largest $|O(V_j) - O(V_i)|$, i.e., the virtual node that connects to largest number of nodes that remain to be covered. It adds a bitmap for u_s to V_j and sets it appropriately. It repeats the process until all the nodes that are reachable from u_s have been covered. For the remaining virtual nodes (if any), the edges from u_s to those nodes are simply deleted since there is no reason to traverse those.

We generalize this basic algorithm to multi-layer condensed graphs by applying the same principle at each layer. Let $V_1^1, ..., V_n^1$ denote the set of virtual nodes pointed to by u_s. Let $N(u_s)$ denote all the real $\_t$ nodes reachable from u_s. For each V_i^1, we count how many of the nodes in $N(u_s)$ are reachable from V_i^1, and explore the virtual node with the highest such count first. At the penultimate layer, the algorithm reduces to the single-layer algorithm described above and appropriately sets the bitmaps. At all points, we keep track of how many of the nodes in $N(u_s)$ have been covered so far, and use that for making the decisions about which bits to set. So after bitmaps have been set for all virtual nodes reachable from V_1^1, if there are still nodes in $N(u_s)$ that need to be covered, we pick the virtual node V_i^1 that reaches the largest number of uncovered nodes, and so on.

One difference here is that we never delete an outgoing edge from a virtual node, since it may be needed for another real node. Instead, we use bitmaps to stop traversing down those paths (e.g., edge $x_2 \rightarrow y_2$ in Figure 7).

Our implementation exploits multi-core parallelism, by creating equal-sized chunks of the set of real nodes, and processing the nodes in each chunk in parallel.

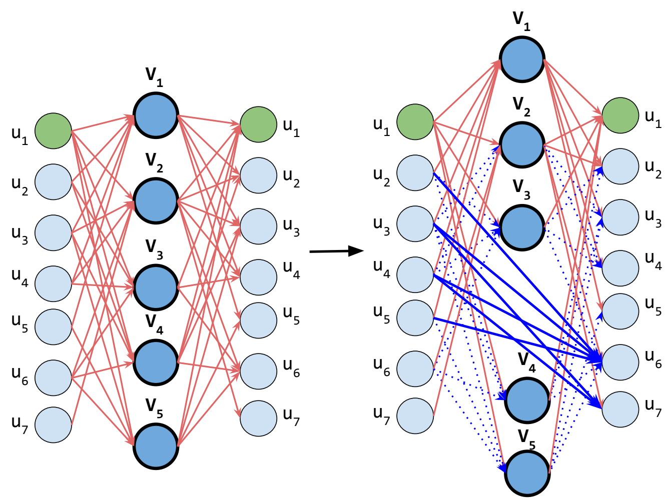
Complexity: The runtime complexity of this algorithm is significantly higher than BITMAP-1 because of the need to re-compute the number of reachable nodes after each choice, and the worst-case complexity could be as high as: $O(n_r * d^{2^k})$. In practice, k is usually 1 or 2, and the algorithm finishes reasonably quickly, especially given our parallel implementation.

5.2 De-duplication for DEDUP-1

The goal with de-duplication is to modify the initial C-DUP graph to reach a state where there is at most one unique path between any two real nodes in the graph. We describe a series of novel algorithms for achieving this for single-layer condensed graphs, and discuss the pros and cons of using each one as well as their effectiveness in terms of the size of the resulting graph. We briefly sketch how these algorithms can be extended to multi-layer condensed graphs; however, we leave a detailed study of de-duplication for multi-layer graphs to future work.

5.2.1 Single-layer Condensed Graphs

The theoretical complexity of this problem for single-layer condensed graphs is the same as the original problem considered by



(a) *44* Edges

$N(V_1)=\{u_1,u_2,u_3,u_4,u_5\}$
$N(V_2)=\{u_1,u_3,u_4,u_6,u_7\}$
$N(V_3)=\{u_1,u_2,u_4,u_6\}$
$N(V_4)=\{u_1,u_2,u_3,u_6\}$
$N(V_5)=\{u_1,u_3,u_5,u_6\}$

