ABSTRACT

Title of dissertation: ENABLING GRAPH ANALYSIS OVER RELATIONAL DATABASES

Konstantinos Xirogiannopoulos
Doctor of Philosophy, 2019

Dissertation directed by: Professor Amol Deshpande
Department of Computer Science

Complex interactions and systems can be modeled by analyzing the connections between underlying entities or objects described by a dataset. These relationships form networks (graphs), the analysis of which has been shown to provide tremendous value in areas ranging from retail to many scientific domains. This value is obtained by using various methodologies from *network science*– a field which focuses on studying network representations in the real world. In particular "graph algorithms", which iteratively traverse a graph's connections, are often leveraged to gain insights. To take advantage of the opportunity presented by graph algorithms, there have been a variety of specialized graph data management systems, and analysis frameworks, proposed in recent years, which have made significant advances in efficiently storing and analyzing graph-structured data. Most datasets however currently do not reside in these specialized systems but rather in general-purpose relational database management systems (RDBMS). A relational or similarly structured system is typically governed by a schema of varying strictness that implements

constraints and is meticulously designed for the specific enterprise. Such structured datasets contain many relationships between the entities therein, that can be seen as latent or "hidden" graphs that exist inherently inside the datasets. However, these relationships can only typically be traversed via conducting expensive JOINs using SQL or similar languages. Thus, in order for users to efficiently traverse these latent graphs to conduct analysis, data needs to be transformed and migrated to specialized systems. This creates barriers that hinder and discourage graph analysis; our vision is to break these barriers.

In this dissertation we investigate the opportunities and challenges involved in efficiently leveraging *relationships* within data stored in structured databases. First, we present GraphGen, a lightweight software layer that is independent from the underlying database, and provides interfaces for graph analysis of data in RDBMSs. GraphGen is the first such system that introduces an intuitive high-level language for specifying graphs of interest, and utilizes in-memory graph representations to tackle the problems associated with analyzing graphs that are hidden inside structured datasets. We show GraphGen can analyze such graphs in orders of magnitude less memory, and often computation time, while eliminating manual Extract-Transform-Load (ETL) effort.

Second, we examine how in-memory graph representations of RDBMS data can be used to enhance relational query processing. We present a novel, general framework for executing GROUP BY aggregation over conjunctive queries which avoids materialization of intermediate JOIN results, and wrap this framework inside a multi-way relational operator called Join-Agg. We show that Join-Agg can

compute aggregates over a class of relational and graph queries using orders of magnitude less memory and computation time.

ENABLING GRAPH ANALYSIS OVER RELATIONAL DATABASES

by

Konstantinos Xirogiannopoulos

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2019

Advisory Committee:
Professor Amol Deshpande, Chair/Advisor
Professor Louiqa Raschid, Dean's Representative
Professor Daniel Abadi
Professor Peter Keleher
Professor Leilani Battle

Dedication

To my family

Acknowledgments

I would like to acknowledge my advisor, Amol Deshpande for his guidance ever since my arrival at UMD and for believing in my ability to publish in the top databases venues and get our paper submissions out on time. I'd also like to thank my committee members: Pete Keleher, Daniel Abadi, Louiqa Raschid and Leilani Battle for all their help and constructive feedback.

This has without a doubt been the most intellectually challenging and complex journey I've ever been on. My family and friends were by my side throughout the whole thing even though they were physically thousands of miles away. Thank you dad, mom and Peggy for talking me through and supporting me all the times I've been discouraged. Thank you Katie for being there for me– it's hard to imagine just how much more difficult the final stretch of this journey would have been without you.

Thank you Ben for all of the opportunities you presented me with, for our stress-relieving strolls through our beautiful campus and the delicious food we shared and/or made albeit sometimes requiring a (very) early rise. Thank you Allen for our intriguing in-depth discussions about academia, research, and obscure topics we're both fascinated by. Thank you Rebecca for the heartfelt and cathartic conversations and your unwavering support along the way. Thank you guys for all your help and feedback you provided on my work.

I would like to thank the Hellenic Graduate Student Association at UMD, who have been an integral part of my support group from the start. Thank you Ioannis

Demertzis, Leda Apergi and Sofia Nikolakaki for the great company and awesome parties we threw and attended together–keeping Greek traditions alive in College Park! I'd also like to thank Kostas Zampogiannis, Moschoula Pternea, Antonis Kyprianidis and the rest of "the Greeks" as well as Adi Hajj-Ahmad, always a part of our Greek shenanigans.

Last but not least, I'd like to thank the Computer Science Department staff: Tom Hurst, Jennifer Story, Stephanie Peters, Sharon McElroy. You all do an incredible job and every single one of us would be constantly anxious and lost without your help.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Program Interface
RDBMS	Relational Database Management System
ETL	Extract-Transform-Load
BI	Business Intelligence
SQL	Standard Query Language
DSL	Domain Specific Language
AST	Abstract Syntax Tree
CSR	Compressed Sparse Row
OLTP	OnLine Transactional Processing
OLAP	OnLine Analytical Processing
GAS	Gather–Apply-Scatter
BFS	Breadth-First Search
ER	Entity-Relationship
DAG	Directed Acyclic Graph
XML	eXtensible Markup Language
RDF	Resource Description Framework
HTML	HyperText Markup Language
QO	Query Optimizer

Chapter 1: Introduction

Real-world datasets often contain distinct entities (nodes) that are connected to each other via evident relationships (edges) that together form *networks*, also called *graphs*. For example, two users in a social network may be connected to each other either directly (if they are friends), or indirectly (if they have a friend in common), an example of which is shown below:

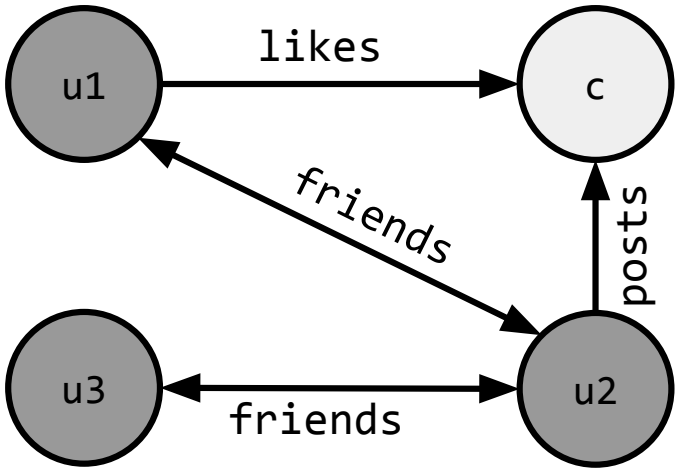


Figure 1.1: A graph where the dark nodes represent users, and the light nodes represent content. Three types of **edges** are depicted in this example (likes, friends, and posts). User `u3` is connected to user `u1` *via* user `u2`.

Graph analytics encompasses any algorithm that aims to compute certain information about the graph or its components through the traversal of its nodes and edges. Graph algorithms (e.g., shortest paths, centrality analysis, influence propa-

gation, community detection, network evolution, etc.) have been shown to provide substantial value in many application domains including finance, social media, education, sciences, and others. In fact, graph-powered applications are reportedly being used today by more than 75% of the Fortune 500 companies, ranging from banks and top retailers, to the majority of top automakers and aircraft manufacturers [1].

Even though the database community has often argued that traditional relational databases are up to the task [2–4], there has been a lot of work on specialized graph analytics frameworks and graph databases to facilitate data management and analytics for graphs. Nevertheless, enterprises continue to organize their data in partially or fully structured databases under some sort of schema, queryable using some flavor of SQL. According to `db-engines.com` [5], the *top 4* most popular databases used for data management today are relational databases, and *7* out of the 10 most popular systems support an SQL-like declarative language, SQL being the most ubiquitous way of interfacing with the relational data model. Such structured or partially structured databases can nevertheless contain a wide variety of graphs. These graphs exist either explicitly (by having the relation(s) that constitute the edges in the graph materialized in the database), or implicitly (if joins are required in order to first compute the edges).

In order to analyze these graphs, organizations often have to move their data to specialized systems. This can be extremely time-consuming and cumbersome since it requires a potentially complex Extract-Transform-Load (ETL) process that has to be done *manually*. In many situations, the ETL required consists of expensive joins

with *large outputs* (i.e., joins that exhibit low selectivity because of the cardinalities of the join condition attributes involved); often these outputs may not fit in memory even if the initial datasets do. This ETL overhead is one of the many *barriers* that stand between the RDBMS users and their ability to get from a data analysis idea to actual results using graph analytics algorithms.

Thesis statement: Graph algorithms are being used to derive value from graphs in many different application domains. Most data however is stored in the form of structured datasets, i.e., under a schema of varying strictness. These datasets typically contain many relationships between data tuples that can be thought of as edges in a graph. These relationships can however only be traversed via the computation of expensive joins. Database *indexes* can be used to make these joins efficient, however graphs hidden within structured datasets can be very dense, and often do not fit in memory.

By building an independent graph analysis layer on top of an RDBMS, accompanied by a high-level language for describing *graphs of interest* within RDBMSs, we can (a) eliminate the need for setting up manual ETL processes when *extracting* such graphs, and (b) reduce memory and time requirements when *storing and analyzing* these graphs. Custom in-memory representations can be used to connect RDBMS tuples that relate to each other. Such representations can mitigate problems of traversing *duplicate edges* from one node to its neighbors in a graph of interest (this duplication is inherent within structured datasets). Lastly, similar graph representations of RDBMS data can provide memory-efficient ways of computing relational *aggregation queries* over large-output [1] joins.

[1] We use this term throughout the dissertation from this point instead of "low selectivity" to avoid confusion.

We begin by putting the graph data model into context with current and past data models starting with the relational model in particular. Then we briefly outline the work that has been done in building specialized graph systems. We next discuss the barriers that still exist in conducting graph analysis over RDBMSs, as well as how our work fits into the broader graph analytics landscape.

1.1 Graph vs Relational Analytics

Graph analytics have established their place in many analytics workloads due to their ability to provide insights about a set of entities inside a network that may be interconnected in very complex ways. While graph theory problems and graph algorithms to solve them have been around for hundreds of years, the relatively recent surge of "big data" has provided many new real-world use cases for these algorithms and has also inspired new ones. One such example is the PageRank algorithm [6], that the founders of Google devised to rank webpages in search results. Graph algorithms are indispensable for many problems that span from optimizing routing processes (in computer networks, road networks, etc.), to understanding the physiology of cells (in biological networks) [7].

Graph Data Model: This data model can be used to store and operate on data in the form of a *graph*. In this dissertation we will deal with two different types of (directed) graphs: *simple graphs*, and *multigraphs*.

We define a *simple directed graph*[2] as a pair $G_s = (V, E)$ where V denotes a

[2]Referred to as a "simple graph" in the rest of the dissertation.

set of vertices (also referred to as *nodes*), and $E \subseteq \{(x, y)|(x, y) \in V^2\}$ denotes a set of *ordered pairs* that represent directed edges between those nodes. Note that *loops* are allowed (i.e., the edge (x, x) is allowed to exist). In particular, *loops* are edges in which the source and destination node is the same. They can represent a relationship that may exist between an entity n, and itself.

Each graph element (every node $v \in V$ and edge $e \in E$) is *unique*, and can contain a set of key-value pairs that represent a set of *properties*. A graph whose elements contain properties is also referred to as a *property graph*.

Example 1.1.1. In a social network context, we may have a simple graph where nodes represent users of the network. Two such nodes v_1, v_2 are users connected by a (directed) edge $v_1 \rightarrow v_2$, if v_1 sent a message to v_2. Node properties here can include the `profile_id` and `name` of each user, and edge properties can include the `time` at which the message was sent, and the `content` of the message.

Another type of graph we will discuss in this dissertation is a (directed) *multi-graph*, defined similarly as a pair $G_m = (V, E)$ where V denotes a set of *nodes*, and $E \subseteq \{(x, y)|(x, y) \in V^2\}$ this time denotes a **multi-set** of *ordered pairs* that represent directed edges between those nodes. In other words, unlike simple graphs, a vertex can have more than one outgoing edge to the same neighboring vertex.

We discuss how users can interact with simple graphs and multigraphs from a systems perspective in Section 2.3.1.

Relational Data Model: The most popular way users organize their data today is by using some iteration of the relational model [8], which views data as a collection

of logical tables (relations), that users can query using some flavor of SQL. The main driver behind the design of this model was to provide better data independence so that layout or design changes in the underlying storage would not cause issues in the application layer and vice versa. The relational model only describes the logical layout of data and is completely independent of the physical layout, which aided the easy implementation of different physical storage models that all implemented the relational model (e.g., row stores, column stores, key-value stores, etc.). This flexibility of the model is part of the reason for its popularity since it has the power to represent almost anything–including graphs.

A relation is associated with a set of attributes that describe the data being stored within. For example, to capture information about the *messages* being sent between users in a social network, one might create a `Users(uid, name)` relation, a `MessageInfo(mid, content)` relation and a `Messaged(sender, receiver, mid)` relation. The flexibility of the model allows for various valid ways to model the data; e.g., instead of the single `Messaged` relation, we could have two relations: `SentMessage(uid, mid)` and `ReceivedMessage(uid, mid)` (see Figure 1.2). Therefore, relations often contain attributes that reference other attributes in different relations–these associations are how data is connected to each other in relational databases. In this regard, the relational model is evidently more flexible than the graph data model. Since a relation R is associated with another relation S, each tuple in R can be associated to one or more tuples in S. Using these associations, relations are combined by computing *joins*, with new relations being output as results of these joins.

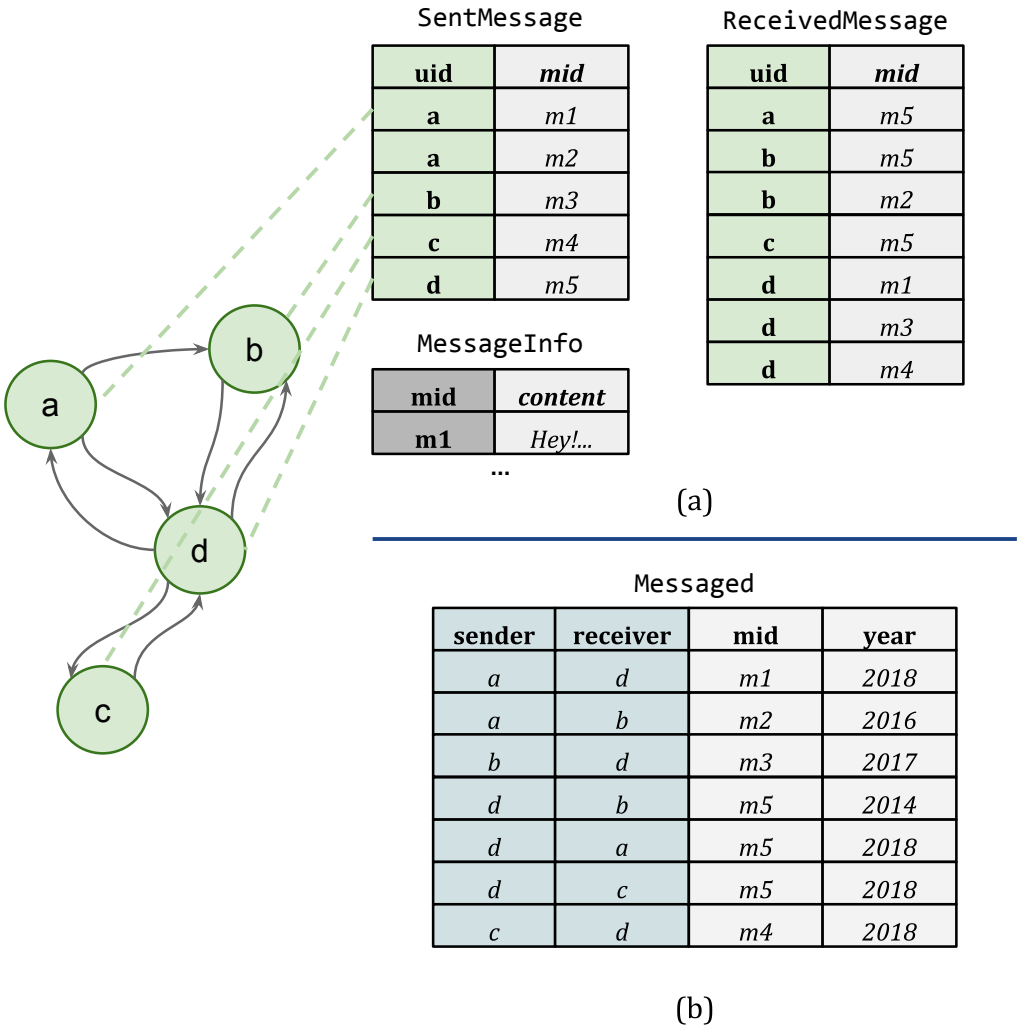


Figure 1.2: An example of a dataset stored under the relational model using two different approaches, and how each one can translate to a graph. In approach (a), in order to obtain the graph on the left, a JOIN between `SentMessage` and `ReceivedMessage` must be computed. The relation `Messaged`, shown in approach (b), is equivalent to the result of the aforementioned JOIN (after we choose a certain set of attributes to project as the result). The reader may assume that the attribute "year" is contained in `MessageInfo`.

1.1.1 Complementary Nature of Relational and Graph Analytics

Analysis of relational data is usually done by using SQL to filter, join and aggregate these relations to form standard "business intelligence" (BI) reports.

Example 1.1.2. Given the previous scenario of the social network, an example of a BI analytics query in this context is: *"What are the total messages received per user per week for the year 2018?"*. Say we had the relation `Messaged(sender, receiver, mid, year)` where `sender` is the id for the user that sent the message, and `receiver` is that of the user that received it (see Figure 1.2b). This query written in SQL would scan the `Messaged` relation, select only the messages from 2018, and then aggregate the tuples grouping them first by receiver, and then by week. Not only are such queries natural to express in SQL, but RDBMSs are also typically designed to *excel* in executing such queries with as few passes over the data as possible.

Now, in the same application as discussed in Example 1.1.2, say we wanted to find the most "popular" or "influential" users that year. This would be a task for the *PageRank* algorithm, which traverses the graph's edges, and computes a score that is based on the score of the neighboring nodes (in a recursive fashion). PageRank takes a holistic view of the network e.g., it considers connections from more popular users as being more important. This type of analysis requires multiple passes over the data and also takes into account the *context* of where the nodes exist in the network compared to its neighbors.

We observe therefore that while both graph and relational analytics can have a very important role to play in analyzing data, their roles are very different. While an RDBMS *could* theoretically be used for the aforementioned graph task, there are various issues that arise:

- Depending on how intensive the algorithm is, this may require ETL in order to materialize the Nodes and Edges relations. Based on how the schema is normalized `Messaged` may not be stored explicitly in a single relation–we could instead store relations `SentMessage` and `ReceivedMessage` as seen in Figure 1.2a.

- SQL is not as intuitive for expressing these iterative graph algorithms. The computation must be thought of as the combination of Edge relations rather than the traversal of a graph in order to be expressed in SQL.

Moving forward, it's important to also look at other models that showcase properties similar to the graph data model in order to provide the appropriate context for our work.

1.1.2 Relevant Historical Data Models

In the late 1970's and 1980's, researchers proposed a large variety of different data models [9]. Several of those models are somewhat reminiscent of the graph data model as their physical representations directly *connect* data tuples to other tuples they are associated with. This is in contrast to the relational model where tuples are grouped into largely *independent* tables, where relationships between data elements that do not coexist in the same relation, are explored by combining relations through *join* operations. Some examples include the "Network Model" (also known as CODASYL), as well as the Object-Oriented model. For the most part these did not see much commercial use except for very particular applications.

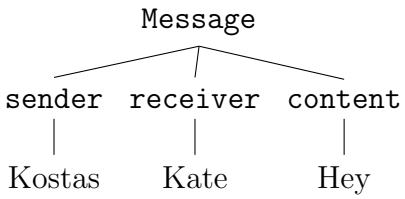
The reason for this was likely their high complexity, combined with the fact that the relational model seemed to be the most viable choice in practice–a trend which continues to this day.

1.1.3 XML

Developed in the late 1990's, eXtensible Markup Language (XML) is a data model that was very widely used and studied at the time. XML is an iteration of the *hierarchical* data model where the schema for the data forms a hierarchy structured as a *tree*, with the data values stored at the leaves. It is considered a "self-describing" data model, where the schema for the data is essentially part of the data itself. It is also "semi-structured" as the schema does not need to be defined apriori, which allows for objects of the same type to contain different sets of attributes. An example of a simple XML document is:

```
<Message>
    <sender>Kostas</sender>
    <receiver>Kate</receiver>
    <content>Hey</content>
</Message>
```

This describes a tree where the root node is `Message`, it has three children `sender, receiver, content` and those in turn have value leaf nodes `Kostas, Kate, Hey` respectively which can be seen below:



XML supports high-level query languages such as *XQuery* and *XPath*, serving the same purpose as SQL for the relational model. These resemble graph query languages in some ways as they traverse this tree structure to find and return a queried subset of data. Their declarative nature however makes it difficult to use for the purposes of expressing iterative graph traversal algorithms, and there are not many other interfaces for accessing XML data.

This tree structure that XML captures is a type of graph, therefore work in the area of efficiently storing and querying XML is at least partially relevant for attempting the same for general graphs. Even though it models interconnected data, the limitation of hierarchical models is that they cannot model an arbitrary graph, only a tree. XML also suffers from a lot of data repetition, as the schema labels must be repeatedly specified essentially for every distinct object in the database.

1.1.4 RDF

Resource Description Framework (RDF) is another relevant data model that came into the scene to accommodate the vision of the Semantic Web, enabling data to be shared and re-used across the web. In this vision, data would only be stored somewhere on the web *once*, and referenced everywhere else. The RDF format is essentially a network format i.e., it can represent arbitrary graphs. An RDF dataset

consists of a set of "triples", which can be thought of as rows in a table, that consist of three attributes: Subject, Predicate, Object. Each such triple can be thought of as an edge in the graph; e.g., in an RDF graph of users connected if one sent a message to the other, an RDF triple would look like this: {`Kostas, messaged, Kate`}. A high-level language called SPARQL is used to query RDF data. While RDF is logically a graph data model, it is not great at storing property graphs since each triple can only contain a single property (Predicate) about the Subject and/or the Object entities. In order to associate an entity with multiple properties one would need to create a separate triple for each property. Apart from SPARQL there are not many interfaces to RDF graphs, and a lot of iterative graph algorithms are difficult to express in a high-level language like SPARQL.

1.2 Specialized Graph Systems

Given the above background on relevant data models, we next briefly discuss past work that has been done to handle graph analytics workloads (see Chapter 6 for a more detailed discussion). The work discussed here focuses both on building specialized systems to handle graph workloads, as well as on leveraging existing systems for those workloads.