(b) *34* Edges

$N(V_1)=\{u_1,u_2,u_3,u_4,u_5\}$
$N(V_2)=\{u_1,\cancel{u_3},\cancel{u_4},u_6,u_7\}$
$N(V_3)=\{u_1,\cancel{u_2},\cancel{u_4},\cancel{u_6}\}$
$N(V_4)=\{u_1,\cancel{u_2},\cancel{u_3},\cancel{u_6}\}$
$N(V_5)=\{u_1,\cancel{u_3},\cancel{u_5},\cancel{u_6}\}$

Figure 8: Deduplicating u_1 using the "real-nodes first" algorithm, resulting to an equivalent graph with a *smaller* number of edges

Feder and Motwani [14], which focuses on the reverse problem of *compressing cliques* that exist in the expanded graph. Although the expanded graph is usually very large, it is still only $O(n^2)$, so the NP-Hardness of the de-duplication problem is the same. However, those algorithms presented in [14] are not applicable here because the input representation is different, and expansion is not an option. We present four algorithms for this problem.

In the description below, for a virtual node V, we use $I(V)$ to denote the set of real nodes that point to V, and $O(V)$ to denote the real nodes that V points to.

Naive Virtual Nodes First: This algorithm de-duplicates the graph one virtual node at a time. We start with a graph containing only the real nodes and no virtual nodes, which is trivially duplication-free. We then add the virtual nodes one at a time, always ensuring that the partial graph remains free of any duplication.

When adding a virtual node V: we first collect all of the virtual nodes R_i such that $I(V) \cap I(R_i) \neq \phi$; these are the virtual nodes that the real nodes in $I(V)$ point to. Let this set be R. A $processed$ set is also maintained which keeps track of the virtual nodes that have been added to the current partial graph. For every virtual node $R_i \in R \cap processed$, if $|O(V) \cap O(R_i)| > 1$, we modify the virtual nodes to handle the duplication before adding V to the partial graph (If there is no such R_i, we are done). We select a real node $r \in O(V) \cap O(R_i)$ at *random*, and choose to either remove the edge $(V \rightarrow r)$ or $(R_i \rightarrow r)$, depending on the indegrees of the two virtual nodes. The intuition here is that, by removing the edge from the lower-degree virtual node, we have to add fewer direct edges to compensate for removal of the edge. Suppose we remove the former $(V \rightarrow r)$ edge. We then add direct edges to r from all the real nodes in $I(V)$, while checking to make sure that r is not already connected to those nodes through other virtual nodes. Virtual node V is then added to a $processed$ set and we consider the next virtual node.

Complexity: The runtime complexity is $O(n_v * d^4)$.

Naive Real Nodes First: In this approach, we consider each *real node* in the graph at a time, and handle duplication between the virtual nodes it is connected to, in the order in which they appear in its neighborhood. This algorithm handles de-duplication between two virtual nodes that overlap in exactly the same way as the one described above. It differs however in that it entirely handles *all duplication* between a single real node's virtual neighbors before moving on to processing the next real node. As each real node is handled, its virtual nodes are added to a $processed$ set, and every new virtual node that comes in is checked for duplication against the rest of the virtual nodes in this $processed$ set. This $processed$ set is however limited to the virtual neighborhood of the real node that is currently being de-duplicated, and is cleared when we move on to the next real node.

Complexity: The runtime complexity is $O(n_r * d^4)$.

Greedy Real Nodes First Algorithm: In this less naive but still greedy approach, we consider each real node in sequence, and de-duplicate it individually. Figure 8 shows an example, that we will use to illustrate the algorithm. The figure shows a real node u_1 that is connected to 5 virtual nodes, with significant duplication, and a de-duplication of that node. Our goal here is to ensure that there are no duplicate edges involving u_1 – we do not try to eliminate all duplication among all of u_1's virtual nodes like in the naive approach. The core idea of this algorithm is that we consult a heuristic to decide whether to remove an edge to a virtual node and add the missing direct edges, or to keep the edge to the virtual node.