1.2.1 Graph Frontend, Graph Backend

Systems in this category include XML and RDF databases, as well as native property graph databases such as Neo4j [10], AWS Neptune [11], and OrientDB [12]

to name a few. These systems are built from the ground up to revolve around the graph data model, and use specialized graph representations in their underlying storage. They support SQL-like high-level query languages such as SPARQL, Cypher or PGQL, and also provide graph APIs like Gremlin or even direct access to the underlying graph, which is a necessity for expressing certain graph algorithms. Some systems (e.g., Neo4j) also offer a library of popular graph analytics algorithms to be used as black boxes. Most also provide support for ACID transactions. Migrating to this type of a system requires a complete buy-in into the graph data model which, as discussed in Section 1.1, is usually not ideal since relational analytics still play a big role in most enterprises. Moreover, these systems are not as mature or scalable as most RDBMSs, which have been studied for many more decades.

1.2.2 Bolt-on Solutions: Graph Frontend, RDBMS Backend

A common design for graph processing systems is to use a thin layer on top of an RDBMS that "shreds" graph data into a set of relational tables. It also converts graph queries from a graph query language or direct graph API into SQL queries to be executed against the RDBMS. These systems load graph data inside relational tables using a variety of different strategies. The early work on this was done in the context of using RDBMSs for XML data management [13], and there has also been work on building RDF databases that function in this fashion [14–18]. More recently there has been work on supporting graph APIs over RDBMSs using this design. Systems like SQLGraph [19] support graph queries on top of graphs stored

in RDBMSs, while systems like Vertexica [20] and Grail [2] use the same design but focus on batch graph analytics. The Titan [21] distributed graph database provides a graph interface over a variety of different multiple distributed back-ends[3]. A major challenge for these systems is designing good schemas and appropriate indexes for storing the data in the underlying RDBMS, since that will dictate the performance to a large degree.

1.2.3 Graph Analytics Frameworks

There is a variety of systems developed in recent years with two main goals in mind: *simplifying* the process of writing graph analysis programs, and *executing* these programs efficiently on very large graphs. These *graph analytics frameworks* are not concerned with transactional graph queries and expect a very particular graph format as their input. Most of the computation models for these systems are inspired by the Bulk Synchronous Parallel model [22]. Google's Pregel [23] is one of the systems that paved the way for multiple such "big graph" frameworks [24], later implemented in a variety of open-source and proprietary systems, one example of which is Apache Giraph [25]. Other systems in this space include GraphLab [26] and PowerGraph [27], that use a similar *Gather-Apply-Scatter* model with small variations in comparison to Pregel.

In order to use these frameworks, users need to manually conduct the appropriate ETL in order to extract their graph of interest from an existing database,

[3]The databases supported by Titan are mostly classified by "key-value stores" instead of relational databases but support SQL-like declarative languages.

transform it into the appropriate input format, and write their graph algorithm which will then be executed by the framework. These computation models are also very particular and they do not provide direct access to the graph for arbitrary traversal– all traversals need to be tailored to fit the computation model.

1.2.4 From Relational to Graph Backend

Lastly, there has also been work on transitioning an entire database from a relational model to a graph data model [28], which is another possible solution. Table2Graph [29] works by exploring the relational schema, in order to translate it into a graph schema i.e., make suggestions for the nodes and edges *types* and *attributes*. Systems like GraphBuilder [30] require a mapping from the relational data to the graph elements and attempt to efficiently extract and store the full graph. As discussed in Section 1.1 relational BI analytics are still a big part of analyses required and users have generally not bought into the graph data model entirely.

1.3 The Gap Between RDBMSs and Graph Analytics

As discussed above, each solution in the current landscape of options for dealing with graph workloads comes with a variety of different *challenges* that form a chasm between data stored in RDBMSs, and the ability to conduct graph analytics on this data. In this section we explain these challenges in detail.

1.3.1 Hidden Graphs in Relational Schemas

We first introduce the notion of "hidden" graphs inside normalized schemas. These are graphs that are not explicitly materialized in the database (i.e., the list of Edges for the graph is not explicitly stored). However, by *joining* various tables in the database, we can connect certain objects to form interesting graphs.

Here we discuss a few examples of such hidden graphs that exist inside relational schemas, and how extracting and analyzing them is important, and challenging:

Example 1.3.1. On the `DBLP` dataset [31], which stores journals, conferences, authors and publications, there are approximately 1.6 million authors, 3 million publications, and 8.6 million author-publication relationships. There is a variety of potential graphs of interest here:

- A *co-authors* graph, where there is a node for every author and two authors are connected by an undirected edge if they have published a paper together – analysis of such a graph can help understand which sets of authors are true collaborators and *suggest* potential collaborations between authors.
- A *co-attendance* graph, where an edge between two authors indicates that they attended a conference together – analysis of such a graph may help understand dissemination of ideas across a research community.
- A *co-published* graph where there is a node for every publication, and two publications are connected if they were presented at the same conference.

Such a graph might help with creating a prediction model that classifies a future publication to an appropriate venue using machine learning techniques.

The **co-authors** graph which, in this dataset, contained 86 million edges required more than *30 minutes* to extract and load on a laptop since this required an expensive *non-key*[4] join– a *self-join* on the `AuthorPublication` table (that stores the association between authors and their publications). As we can see here, the graph is an order of magnitude larger than the underlying `AuthorPublication` table.

Example 1.3.2. The `TPC-H` dataset [32], is an artificially generated supply chain dataset that maintains customers, orders, items, suppliers, etc. An interesting graph to analyze here would be a **graph of customers that have bought a common item**. This dataset (for scale factor `SF=1`) contains about 150,000 customers, that made 1.5 *million* orders containing 200,000 distinct items, and 6 million different order-item pairs. In contrast with `DBLP`, the `TPCH` schema is more *normalized*. As can be seen in Figure 1.3, extracting this graph first requires a join over the `LineItem` and `Orders` table to figure out which customer bought which item. Afterwards, a self-join over the result gives us the final set of edges in the graph. Due to the fact that there is a small number of parts but a large number of customers and orders, this graph is especially dense, at over 186 million edges. From a quick look at the schema in Figure 1.3, one can point out many more graphs "hidden" within this dataset like (i) a graph of suppliers connected if they sell a common item, (ii) a

[4] A join where the relationship between the joining attributes is *not* a key-foreign key relationship

graph of parts sold by a common supplier, etc.

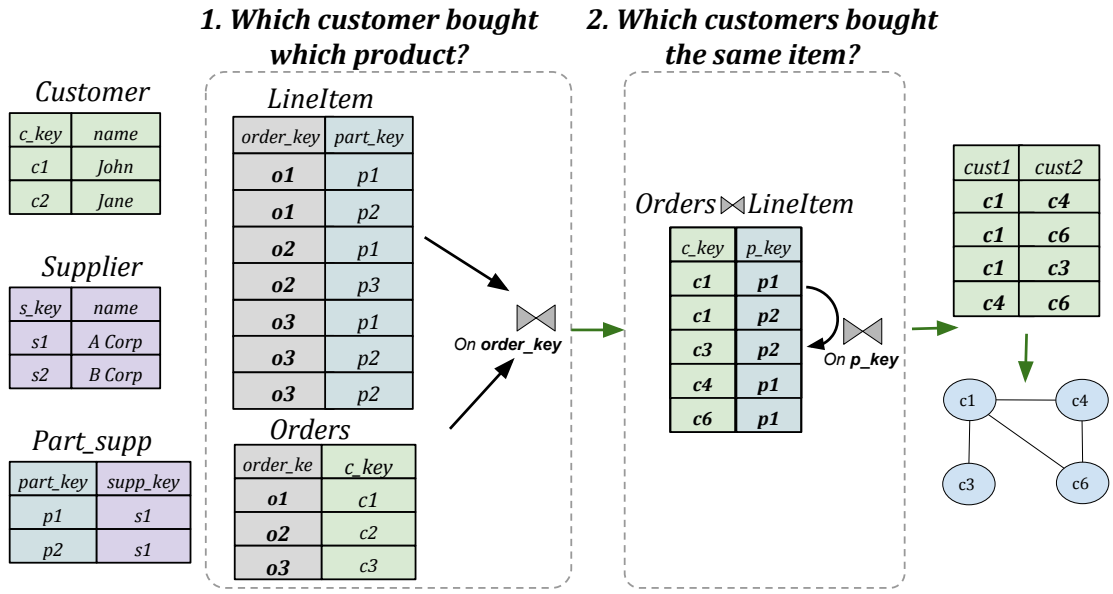


Figure 1.3: Extraction of a "hidden" graph of customers, connected if they've bought a common item from the TPCH dataset.

The above examples are not atypical – data is usually at least loosely structured and stored under some sort of schema where graphs exist implicitly and require expensive joins to extract. The first challenge we attempt to tackle in this dissertation is: *how do we enable users to extract and analyze such graphs efficiently, and intuitively?* We view the goals of providing *intuitive interfaces* and *efficient execution* as interconnected. There is a variety of sub-challenges and concerns in tackling these goals and being able to analyze graphs in situations like the above examples:

1. **ETL (Extract, Transform, Load):** The user would be required to manually formulate the appropriate SQL queries in order to extract the nodes and edges from the underlying database. Moreover they would need to parse and transform the tuples returned by the database into the appropriate in-memory

data structures or serialization formats, to load the graph into a specialized graph processing system. In the case of a specialized system they would need to have that system set-up in an appropriate environment (single-machine or cluster), and be sufficiently familiar with the system's interfaces for loading the data.

2. **Expressing Graph Analytics:** The user needs to somehow access/operate over the graph in order to conduct their analysis. Depending on the system they have access to, they would be required to have sufficient understanding of the underlying *execution framework*. Usually users or organizations only have a small set of specialized systems for graph analysis (if any) set up in their workflows, which often might not cover the spectrum of different types of graph analytics that could arise (as discussed in more detail in Chapter 6). Users may then need to figure out unorthodox ways of using the frameworks at their disposal to complete their analysis. This is a cumbersome and highly inefficient process that slows down end-to-end analysis significantly.

3. **Large-Output Joins:** Typically, there is a one-to-one relationship between graph nodes and tables in a database, e.g., for an "author" node, there will exist some sort of an *author table*. In many cases, normalized schemas maintain various different attributes about each entity in separate relations to avoid data duplication. Such attributes can be fetched with "key-foreign key" joins which are easy for the database and do not "blow up" in terms of the result size. Joins that connect entities together however could be non-key joins. A

non-key join between multiple relations can (in the worst case) yield a result *exponentially* larger than the input relations. We refer to such joins that "blow up" as *large-output* joins. 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**.

1.3.2 Large Output Joins in Relational Query Processing

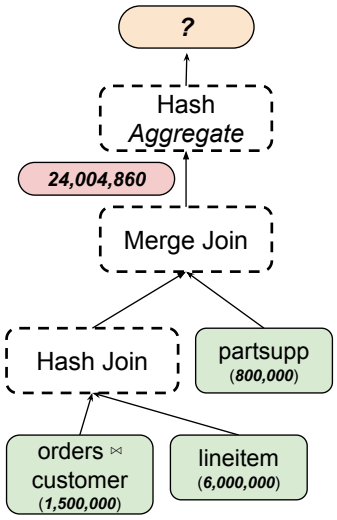
Large-output joins are at the forefront of *challenges* that we face when trying to efficiently interface users with graphs within databases. They constitute a barrier very difficult to break, since such joins entirely *block* the analysis until they are completed. Users often want to compute *aggregations* over such large-output joins, where the results can be orders of magnitude smaller than the size of the join. These aggregations can happen in the context of a graph (in the form of aggregating edges), or can simply be viewed from the lens of generalized select-project-join query processing. Data *pipelining* can be used in some situations during query execution, which will avoid materializing intermediate join results. Pipelining works by conducting the entire join one tuple at-a-time. The main limitations to pipelining are twofold: (a) the entire join result will still need to be enumerated, and (b) if hash-aggregation is used, the memory required to store the hash-table is difficult to predict. If sort-aggregation is used, the final join result prior to aggregation will need to be *materialized*.

The second challenge we therefore contend with in this dissertation is: **how do we efficiently compute these aggregations without having to store the full intermediate join result?**

Example 1.3.3. Consider a query like `[Q1]` in Listing 1.1 over the standard `TPC-H` dataset. The `LineItem` table includes all orders of parts that were supplied, the individual parts each order contains, as well as *which supplier* each part was purchased from. The goal of `[Q1]` is to compute the *count* of `(order,customer,item)` records we are storing for each supplier for every *zip code* in which that supplier satisfied orders, *given* the transaction data that we already have. Note that `c_zipcode` isn't a distinct field in the `customer` table, but is typically extracted from the `c_address` attribute. This type of complex *decision-support* query requires a *non-key* join that could yield very large intermediate results that will be fed as input to the aggregation operator. As shown in Figure 1.4, running `[Q1]` over `TPC-H` (using scale factor `SF=1`), the intermediate join result for this query contains over 24 million tuples. The size of the result post-aggregation would be bounded by the number of distinct zip codes times the number of suppliers, and therefore is highly likely to be orders of magnitude smaller than the join result.

```
SELECT ps_suppkey, c_zipcode, COUNT(*)
FROM partsupp, lineitem, orders, customer
WHERE ps_partkey = l_partkey AND
      o_orderkey = l_orderkey AND
      o_custkey = c_custkey
GROUP BY ps_suppkey, c_zipcode;
```

Listing 1.1: `[Q1]` Query for finding the total count of `(order,customer,item)` records we have for each supplier *per zip code* in which that supplier satisfied orders (`TPC-H` dataset)



(a) Query plan for query [Q1]

Figure 1.4: Query plan for query [Q1]. Aggregate queries can have very large intermediate results even though the number of output groups could be small

Example 1.3.4. Another example of queries that require large-output joins include *path aggregation queries* in graphs. Any graph stored inside a relational database in the form of a *Nodes* and *Edges* table, is conducive to queries that count the number of paths that follow a certain pattern in terms of the nodes. If we had an edge table `Edges(src,dst)`, and a `Nodes(id,label)`, a query like [Q2] shown in Listing 1.2, counts the paths between nodes with certain labels. Such queries end up outputting a huge number of intermediate results corresponding to the sub-paths for each intermediate stage of the graph traversal.

```
SELECT n1.label, n2.label, COUNT(*)
FROM Nodes n1, Edges e1,
     Edges e2, Nodes n2
WHERE n1.id = e1.src AND
      e1.dst = e2.src AND
      n2.id = e2.dst
GROUP BY n1.label, n2.label;
```

Listing 1.2: [Q2] Generic graph pattern counting query

1.4 Contributions

In this dissertation, we tackle the challenges associated with closing the gap between the current way most data is stored, and the ability to conduct graph analytics on them efficiently. Our overall contributions are three-fold:

GraphGen **System:** We built a system called GraphGen, that acts as a "bolt-on" layer on top of RDBMSs, and provides a full-featured set of graph analysis interfaces, enabling the user to express their analysis on a per-case basis, using a single lightweight system. From this viewpoint, our contributions are the following:

- **A Graph Definition Language:** We propose a high-level declarative Domain Specific Language (DSL) for graph definition called GraphGenDL based on Datalog for specifying graph extraction queries. This language is an intuitive, declarative way of combining underlying RDBMS tables to populate graph elements (nodes and edges) and their properties. This level of language abstraction allowed us to also include extensions for describing *collections of graphs*. We show how such language constructs can also unlock the ability to conduct *what-if* analysis over graphs.

- **Extraction of Graph Collections:** We develop a novel technique called *tagging*, which employs a set of query rewrite rules for efficiently extracting *a collection of graphs* from RDBMSs. Our preliminary experiments show that tagging enables orders of magnitude speedups in extracting collections of graphs when compared to executing multiple queries over the database.

Condensed Graph Representations: We introduce the idea of a *condensed* graph, a physical graph representation that we leverage this novel representation in order to efficiently analyze hidden graphs in structured datasets. It represents a set of *source* nodes *indirectly* connected to another set of *destination* nodes. We call the source and destination nodes *real* nodes, and they are connected *via* a series of intermediate *virtual* nodes. The only difference between real and virtual nodes is that the latter are *not accessible* by users and are leveraged to reduce the memory required to maintain all the connections (edges) between the real nodes. Therefore, real nodes can be connected to the same neighbor through one or more virtual nodes (or combination of such virtual nodes)–we refer to this property of condensed graphs as *duplication*. It's important to note that a condensed graph describes a *physical* data representation, and can be *interfaced* as either a simple graph or a multigraph.

Condensed graphs exist inherently within RDBMSs. We describe a general framework for extracting a *condensed* representation for *acyclic*[5], aggregation-free extraction queries over arbitrary RDBMS schemas. Condensed graph representations enable the analysis of very dense graphs, using orders of magnitude less memory. They also allow for more time-efficient analysis of such graphs for a certain class of graph algorithms. We propose a suite of novel pre-processing and *de-duplication* techniques we have developed over condensed graphs allow for the execution of arbitrary graph algorithms over condensed graphs. Each class of these de-duplication

[5]A conjunctive (join) query is intuitively defined as *acyclic* if the *hypergraph* associated with the query has no cycles. More formally, a query is acyclic if it is reduced down to an empty hyperedge after the GYO reduction algorithm [33, 34] has been applied to it.

techniques yields a distinct physical graph *representation*. We systematically analyze the benefits and trade-offs offered by extracting and operating on these representations, we provide extensive microbenchmarks of each representation, and discuss how GraphGen decides which representation to use each time.

Memory-Efficient Aggregation Processing using Graph Representations: We propose a new *multi-way* database operator called Join-Agg, which enables the efficient computation of aggregation queries, *without materializing* any intermediate join results, by computing the *join* and *aggregation* simultaneously. We describe a novel *general framework* for executing aggregation over conjunctive queries involving arbitrary numbers of relations, and an arbitrary set of group-by attributes that may be derived from any participating relation, by leveraging a *graph* representation of the underlying *data* (restricted to acyclic queries). We implement a prototype of the Join-Agg operator *outside* of the RDBMS and experimentally showcase the benefits of our operator over synthetic and real datasets. We also provide a comprehensive complexity analysis of common categories of queries that benefit from our Join-Agg operator. We compare our technique against the classical RDBMS model, or other less general techniques such as pre-aggregation [35] which only looks at reducing intermediate data size at the level of each individual join instead of looking at the join as a whole. We show that in terms of computational complexity, Join-Agg is comparable or asymptotically better than those techniques, particularly in the general case of complex acyclic branching join queries. We also show that Join-Agg is overall *better* than those techniques in terms of memory

complexity.

1.5 Outline and Previously Published Work

The remainder of this dissertation is organized as follows. We begin by describing the inner workings and system implementation details of GraphGen, as well as our graph specification language GraphGenDL in Chapter 2. Chapter 3 discusses our work on enabling graph analysis over hidden graphs inside RDBMSs using GraphGen, including *condensed* graph representations and techniques for conducting *de-duplication* over them. Chapter 4 continues the discussion on GraphGen by describing various extensions to our GraphGenDL language towards supporting the extraction of complexly defined *collections of graphs*. Chapter 5 presents our proposed Join-Agg multi-way database operator, that leverages graph representations for the efficient processing of aggregation queries over large-output joins. Chapter 6 discusses related work, and Chapter 7 concludes with a discussion on what we have learned from this investigation.

Chapter 1, Chapter 2, Chapter 3, and Chapter 6 contain material from our published work [36–38]. Chapter 5 contains material from one of our arxiv preprints [39].

Chapter 2: GraphGen System Overview

In this chapter we delve into the inner workings of GraphGen. We introduce the implementation details of the underlying system and all of its moving parts, and discuss the design decisions we made. We also enumerate the different ways users can interface with graphs inside their RDBMSs using GraphGen.

As discussed in the previous chapter, the major focus of the GraphGen system is to: (a) enable analysis on *very large* graphs that are hidden in structured datasets and would typically not fit in memory, as well as (b) enable analysis on *graph collections* that would normally take a substantial amount of manual ETL.

We begin with a brief description of the key components of GraphGen, and how data flows through them. We then describe our Datalog-based DSL (Domain-specific Language) that we have designed for specifying extraction jobs (called GraphGenDL), and APIs provided to the users after a graph has been loaded into memory.

2.1 Architecture

The inner workings of GraphGen and the components that orchestrate its functionality are demonstrated in Figure 2.1.

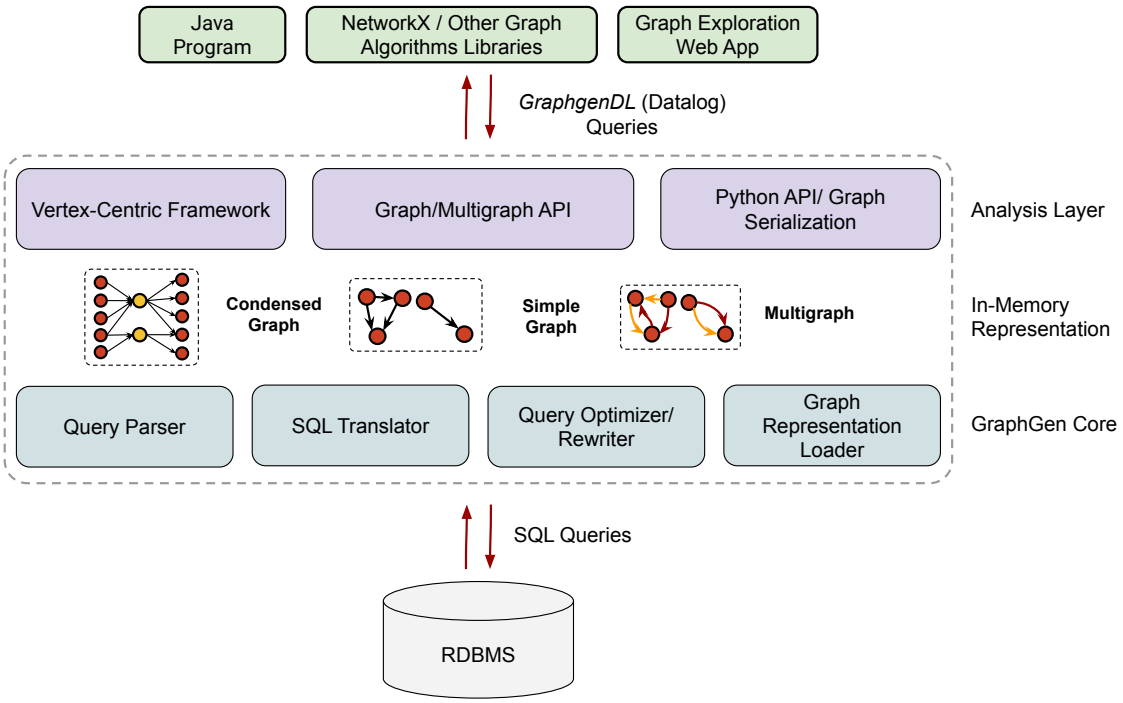


Figure 2.1: The high-level architecture of GraphGen, a bolt-on analysis layer on top of RDBMSs that enables efficient extraction and analysis of "hidden" graphs that exist within RDBMS schemas.