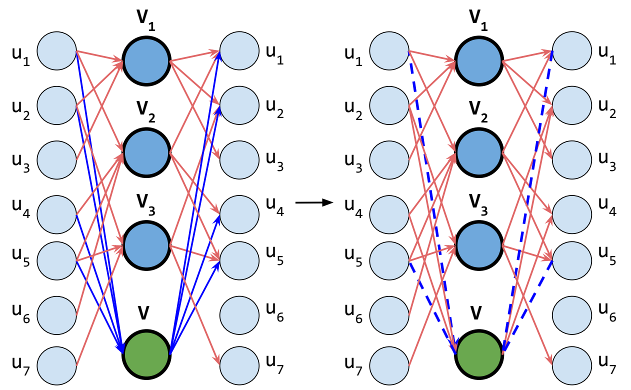
Let $\mathcal{V}'$ denote the set of virtual nodes to which u_s remains connected after deduplication, and $\mathcal{V}''$ denote the set of virtual nodes from which u_s is disconnected; also, let E denote the direct edges that we needed to add from u_s during this process. Our goal is to minimize the total number of edges in the resulting structure. This problem can be shown to be NP-Hard using a reduction from the *exact set cover* problem (see the extended version of the paper for details).

We present a heuristic inspired by the standard greedy set cover heuristic which works as follows. We initialize $\mathcal{V}' = \varnothing$, and $\mathcal{V}'' = \mathcal{V}$; we also logically add direct edges from u_s to all its neighbors in $N(u_s)$, and thus $E = \{(u_s, x) | x \in \cup_{V \in \mathcal{V}} O(V)\}$. We then move virtual nodes from $\mathcal{V}''$ to $\mathcal{V}'$ one at a time. Specifically, for each virtual node $V \in \mathcal{V}''$, we consider moving it to $\mathcal{V}'$. Let $X = \cup O(\mathcal{V}')$ denote the set of real nodes that u is connected to through $\mathcal{V}'$. In order to move V to $\mathcal{V}'$, we must disconnect V from all nodes in $V \cap X$ – otherwise there would be duplicate edges between u and those nodes. Then, for any $a, b \in V \cap X$, we check if any other virtual node in $\mathcal{V}''$ is connected to *both* a and b – if not, we must add the direct edge (a, b). Finally, for $r_i \in V - V \cap X$, we remove all direct edges (u, r_i).

The benefit of moving the virtual node V from $\mathcal{V}''$ to $\mathcal{V}'$ is computed as the *reduction* in the total number of edges in every scenario. We select the virtual node with the highest benefit (> 0) to move to $\mathcal{V}'$. If no virtual node in $\mathcal{V}''$ has benefit > 0, we move on to the next real node and leave u connected to its neighbors through direct edges.

Complexity: The runtime complexity here is roughly $O(n_r * d^5)$.

Greedy Virtual Nodes First Algorithm: Exactly like the naive version above, this algorithm de-duplicates the graph one virtual node at a time, maintaining a de-duplicated partial graph at every step. We start with a graph containing only the real nodes and no virtual nodes, which is trivially de-duplicated. We then add the virtual nodes one at a time, always ensuring that the partial graph does not have any duplication. Let V denote the virtual node under consideration. Let $\mathcal{V} = \{V_1, ..., V_n\}$ denote all the virtual nodes that share at least 2 real nodes with V (i.e., $|O(V) \cap O(V_i)| \geq 2$). Let $C_i = O(V) \cap O(V_i)$, denote the real nodes to which both V and



(a) *28* Edges (b) *24* Edges

$N(V_1)=\{u_1,u_2,u_3\}$ $N(V_1)=\{u_1,u_2,u_3\}$
$N(V_2)=\{u_1,u_4,u_5,u_6\}$ $N(V_2)=\{u_1,u_4,u_5,u_6\}$
$N(V_3)=\{u_2,u_5,u_7\}$ $N(V_3)=\{u_2,u_5,u_7\}$
$N(V)=\{u_1,u_2,u_4,u_5\}$ $N(V)=\{\cancel{u_1},u_2,u_4,\cancel{u_5}\}$

Figure 9: De-duplication using the greedy virtual nodes first

V_i are connected. At least $|C_i|-1$ of those edges must be removed from V and V_i combined to ensure that there is no duplication.

The special case of this problem where $|C_i| = 2, \forall i$, can be shown to be equivalent to finding a *vertex cover* in a graph (we omit the proof due to space constraints). We again adopt a heuristic inspired by the greedy approximation algorithm for vertex cover. Specifically, for each node in C_i, we compute the *cost* and the $benefit$ of removing it from any V_i versus from V. The *cost* of removing the node is computed as the number of direct edges that need to be *added* if we remove the edge to that virtual node, whereas the $benefit$ is computed as the *reduction* in the total number of nodes in the intersection with V_i ($\Sigma|C_i|$) (removing the node from V_i always yields a $benefit$ of 1, whereas removing it from V may have a higher benefit). We then make a more informed decision and choose to remove an edge from a real node rn that leads to the overall highest $benefit/cost$ ratio.

Complexity: The runtime complexity here is: $O(n_v d(n_v d^2 + d))$. We note that, these complexity bounds listed here make worst-case assumptions and in practice, most algorithms run much faster.

5.2.2 *Multi-layer Condensed Graphs*

Deduplicating multi-layer condensed graphs turns out to be significantly trickier and computationally more expensive than single-layer graphs. In single layer graphs, identifying duplication is relatively straightforward; for two virtual nodes V_1 and V_2, if $O(V_1) \cap O(V_2) \neq \phi$ and $I(V_1) \cap I(V_2) \neq \phi$, then there is duplication. We keep the neighbor lists in sorted order, thus making these checks very fast. However, for multi-layer condensed graphs, we need to do expensive depth-first traversals to simplify identify duplication.

We can adapt the naive virtual nodes first algorithm described above to the multi-layer case as follows. We (conceptually) add a dummy node s to the condensed graph and add directed edges from s to the $\_s$ copies of all the real nodes. We then traverse the graph in a depth-first fashion, and add the virtual nodes encountered to an initially empty graph one-by-one, while ensuring no duplication. However, this algorithm turned out to be infeasible to run even on small multi-layer graphs, and we do not report any experimental results for that algorithm. Instead, we propose using either the BITMAP-2 approach for multi-layer graphs, or first converting it into a single-layer graph and then using one of the algorithms developed above.

6. EXPERIMENTAL STUDY

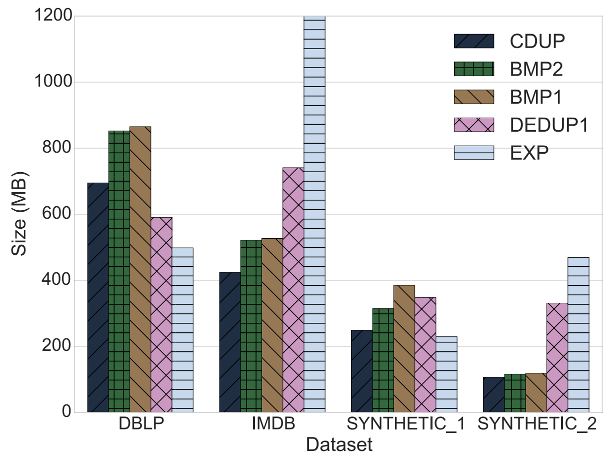


Figure 10: Memory footprint of each representation for each of the explored datasets.

Dataset	Real Nodes	Virt Nodes	Avg Size	EXP Edges
DBLP	523,525	410,000	2	1,493,526
IMDB	439,639	100,000	10	10,118,354
Synthetic_1	20,000	200,000	7	2,032,523
Synthetic_2	200,000	1000	94	4,135,768

Table 2: Small Datasets: **avg size** refers to the average number of real nodes contained in a virtual node

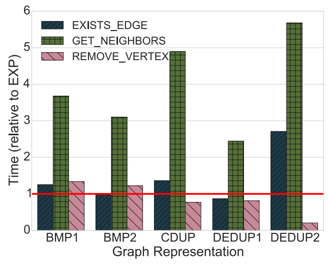
In this section, we provide a comprehensive experimental evaluation of GraphGen using several real-world and synthetic datasets. We first present a detailed study using 4 small datasets, including micro-benchmarks that showcase the performance of each of the aforementioned representations on a few basic graph operations, and on three common graph algorithms. We then compare the performance of the different de-duplication algorithms. We then present an analysis using much larger datasets, but for a smaller set of representations. All the experiments were run on a single machine with 24 cores running at 2.20GHz, and with 64GB RAM.