The system is composed of two main layers: The *GraphGen Core*, and the *Analysis* layer. The core layer is where most computation happens– this includes parsing user-specified *graph extraction tasks*, translating those into SQL, and handling the responses from the RDBMS. The extracted graphs are maintained in one of a variety of *in-memory representations* depending on the nature of the extracted graph and possibly the nature of the analysis being performed. It is important to note that in-memory representation in Figure 2.1 shows the *physical* representation of the graph data, which is de-coupled from the way the user *interfaces* with the graph. The analysis layer contains this set of interfaces that communicate with the in-memory representation and access the physical graph in different ways.

At a higher level, GraphGen 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 called GraphGenDL, where the user specifies *how* to construct the nodes and the edges of the graph (in essence, as *views* over the underlying tables).

We have built a custom parser for GraphGenDL described above using the ANTLR [40] parser generator. The parser creates the Abstract Syntax Tree (AST) of the query which is in turn used for translation into SQL. The translation itself requires a full walk of the AST, during which the system gathers information about the statement, loads the appropriate statistics, and attribute information for each involved relation from the database, and creates an intermediate representation of the query as a `GraphQuery` object, which then passes through the custom Query Optimizer (QO). The QO then makes the appropriate decision as follows:

- If the graph extraction task specifies a *single* graph, then the system analyzes the *selectivities* of the joins required to construct the graph by using the statistics in the system catalog. This analysis is used to estimate the result *sizes* of the joins required to extract the requested graph output, and 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 (see Section 3.2.2). Note that a single graph could either be a simple graph or a multigraph.

- On the other hand, if the graph extraction task specifies a *collection* of graphs (a *set* of distinct graphs), the QO applies our query rewrite rules to the input

`GraphQuery` object, and sets the appropriate *tags* to the elements of the base graph (see Chapter 4 for more details on tagging). Again, note that a collection of graphs could again consist of simple graphs or multigraphs (or a combination of both).

The QO uses the Query Translator (which automatically translates `GraphQuery` objects into SQL) to generate the appropriate set of the final SQL queries to pull the graph data from the RDBMS. The mechanism by which we load the graph into memory assumes that the total size of the graph(s) described by a single extraction task is *smaller* than the total amount of memory available so that graphs can be analyzed without requiring disk I/O. As graph algorithms typically require random access to the entire graph, GraphGen (like most other high performance graph analysis libraries) assumes that there is sufficient memory to load at least a condensed representation of the graph (as we will see, the condensed representation is often orders of magnitude smaller than the actual graph, enabling GraphGen to analyze very large hidden graphs).

The SQL queries are executed in sequence, and the output graph object is handed to the user program. After extraction, users have the following options:

- Operate *directly* upon any portion of the graph using the Java Graph API (discussed in Section 2.3.1)
- Define and execute *vertex-centric* programs on it (which can take advantage of multiple CPU threads) – the "vertex-centric" framework is a widely used framework for expressing and executing graph analysis algorithms [41].

- *Visually* explore the graph through our front-end web application.
- *Serialize* the graph onto disk (in its expanded or condensed form) in a standardized file format, so that it can be further analyzed using any specialized graph processing framework or graph library (e.g., NetworkX, GraphFrames, Neo4j, etc.)

2.2 GraphGenDL

Datalog has been increasingly used for expressing data analytics workflows, and especially graph analysis tasks [42–44]. 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. GraphGenDL is a declarative language that enables users to map nodes and edges in graphs of interest to combinations of RDBMS relations. It is based on a limited non-recursive subset of Datalog, augmented with range and aggregation constructs; in essence, it 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 is **not** to specify a graph *algorithm* itself using Datalog (like Socialite [43]); however we do plan to explore this avenue in future work, enabling users to specify graph queries or analysis tasks using Datalog together with the graph extraction query.

GraphGenDL naturally generates *directed* graphs, and undirected graphs are represented using *bidirectional* edges. The typical workflow for a user when writing an extraction query would be to initially inspect the database schema, figure out

which relations are relevant to the graph they are interested in exploring, and then choose which attributes in those relations would connect the defined entities in the desired way.

The syntax of GraphGenDL is quite simple– there are technically only two parts to every query. The first part is the graph view *type definition*, followed by the *element definitions* (definitions of the nodes/edges) for the specific graph view. GraphGenDL can be used to specify extraction tasks for a *single* graph (which could either be a simple graph or a multigraph), or a *collection* of graphs. At a high level, users specify the type of graph(s) of interest and then express a set of mappings from RDBMS views to Nodes/Edges.

2.2.1 Syntax: Single Graphs

Type Definition: A generic example of a single graph extraction task can be seen in Listing 2.1. For *simple* graph views, the syntax for the type definition is: `CREATE GRAPHVIEW` α, where "α" is the alias we want to use to refer to the graph view. `SQ1` in Listing 2.2 is one example of a simple graph view.

```
CREATE GRAPHVIEW α
  Nodes(ID, p) :- S(ID, p).
  Edges(ID1, ID2) :- R1(ID1, a1), R2(a1, a2), ..., Rn(an−1, ID2)
```

Listing 2.1: A generic GraphGenDL query expressing a *single* graph extraction task.

Any query that follows the template shown in Listing 2.1, could also be representing a *multigraph*. This can happen if there are multiple instances of the same edge in the resulting `Edges` view. By simply using `CREATE MULTIGRAPH VIEW` in

```
[SQ1]
CREATE GRAPHVIEW co-authors
Nodes(ID, Name):- Author(ID, Name).
Edges(ID1, ID2):- AuthorPub(ID1, PubID), AuthorPub(ID2,
   PubID).

[SQ2]
CREATE MULTIGRAPHVIEW cust_common_item
  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).

[SQ3]
CREATE GRAPHVIEW instructors_students
  Nodes(ID, Name) :- Instructor (ID, Name).
  Nodes(ID, Name) :- Student (ID, Name).
  Edges(ID1, ID2) :- TaughtCourse (ID1, courseId),
     TookCourse(ID2, courseId).
```

Listing 2.2: `SQ1` extracts a simple graph of authors that are connected if they have published at least one paper together. `SQ2` extracts a multigraph of customers connected to each other once for every item they have bought in common. `SQ3` extracts a graph with two types of nodes: instructors, and students. There is an edge between an instructor and a student if the student was in at least one of the instructor's classes.

the graph definition statement, the returned graph is treated as a multigraph and its edges are therefore *not deduplicated* (see Chapter 3 for more on deduplication). This means that any call to `getNeigbors()` may return the same neighbor more than once (see `SQ2` in Listing 2.2).

Element Definitions: This is the part of the query that defines what underlying relations need to be combined (and how) in order to form the nodes/edges elements of the simple graph or graph collection (as shown in Listing 2.1). Each Nodes/ Edges statement in a query specifies a set of nodes/edges that will be part of the graph. We use the datalog-like syntax of GraphGenDL in our examples for the

sake of simplicity, but the user can choose to express their queries in SQL as well, and leverage all the capabilities of modern SQL. This does not change the way the system works as the input datalog is translated to SQL regardless. SQL is the only way GraphGen communicates with the underlying RDBMS as seen in Figure 2.1.

The left hand side of each statement specifies the *schema* (id, and properties) of the node/edges specified by that statement. The user writes `Nodes(ID,` p`)` for nodes statements, and `Edges(ID1,ID2,` p`)` for edges statements, where p is a set of *properties*. Note that the identifier for each node element is `ID` and is *always going to appear first* in the schema. The same applies for `ID1, ID2` in the case of edges statements; `ID1` is the `ID` of the source node for this edge, while `ID2` is the destination.

Multiple Nodes/Edges Definitions: To make the process more intuitive, we allow for *multiple* Nodes and/or Edges statements within a query (e.g., `SQ3` in Listing 2.2). This serves two purposes: first, it enables users to separate the definition of one *type* of node/edge from another, especially in the case of heterogeneous graphs with multiple types of nodes or edges that typically reside in different relations. Second, if attributes for one element are stored in different relations, the user can pull each attribute from its individual relation with a separate statement.

Moreover, entities might contain a certain property in their schema but lack a *value* for another property in the database. A graph where elements have null values for some of their properties does not pose a problem as long as the interconnection structure is accurate. By providing two separate statements the user communicates

to the system that they require all nodes that have one property *and* all nodes that contain the other. If values for both nodes appear for the same element id, a single node that contains a value for both properties is loaded. Our system translates these sets of queries into *full outer joins*, which load a union of all specified node entities.

User Defined Views: Users may also define other views that can be used as auxiliary views within Nodes and Edges statements.

2.2.2 Syntax: Graph Collections

GraphGenDL can also express the mapping for complex graph collections that exist in RDBMSs. Under the hood, GraphGen extracts collections of graphs efficiently by viewing them as a single *multigraph*, and optimizing the extraction SQL queries accordingly, which in a sense groups each of the vertex and edge in their respective *distinct* graphs. An alternate implementation where each graph in the collection is extracted independently is also possible, but would not be able to exploit the overlaps typically seen in such graph collections (see Chapter 4). Four primary arguments are required to define a graph collection over a database: a) *nodes statements*, and b) *edges statements*, which define the *base* graph, and a parameterization, defined using a c) *tagging predicate*, and d) the *range* of values over which this extraction task should be parameterized.

Type Definition: Using `CREATE GRAPHVIEW COLLECTION` yields a graph collection as seen in Listing 2.3. An additional `WHERE X IN RANGE (FROM, TO, STEP)` statement is expected to follow after the name of the graph collection. This statement

returns a set of graphs (each of which could be a simple graph or multigraph).

Element Definitions: Users can specify graph collections in GraphGenDL by expressing the as a *parameterization* over a single *base* graph. The element definitions for this base graph are made in exactly the same way as for single graphs discussed in Section 2.2.1. The same exact details apply for graph collections when it comes to multiple node/edge definitions and user defined views.

```
CREATE GRAPHVIEW COLLECTION m
WHERE X IN RANGE (FROM, TO, STEP)
  Nodes(ID,p1,...,C) :- R(ID,p1,...,C), f(C,X).
  Edges(ID1,ID2,p1,...) :- S(ID1,ID2,p1,...).
```

Listing 2.3: Generic graph collection query.

Parameterization: The `WHERE` clause (seen in Listing 2.3) in association with the tagging predicate f specifies the way in which the graph to be extracted will be split into a collection of graphs. Every value for `X` in the `RANGE` specified by the query will correspond to a different graph in the collection. The variable `C` refers to the appropriate attribute in the element definitions that will associate each element with a set of distinct graphs based on the tagging predicate function $f(\texttt{C}, \texttt{X})$. Either the nodes or the edges definitions (or both) will contain one or more references to `C`, which is the property of the (vertex or edge) element whose value dictates which subgraph said element will be a part of. It's important to note that because a `C` variable must be properly defined, the "`ID`" attribute might not appear in the left hand side of a statement for graph collection queries (e.g., `Nodes(C,name)` in Listing 2.4)), we know that `C` references the node ID because it appears as the first property in the schema.

```
CREATE GRAPHVIEW COLLECTION ego-graphs
WHERE X IN RANGE (Author(C))
  Nodes(C,name):- Author(C,name), C = X.
  Nodes(ID,name):- Authorpub(C,p), Authorpub(ID,p),
     Author(ID,name).
  Edges(ID1,ID2):- AuthorPub(ID1,p),Authorpub(ID2,p).
```

Listing 2.4: Extracting a set of ego-graphs over a graph of co-authors. Note that SQL can be used for any Nodes/Edges statement instead of our Datalog syntax.

In `Q1` shown in Listing 2.5, `C` refers to the date of of birth of an author (`year(C)` extracts the year of birth). Our tagging predicate here is *f*`(C,X) = (year(C)=X OR year(C)=X-1 OR year(C)=X+1)`, we are running this over a range of values defined by the range expression `X IN RANGE (1900,2000,1)`.

Range Definition: The generic syntax for specifying a range of values for a parameter variable `X` is "`X IN RANGE (`*r*`)`", where *r* can take *three* different forms. Users can state the range of possible values for variable `X` in a `START, END, STEP` format, which would yield all values from `START`, to `END` inclusive, incrementing by `STEP` (see Listing 2.5). A *query* that returns a list of values can also be provided instead–e.g., querying a graph for every author id could be done by: `X IN RANGE (Author(ID))`. Any generic SQL query can be specified as long as it only projects *a single column* of distinct values. Lastly, users may also specify an ad-hoc list of values like `X IN RANGE ({a, b, c, d})`.

2.3 Internal Data Structures & Interfaces

The most efficient means to utilize GraphGen is to directly operate on the graph either using our native Java API, or through a *vertex-centric API* that we

```
CREATE GRAPHVIEW COLLECTION co-author-peers
WHERE X IN RANGE (1900, 2000, 1)
  Nodes(ID, name, C) :- Author(ID, name, C), year(C)=X
     OR year(C)=X-1 OR year(C)=X+1.
  Edges(ID1, ID2) :- AuthorPub(ID1,pub), AuthorPub(ID2,
     pub).
```

Listing 2.5: A graph collection extraction task that queries a co-author graph for every year X, which contains only authors that were born a within a year of X.

provide. Both of these have been implemented to operate on all the in-memory (condensed or otherwise) representations that we present in Chapter 3.

Simple Graphs and Multigraphs: The basic data structure that we use for storing the graphs is a variant of traditional Compressed Sparse Row (CSR) representation [45]. We store all node data in a `HashMap` *index* where each node id is associated with that node's data properties. Briefly, for each node, we maintain two *mutable* `ArrayLists`– one `InNeighbors` for its *in-coming* edges and one `OutNeighbors` for its *out-going* edges. We use Java `ArrayLists` instead of linked lists for space efficiency. The trade-off we make is that, it makes *vertex deletions* more expensive because those require rebuilding of the entire index of vertices. We 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 `InNeighbors` and `outNeighbors`, 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. This representation is used to represent simple graphs as well as multigraphs. We are not storing any properties for every distinct edge, but this representation can easily be tweaked in order to do so.

Graph collections: We use a variation of the CSR representation for storing graph collections as well. Graph collections can often portray large overlaps in their node and edge sets. Say we had a graph that is not prone to changes very often, e.g., a co-authors graph. If we were to extract snapshots of that graph every year, each snapshot would contain a lot of the same nodes and edges apart from any new connections that may have been formed by folks who were first time co-authors. For this reason, our physical storage representation of a graph collection uses a CSR representation, which now contains sets of *multi-vertices*, and *multi-edges* instead of simple vertices and edges. We also refer to multi-vertex and multi-edge elements as *multi-elements*.

A `MultiEdge` or `MultiVertex` is a single vertex or edge object that contains multiple different property sets e.g., if a single edge (v_1, v_2) exists *both* in graph g_0 and graph g_1, it can have different attribute values in each graph. Similarly, vertex v_1 can exist in both of the aforementioned graphs in the collection, but contain different attribute values in each one. Each `Multi`-element has a unique identifier, but contains a list of attribute sets called `properties_set` which stores the different sets of possible properties for the particular vertex/edge, as well as an array of BitSets called `exists`. Intuitively, a specific set of properties `properties_set[i]` contains the property values for the element, and `exists[i]` contains the BitSet that dictates in which graphs the particular element (a) exists, and (b) has properties `properties_set[i]`. Each BitSet in `exists` is the same size as the *number* of graphs in the collection. Note that the BitSets in `exists` don't overlap i.e., we only store a specific version of each multi-element once. In other words, if the same

vertex/edge *with the same attribute values* appears in both graph g_0 and graph g_1, we are only storing it once. Lastly, we store all multi-vertices in a single `vertices` list, and all multi-edges inside a single `edges` list which only contains the *out-going* edge mappings.

2.3.1 Java APIs

All of our in-memory representations implement the following API.

Simple Graphs and Multigraphs: The API supports 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.

Graph Collections: A graph collection is a *set* of graphs, each one distinguished by a unique identifier– we call that identifier a `versionId`. The set of `versionId`s

is stored in the system as a Java `BitSet`, where the index of each set bit represents the `versionId`. We also store a mapping from `versionId` to a tag that represents the identifier for each unique graph in the collection. Graph collections are stored in a special `GraphCollection` class.

There are three main classes important to understanding the structure of Graph Collection in GraphGen: `GraphCollection, MultiVertex, MultiEdge, SingleEdge`.

The main `GraphCollection` API supports the following operations:

- `getVertexList(versionId):` Returns a list of vertices that appear in the graph with identifier `versionId`.
- `getNeighbors(v):` This will return *all* neighboring vertices to `v`, from all graphs in the collection. It returns a list of `MultiEdge` objects.
- `getNeighbors(v, versionId):` Returns a list of `SingleEdge` objects, which are the distinct edges that exist *only* in the graph with identifier `versionId`.

Multi-Elements: The main `MultiVertex/MultiEdge` API allows the following operations:

- `getId():` Returns the unique identifier for a vertex, which may exist (with the same or different attribute values) across multiple `versionId`s in the graph collection.
- `getVersions():` Returns the set of versions this vertex/edge appears in. This version set is stored as a Java `BitSet`.
- `getAttributeSets():` Returns the list of `Attribute[]`, each array in the list

contains the set of attribute values for the particular vertex/edge across all `versionId`s in the graph collection.

2.3.2 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 on 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 [26] in which nodes communicate by directly accessing their neighbors' data, thus avoiding the overhead of explicitly storing messages in some intermediary data structure.

2.3.3 External Libraries

GraphGen can also be used through a library called *graphgenpy*[1], a Python wrapper over GraphGen allowing users to run queries in GraphGenDL through simple Python scripts and serialize the resulting graphs in a standard graph format, thus opening up analysis to any graph computation framework or library (Section 3.4.4). A similar workflow was used in the implementation of our front-end web application [36] through which users can visually explore the graphs that exist within their relational schema.

The library provides a static `generateGraph()` method, that takes as input a graph extraction query in our DSL, and either returns a single `Graph` object, or a `MultiGraph` object.

Simple `Graph`s also implement the widely used *Blueprints* API [46]. Blueprints is a generic graph Java API, that provides graph access methods like `getVertices()`, `getEdges()`, etc., and is used by several graph processing and programming frameworks (including Gremlin [47], a popular graph traversal framework). By supporting the Blueprints API, we immediately enable use of many of these already existing toolkits over extracted graphs (whether they are one of our condensed representations or not).

In our current implementation, a returned `Graph` object may be a `TinkerGraph`, or a subclass that supports a condensed representation (see Section 3.2.3). `TinkerGraph` is an in-memory implementation of the property graph model, and is part of the

[1] `http://konstantinosx.github.io/graphgen-project/`

open-source TinkerPop stack (`http://www.tinkerpop.com/`).

2.4 Web Based Graph Exploration Interface

GraphGen also features a graph discovery and exploration component that provides two main functionalities. First, it allows a user to specify a graph extraction query and to interactively explore the returned graphs. 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. The latter is work done primarily by Udayan Khurana and more details about that portion of the web application can be found in the related co-authored paper [36].

The front-end allows a user to: (a) connect to an existing relational database and view its schema, (b) write queries in GraphGenDL 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 2.2 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 back-end, 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. We currently support exporting graphs in `GSON` or `GraphSON` formats. Finally, if the user is unfamiliar with the dataset and wants to explore, they 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.

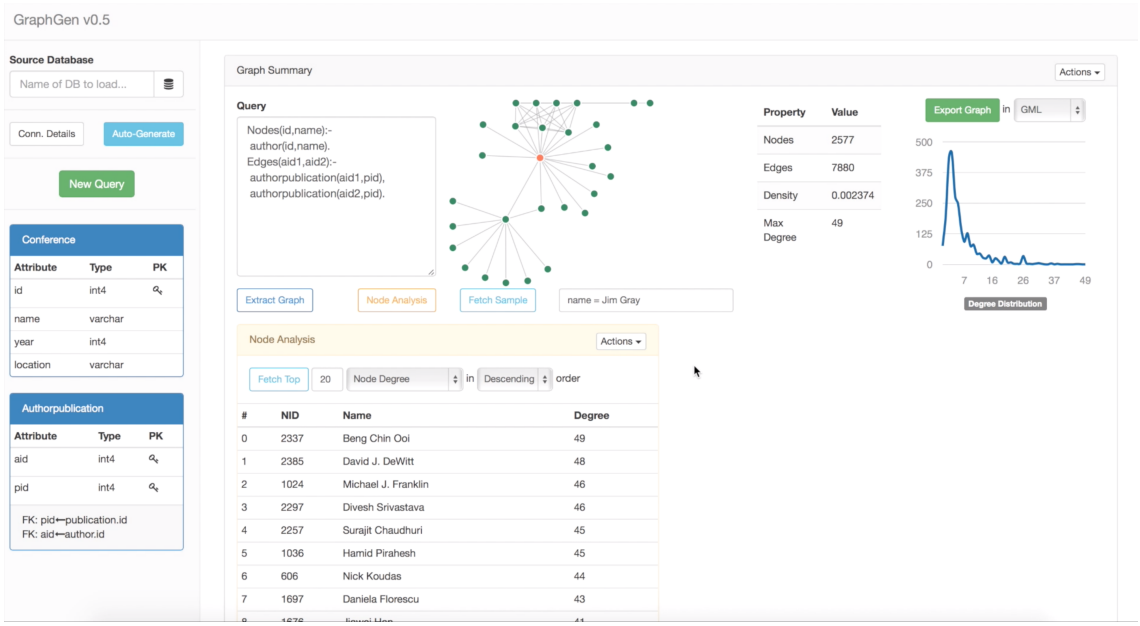


Figure 2.2: The GraphGen explorer web application can connect to a database, load in the schema (left-hand side), and allow users to write extraction queries in GraphGenDL. They can then visualize 1-hop neighborhood samples of the graphs, or conduct standard analysis over them.

Chapter 3: Extracting and Analyzing Graphs in RDBMSs

In this chapter we discuss the ways we tackle the *challenges* we encountered while building GraphGen, focusing on the extraction and analysis of a *single* graph at a time (we discuss collections of graphs in Chapter 4). Specifically we discuss how we can efficiently extract and analyze graphs that are "hidden" within structured databases when these graphs are not explicitly materialized in the RDBMS. Note that these hidden graphs can either be simple graphs or multigraphs.

In Section 3.1, we review the problem of analyzing hidden graphs and how we tackle that problem with GraphGen. In Section 3.2, we discuss a novel *condensed* representation that we use for storing hidden graphs and discuss how to extract it, and why it is ideally suited for this purpose. We also discuss the *duplication* issues that come with this representation. In Section 3.3 we propose a series of in-memory variations of the basic condensed representation that handle the duplication issue through uniquely characterized approaches, each of which results in one of the condensed representations we have developed. These representations are products of a single-run *preprocessing* phase on top of the condensed representation using an array of algorithms that are also discussed in detail in Section 3.3. Lastly, we study the potential of these representations, as well as the benefits and trade-offs that are

associated with them in Section 3.4 and delve into the details of our experimental setup in Section 3.5.