6.1 Small Datasets

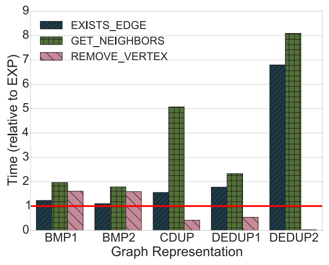
First we present a detailed study using 4 relatively-small datasets. We use representative samples of the *DBLP* and *IMDB* datasets in our study (Table 2), extracting *co-author* and *co-actor* graphs respectively. We also generated a series of synthetic graphs so that we can better understand the differences between the representations and algorithms on a wide range of possible datasets, with varying numbers of real nodes and virtual nodes, and varying degree distributions and densities. Since we need the graphs in a condensed representation, we cannot use any of the existing random graph generators for this purpose. Instead, we built a synthetic graph generator that aims to generate graphs consistent with the the Barabàsi–Albert model [7] (also called the *preferential attachment model*). We omit a detailed description of our generator due to lack of space.

6.1.1 *Memory Footprint*

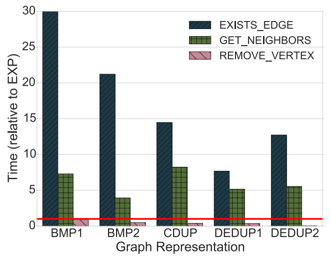
Due to the dynamic nature of the JVM and lack of effective user-control mechanisms over garbage collection, it is difficult to measure memory consumption of a Java object accurately. The `System.gc()` call simply hints to the JVM that it may want to call the garbage collector (GC), but the actual decision on whether or not to call the GC is ultimately made internally by the JVM. We anecdotally confirmed that the `System.gc()` call works as expected, and initiates garbage collection the majority of the time. For our purposes, we therefore take this simple approach of calling



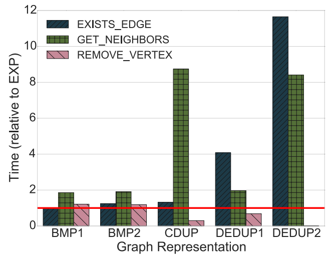
(a) DBLP



(b) IMDB



(c) Synthetic_1



(d) Synthetic_2

Figure 11: Microbenchmarks for each representation

`System.gc()` and calling `Thread.sleep()` for a small period of time (500ms), thus attempting to trigger the GC. We do this 5 times consecutively, record the used heap size after each call, and finally keep the minimum recorded value. We also take care to *only* have the graph object loaded in the current process at that time.

Figure 10 shows the memory footprint for each individual representation in each dataset; the algorithm used for *DEDUP-1* was Greedy Virtual Nodes First, described in Section 5.2.1. When the average degree of virtual nodes is small and there is a large number of virtual nodes (*DBLP* and *Synthetic_1*), we observe that there is a relatively small difference in the size of the condensed and expanded graphs, and de-duplication (*DEDUP1* and *DEDUP2*) actually results in an even smaller footprint graph. The overhead of the bitmaps in the *BITMAP* representation also shows up heavily in these datasets.

On the other hand, the *IMDB* dataset shows a 6-fold difference in size between EXP and C-DUP and over a 2-fold difference with all other representations. Here, the BITMAP representation works very well relative to the edge explosion of EXP. *Synthetic_2* portrays the amount of compression possible in graphs with very large, overlapping cliques. The BITMAP representations prevail here as well; however this dataset also shows how the *DEDUP2* representation can be significantly more compact than *DEDUP1*, while maintaining its natural, more portable structure compared to the BITMAP representations.

We report memory footprints for larger datasets in Section 6.2.

6.1.2 Microbenchmarks

We conducted a complete set of micro-benchmarks on each function in our Graph API described in Section 4. Figure 11 shows the results for some of the more interesting graph operations. The results shown are normalized using the values for the full *EXP* representation, which typically is the fastest and is used as the baseline. Since most of these operations take micro-seconds to complete, to ensure validity in the results, the metrics shown are the result of the mean of 3000 repetitions for each operation, on a specific set of the same 3000 randomly selected nodes for each dataset.

Iteration through each real node's neighbors via the GETNEIGHBORS() method is naturally more expensive on all other representations compared to the expanded graph. This portrays the natural tradeoff of extraction latency and memory footprint versus performance that is offered by these representations. *DEDUP2* is usually least performant here because of the extra layer of indirection that this representation introduces. *DEDUP1* is typically more performant than the *BITMAP* representations in datasets where there is a large number of small cliques.