3.1 Overview

This section provides a quick overview of the challenges in analyzing hidden graphs within RDBMSs, in order to set up the work that is presented in the rest of the chapter towards tackling those challenges.

3.1.1 Review: Hidden Graphs and Challenges

We define the term "hidden" graphs as graphs for which the `Edges` table is not explicitly materialized in the RDBMS, but rather needs to be computed by combining various other tables through joins. There can be a large variety of hidden graphs inside RDBMSs that users might be interested in analyzing. 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*) (we discuss how GraphGen enables these analysis tasks in Chapter 4).

To reiterate a few earlier examples, in a DBLP dataset of authors, publications and conferences, one of the many interesting graphs hidden within is (a) a *co-authors* graph, where there is a node for every author and two authors are connected by an undirected edge if they have published a paper together, or (b) a *co-attendance*

graph, where an edge between two authors indicates that they attended a conference together (detailed examples can be seen in Example 1.3.1 and Example 1.3.2).

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 (queries which may have trouble completing their execution 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.

3.1.2 Analyzing Hidden Graphs with GraphGen

As previously discussed in Chapters 1 and 2, our GraphGen system aims to make it easy for users to extract a variety of different types of graphs from an RDBMS, and execute graph analysis tasks or algorithms over them in memory. As depicted in Figure 3.1, GraphGen is a lightweight software layer on top of RDBMSs which provides users *efficient* access to hidden graphs within those RDBMSs, through a variety of different interfaces.

GraphGen supports a DSL called GraphGenDL based on *Datalog* [48], allowing users to specify a single graph or a collection of graphs to be extracted from

the RDBMS (in essence, as *views* on the database tables). GraphGenDL works by allowing the user to map sets of nodes and edges to *views* of the underlying database. GraphGen uses a translation layer to 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 3.2 shows a toy DBLP-like dataset, and the query that specifies a *co-authors* graph to be constructed. Figure 3.2c shows the requested co-authors graph (GraphGen naturally extracts *directed graphs*, and undirected graphs are represented using *bidirectional* edges).

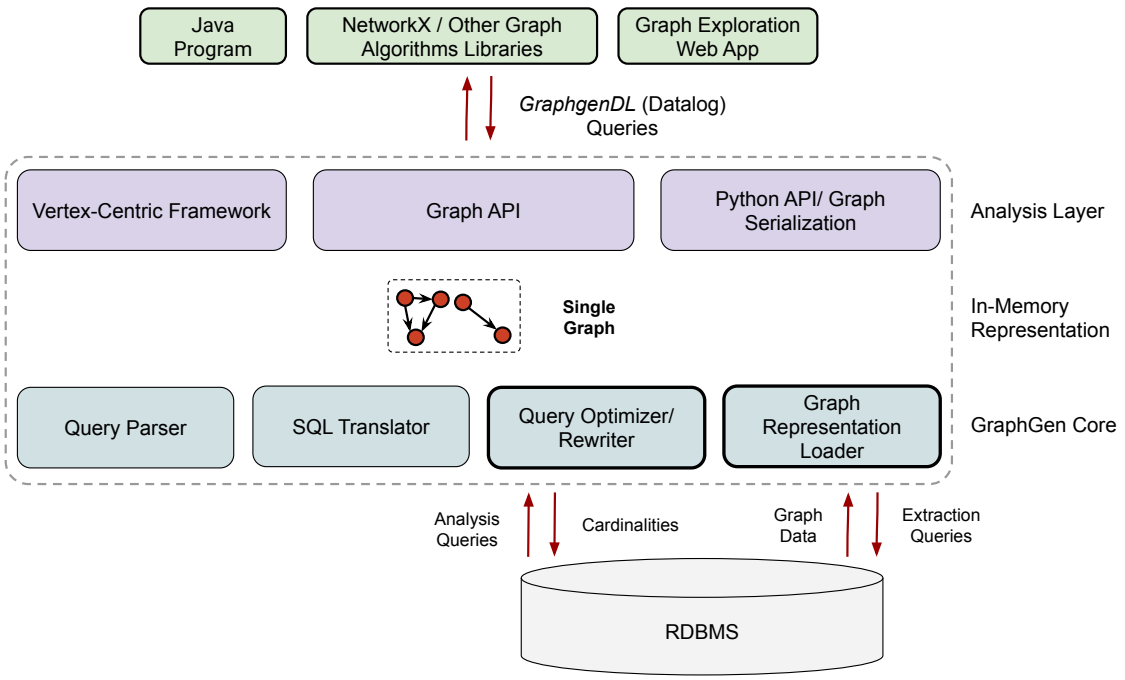


Figure 3.1: GraphGen Overview for analyzing a *single* graph.

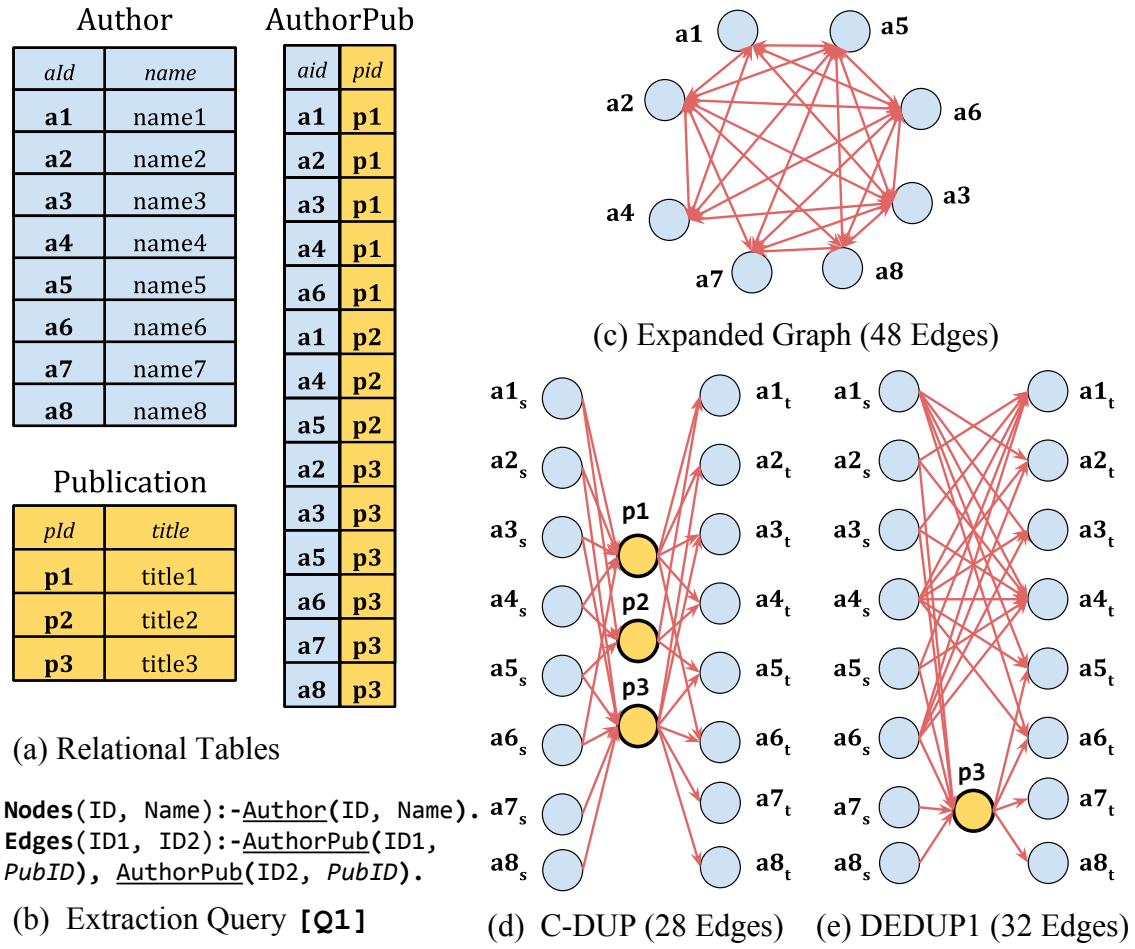


Figure 3.2: Key concepts of GraphGen. For C-DUP and DEDUP-1, the author nodes are shown twice (with subscripts $_s$ and $_t$) to avoid clutter (by separating the in-edges and out-edges); physically they are *not* stored separately.

3.1.3 Condensed In-memory Representations and Duplication

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 (large-output) joins. Because of this, the extracted graph may be much larger than the input size itself. Instead, we propose maintaining and operating upon the extracted graph in a *condensed*

fashion. Table 3.1 shows several examples of this phenomenon for different types of graphs hidden in different datasets.

The co-author graph (from the DBLP dataset) described in an earlier examples is, in some sense, a *best-case* scenario since the average number of authors per publication is relatively small. Constructing a *co-actors* graph from the IMDB dataset results in a similar space explosion. Likewise, a graph connecting pairs of customers who bought the same item in a small sample of the TPCH dataset results in a graph much larger 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.

Graph	Representation	Edges	Extraction Time (s)
DBLP	Condensed	17,147,302	105.552
$10M$ rows	Full Graph	86,190,578	$>$ 1200.000
IMDB	Condensed	8,437,792	108.647
$4.7M$ rows	Full Graph	33,066,098	687.223
TPCH	Condensed	52,850	15.52
$765K$ rows	Full Graph	99,990,000	$>$ 1200.000
UNIV	Condensed	60,000	0.033
$32K$ rows	Full Graph	3,592,176	82.042

Table 3.1: Extracting graphs in GraphGen using our *condensed* representation (C-DUP) vs extracting the full graph (EXP). GraphGen enables scalable extraction and analysis on graphs that may not fit in memory. 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*.

We show how to analyze such large graphs by storing and operating upon them using a novel *condensed* representation, for extraction queries that are equivalent to unions of acyclic conjunctive queries without aggregations.

The relational model provides a natural such condensed representation for

queries like this, that we call *C-DUP*, obtained (essentially for free) by omitting some of the large-output joins from the query required for graph extraction. Figure 3.2d shows an example of C-DUP 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_s to v_t in C-DUP. This representation generalizes the idea of using cliques and bicliques for graph compression [49, 50]; however, the key challenge for us is not generating the representation, but rather dealing with *duplicate paths* between two nodes.

In Figure 3.2, we can see such a duplication for the edge $a1 \rightarrow a4$ since they are connected through both $p1$ and $p2$. This duplication problem 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 "deduplication" 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 (one of which, called DEDUP-1, is shown in Figure 3.2e).

The rest of this chapter focuses on queries in which 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 3.2.2). This corresponds to the edges being constructed using a *union of acyclic conjunctive queries*, and covers many natural graph extraction tasks, including all the examples discussed so far. Even for this class of queries, extracting and operating upon the graph in a condensed form is computationally challenging.

3.2 In-Memory Representation and Task Execution

This section describes the algorithm by which the condensed representation is extracted, delves into the *duplication* problem inherent in the condensed representation, and introduces a series of in-memory representations that are designed to deal with the duplication problem.