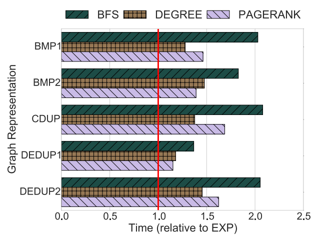
In terms of the EXISTSEDGE() operation, we have included auxiliary indices in both virtual and real vertices, which allow for rapid checks on whether a logical edge exists between two real nodes. Latency in this operation is relative to the total number of virtual nodes, the indexes of which need to be checked. The REMOVEVERTEX() operation is actually *more efficient* on the CDUP, DEDUP1 and DEDUP2 representations than EXP. In order for a vertex to be removed from the graph, explicit removal of all of its edges is required. In representations like DEDUP1 and DEDUP2, that employ virtual nodes, we need to remove a smaller number of edges on average in the removal process. DEDUP2 is most interesting here because a real node is always connected to only 1 virtual node, therefore the removal cost is constant.

6.1.3 Graph Algorithms Performance

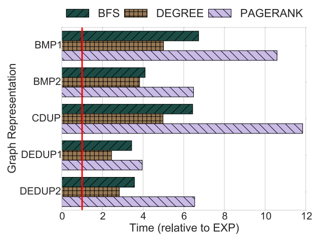
Figure 12 shows the results of running 3 different graph algorithms on the different in-memory representations. We compared the performance of *Degree* calculation, *Breadth First Search* starting from a single node, as well as *PageRank* on the entire graph. Again, the results shown are normalized to the values for the full EXP representation. Degree and PageRank were implemented and run on our custom vertex-centric framework described in Section 3.4, while BFS was run in a single threaded manner starting from a single random node in the graph, using our Graph API to operate directly on top of each of the representations. Again, the Breadth first search results are the mean of runs on a specific set of 50 randomly selected real nodes on all of the representations, while the PageRank are an average of 10 runs. As we can see, *BFS* and PageRank both follow the trends of the micro-benchmarks in terms of differences in performance between representations.

For IMDB and Synthetic_2, both of which yield very large expanded graphs, we observed little to no overhead in real world performance compared to EXP when actually running algorithms on top of these representations, especially when it comes to the BITMAP and DEDUP1 representations (we omit these graphs). DBLP and Synthetic_1 datasets portray a large gap in performance compared to EXP; this is because these datasets consist of a large number of small virtual nodes, thus increasing the average number of virtual nodes that need to be iterated over for a single calculation. This is also the reason why DEDUP1 and BITMAP2 typically perform better; they feature a smaller number of virtual neighbors per real node than representations like C-DUP and BMP1, and sometimes DEDUP2 as well.

6.1.4 Comparing De-duplication Algorithms

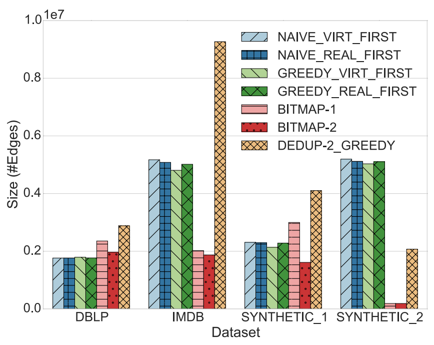


(a) DBLP

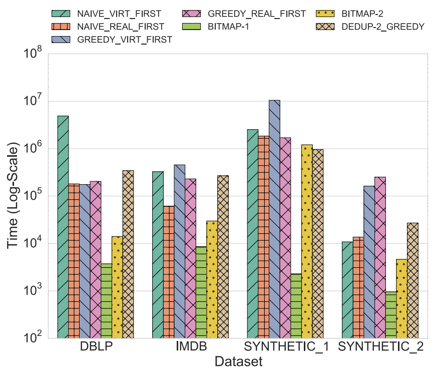


(b) Synthetic_1

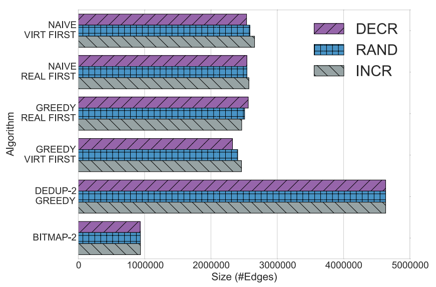
Figure 12: Performance of Graph Algorithms on Each Representation for two datasets



(a) Performance comparison of the de-duplication algorithms in terms of number of edges in output graph. Random (RAND) vertex ordering was used where applicable.



(b) De-duplication time comparison between algorithms. Random (RAND) vertex ordering was used where applicable.



(c) Small variations caused by node ordering in de-duplication

Figure 13: Deduplication Performance Results

Dataset	CDUP				BMP-DEDUP					EXP			
	Degree	PR	BFS	**Mem (GB)**	Degree	PR	BFS	**Mem (GB)**	Dedup Time	Degree	PR	BFS	**Mem (GB)**
Layered_1	40	1211	382	**1.421**	30	1025	284	**2.737**	1714	DNF	DNF	DNF	**>64**
Layered_2	12	397	129	**1.613**	10	339	111	**2.258**	553	11	83	85	**19.798**
Single_1	2	30	0.01	**1.276**	1.8	25	0.02	**1.493**	10.4	1.6	14.7	0.01	**1.2**
Single_2	202	DNF	1.3	**9.901**	81	3682	.12	**13.042**	5871	DNF	DNF	DNF	**>65**
TPCH	3.5	58	86	**.023**	0.4	6	8.6	**.049**	1207	1.470	8	16	**7.398**

Table 3: Comparing the performance of C-DUP, BMP, and EXP on large datasets: the table shows the running times (in seconds) for three graph algorithms, and total memory consumption (in GB); the table also shows the time required for bitmap de-duplication

Finally, we compare the various de-duplication algorithms presented in Section 5. Figure 13a compares the number of edges in the resulting graph after running the different de-duplication algorithms. As we can see, the differences between the different DEDUP-1 algorithms are largely minor, with the Virtual Nodes First Greedy algorithm having a slight edge on most datasets. The comparisons across different representations mirror the relative memory footprint performance (Figure 10), with the main difference being the overheads associating with bitmaps in BITMAP representations that are not counted here.

Figure 13b shows the running times for the different algorithms (on a log-scale). As expected, BITMAP-1 is the fastest of the algorithms, whereas the DEDUP-1 and DEDUP-2 algorithms take significantly more time. We note however that de-duplication is a one-time cost, and the overhead of doing so may be acceptable in many cases, especially if the extracted graph is serialized and repeatedly analyzed over a period of time. Finally, Figure 13c shows how the performance of the various algorithms varies depending on the processing order. We did not observe any noticeable differences or patterns in this performance, and recommend using the random ordering for robustness.

6.2 Large Datasets

To reason about the practicality and scalability of GRAPHGEN, we evaluated its performance on a series of datasets that yielded larger and denser graphs (Table 3). Datasets *Layered_1* and *Layered_2* are synthetically generated multi-layer condensed graphs, while Single_1, Single_2 are standard single-layer condensed graphs. These graphs were derived from joins on synthetically generated tables, where the selectivity of the join condition attributes were tweaked accordingly. At this scale, only the C-DUP, BMP-2, and EXP are typically feasible options, since none of the de-duplication algorithms (targetting DEDUP-1 or DEDUP-2) run in a reasonable time.

Comparing the memory consuption, we can see that we were not able to expand the graph in 2 of the cases, since it consumed more memory than available ($> 64GB$); in the remaining cases, we see that EXP consumes more than 1 or 2 orders of magnitude more memory. In one case, EXP was actually smaller than C-DUP; our pre-processing phase (Section 4.2), which was not used for these experiments, would typically expand the graph in such cases. Runtimes of the graph algorithms show the patterns we expect, with EXP typically performing the best (if feasible), and BMP somewhere in between EXP and C-DUP (in some cases, with an order of magnitude improvement). Note that: we only show the base memory consumption for C-DUP – the memory consumption can be significantly higher when executing a graph algorithm because of on-the-fly de-duplication that we need to perform. In particular, C-DUP was not able to complete PageRank for Single_2, running out of memory.

As these experiments show, datasets don't necessarily have to be large in order to hide some very dense graphs, which would normally be extremely expensive to extract and analyze. This is shown in the TPCH dataset where we extracted a graph of customers who have bought the same item. With GRAPHGEN, we are able to load them into memory and with a small de-duplication cost, are able to achieve comparable iteration performance that allows users to explore, and analyze them in a fraction of the time, and using a fraction of the machine's memory that would be initially required.

7. CONCLUSION

In this paper, we presented GRAPHGEN, a system that enables users to analyze the implicit interconnection structures between entities in normalized relational databases, without the need to extract the graph structure and load it into specialized graph engines. GRAPHGEN can interoperate with a variety of graph analysis libraries and supports a standard graph API, breaking down the barriers to employing graph analytics. However, these implicitly defined graphs can often be orders of magnitude larger than the original relational datasets, and it is often infeasible to extract or operate upon them. We presented a series of in-memory condensed representations and de-duplication algorithms to mitigate this problem, and showed how we can efficiently run graph algorithms on such graphs while requiring much smaller amounts of memory. The choice of which representation to use depends on the specific application scenario, and can be made at a per dataset or per analysis level. The de-duplication algorithms that we have developed are of independent interest, since they generate a compressed representation of the extracted graph. Some of the directions for future work include tuning the selectivity estimates for complex extraction queries, and extending our de-duplication algorithms to handling general directed, heterogeneous graphs.

Acknowledgements I'd like to extend a big thanks to Prof. Amol Deshpande, for his contributions and guidance in this work.

8. REFERENCES

[1] Neo4j - native graph database. https://neo4j.com/.

[2] Orientdb - second generation distributed graph database. http://orientdb.com/orientdb/.

[3] Titan - distributed graph database. http://titan.thinkaurelius.com/.

[4] Demonstration proposal; anonymized for double-blind reviewing. 2015.

[5] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *arXiv preprint arXiv:1503.02368*, 2015.

[6] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[7] R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[8] Apache. Giraph. http://giraph.apache.org/.

[9] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proc. VLDB Endow.*, 8(2):161–172, 2014.

[10] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pages 95–106. ACM, 2008.

[11] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA*, 2009.

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

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

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

[15] J. Gao, J. Zhou, C. Zhou, and J. X. Yu. Glog: A high level graph analysis system using mapreduce. In *2014 IEEE 30th International Conference on Data Engineering*, pages 544–555. IEEE, 2014.

[16] J. Gonzalez, R. Xin, A. Dave, D. Crankshaw, M. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[17] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[18] T. J. Green, M. Aref, and G. Karvounarakis. *Logicblox, platform and language: A tutorial*. Springer, 2012.

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

[20] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph analytics using the Vertica relational database. *arXiv preprint arXiv:1412.5263*, 2014.

[21] 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.

[22] C. Karande, K. Chellapilla, and R. Andersen. Speeding up algorithms on compressed web graphs. *Internet Mathematics*, 2009.

[23] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[24] S. Lee, B. H. Park, S.-H. Lim, and M. Shankar. Table2graph: A scalable graph construction from relational tables using map-reduce. In *IEEE BigDataService*, 2015.

[25] J. Leskovec and R. Sosič. SNAP: A general purpose network analysis and graph mining library in C++. http://snap.stanford.edu/snap, June 2014.

[26] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[27] H. Ma, B. Shao, Y. Xiao, L. J. Chen, and H. Wang. G-sql: fast query processing via graph exploration. *Proceedings of the VLDB Endowment*, 9(12):900–911, 2016.

[28] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[29] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[30] T. Parr. Another tool for language recognition (antlr). http://www.antlr.org/.

[31] Y. Perez, R. Sosič, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec. Ringo: Interactive graph analytics on big-memory machines. In *SIGMOD*, 2015.

[32] M. Rudolf, M. Paradies, C. Bornhövd, and W. Lehner. The graph story of the sap hana database. In *BTW*, volume 13, pages 403–420. Citeseer, 2013.

[33] J. Seo, S. Guo, and M. S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 278–289. IEEE, 2013.

[34] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, volume 48, pages 135–146. ACM, 2013.

[35] D. Simmen, K. Schnaitter, J. Davis, Y. He, S. Lohariwala, A. Mysore, V. Shenoi, M. Tan, and Y. Xiao. Large-scale Graph Analytics in Aster 6: Bringing Context to Big Data Discovery. *PVLDB*, 7(13), 2014.

[36] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. SQLGraph: an efficient relational-based property graph store. In *SIGMOD*, 2015.

[37] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *Proceedings of the VLDB Endowment*, 9(5):420–431, 2016.