3.2.1 Condensed Representation & Duplication

The idea of compressing graphs through identifying specific types of structures has been around for a long time [49, 51]. Those prior techniques (see Chapter 6) are not directly applicable here since they require the input graph to exist in an expanded form *before* compression can take place. Instead, we propose a novel condensed representation, called C-DUP, that is effectively free to construct from the database and requires less memory to maintain. 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. Since in the C-DUP representation there are only edges between *real* nodes and *virtual* nodes, we denote vertices with subscript $_s$(source) to describe the vertices that have out-edges to virtual nodes, and vertices with subscript $_t$(target) for vertices that have in-edges from virtual nodes (for now we assume there is only a single large-output join in the query). 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 3.3 shows two examples of such condensed graphs, the extraction queries for which can be seen in Listing 3.4. In the second case, where a heterogeneous bipartite graph is being extracted, there are no outgoing edges from $s1_s, s2_s, s3_s$ or incoming edges to $i1_t, i2_t$, 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). There may be self-edges in the extracted graph (e.g., $c1_s \rightarrow c1_t$ in Figure 3.3a); however, since our extraction queries are acyclic, G_C itself is always a directed acyclic graph (DAG).

Duplication Problem: Although the C-DUP representation is easy to construct, it 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 (i.e., "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); the notion of *representation-independent graph analytics* from recent work could be used to further increase the applicability of the C-DUP representation [52,53]. However, this *duplication* causes

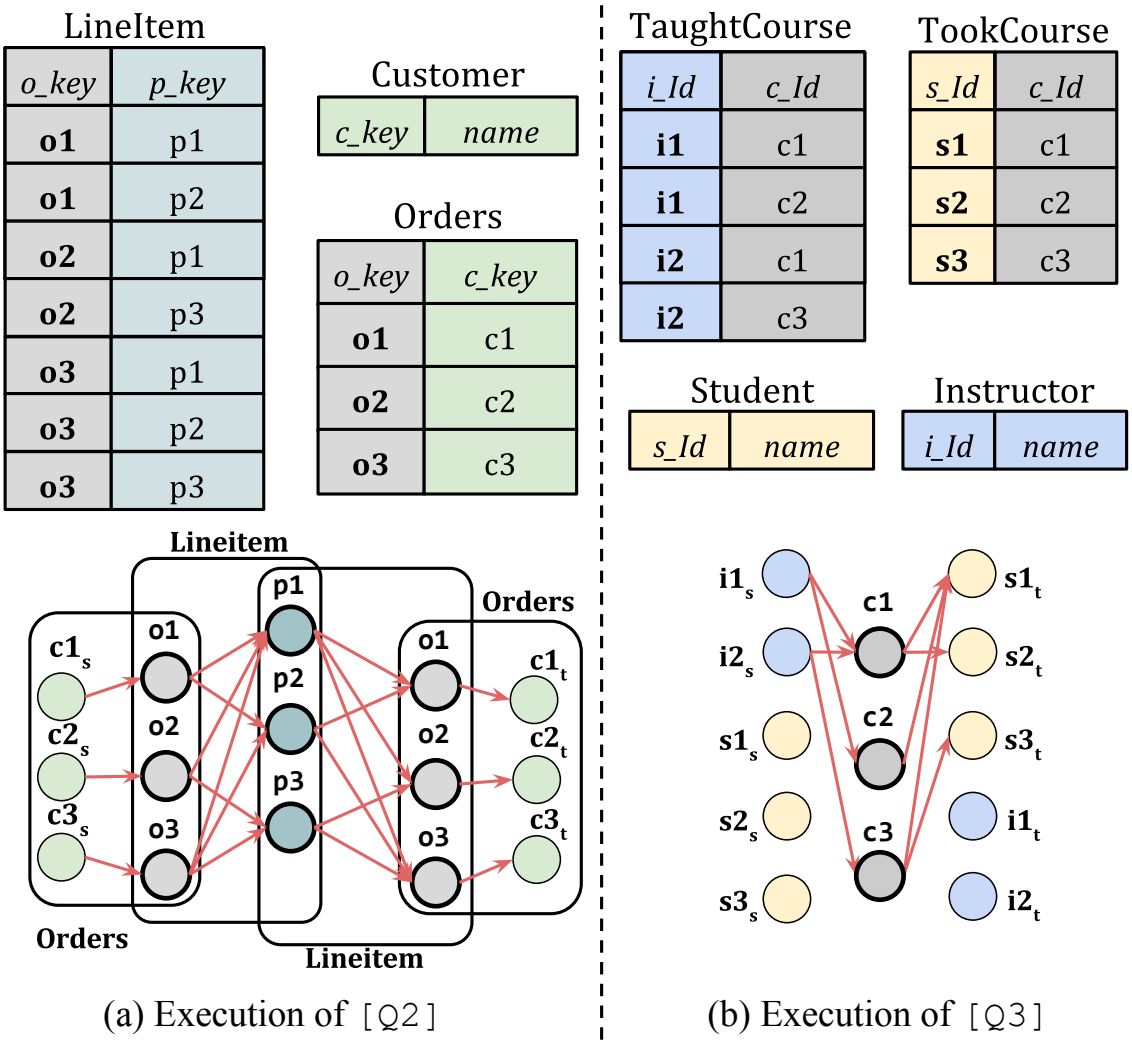


Figure 3.3: Extraction examples: (a) Multi-layered condensed representation, (b) extracting a heterogeneous bipartite graph (we only list the schemas for some of the tables, and omit tuples for clarity).

```
[Q2]
  CREATE GRAPHVIEW customers_items
     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).

[Q3]
  CREATE GRAPHVIEW instructors_students
     Nodes(ID, Name) :- Instructor(ID, Name).
     Nodes(ID, Name) :- Student(ID, Name).
     Edges(ID1, ID2) :- TaughtCourse(ID1, courseId), TookCourse(ID2, courseId).
```

Figure 3.4: Graph Extraction Query Examples (see Figure 3.2 for [Q1]).

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 3.3 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). Each layer of virtual nodes represents a large-output join in the graph extraction query. In the majority of cases, most of the joins involved in extracting these graphs will be simple key-foreign key joins, and large-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.

3.2.2 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 then putting it through a pre-processing phase 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 (*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. 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 *large-output* join and mark it so (this formula assumes that the join attribute is uniformly distributed and may miss a large-output join and could be easily substituted with a more sophisticated selectivity estimator).

Step 3: We then consider each subsequence of the relations without a large-output join, construct an SQL query corresponding to it, and execute it against the database. Let $a_l, a_m, ..., a_u$ denote the join attributes which are marked as *large-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 $attr \in \{a_l, a_m, ..., a_u\}$, 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 (Preprocessing): For a virtual node, let in and out denote the number of incoming and outgoing edges respectively; if $in \times out \leq (in + out + 1)$, we "expand" this node, i.e., we remove it and add directed edges from its in-neighbors to its out-neighbors. This preprocessing 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. Finally, the system also computes the number of edges in the expanded graph (this can be computed for free as a side-effect of all of our deduplication algorithms), and expands the graph if the increase in size is small.

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.

In the example shown in Figure 3.3a, the graph specified in query `[Q2]` that is extracted assumes that all three of the joins involved are *large-output* joins, 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.

Example 3.2.1. Figure 3.4 demonstrates several 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 3.2, only requires a single join (in this case a self-join on the `AuthorPub` table), while `[Q2]` as shown in Figure 3.3a would require a total of 3 joins, some of which (in this case Orders(**order_key1**, ID1) ⋈ LineItem(**order_key1**, part_key), and Orders(**order_key2**, ID2) ⋈ LineItem(**order_key2**, part_key)) will be handed off to the database since they are small-output key-foreign key joins.

The extraction query `[Q3]` extracts a bi-partite (heterogeneous) directed graph between instructors and students who took their courses (Figure 3.3b).

3.2.3 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 *deduplicated* state. Here we discuss the representation formats and their key properties, and with a specific focus on the implementation of the `getNeighbors()` iterator, which underlies most graph algorithms. We note that, in some cases, we use the same term to both denote an in-memory representation, as well as the algorithm for constructing that representation; e.g., we discuss the DEDUP-1 representation below

and outline its key properties (e.g., it does not suffer from edge duplication), and we discuss several algorithms for constructing the DEDUP-1 representation in the next section. We note that our representations primarily explore different ways to do structural compression, and could be combined with other graph compression approaches [54] to further reduce the memory footprint.

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 deduplication, i.e., by doing deduplication *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 (using a *hashset*) and skips over them if the neighbor is seen again.

This is typically the most storage-efficient representation, does not require any preprocessing 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 Deduplicated Representation: This representation format is identical to C-DUP in its use of virtual nodes, with the major difference being that it does not suffer from duplicate paths, and thus does not require the on-the-fly deduplication used in C-DUP (i.e., `getNeighbors()` does not need to use the *hashset*). 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 has reduced overhead of neighbor iteration. The trade-offs here also include the one-time cost of removing duplication; deduplicating 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 the appropriate iterator.

BITMAP: Deduplication using Bitmaps: This representation results from applying a different kind of preprocessing based on maintaining *bitmaps*, for *filtering*

out duplicate paths between nodes. Specifically, a virtual 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 is a bitmap corresponding to u_s; if not, we traverse each of the outgoing edges in sequence. 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., for every bit that is set to 1, we traverse the corresponding 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 preprocessing 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:

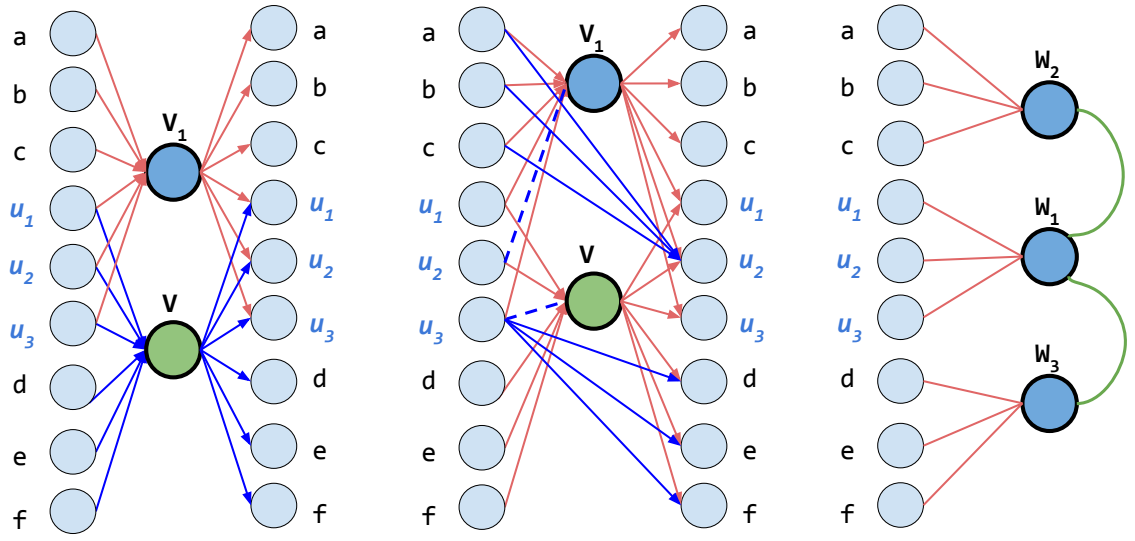
This optimized representation can significantly reduce the memory requirements for dense graphs, for the special case of a single-layer, *symmetric* condensed graph (i.e., $\langle u_s \rightarrow v_t \rangle \implies \langle v_s \rightarrow u_t \rangle$); many graphs satisfy these conditions. In such 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 3.5 illustrates an example of the same graph if we were to use all three deduplication 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 3.5b) results in a substantial increase in

the number of edges, because of the large number of duplicate paths. The DEDUP-2 representation (Figure 3.5c) 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 (Section 3.4).

Generating a good DEDUP-2 representation for a given C-DUP graph is much more intricate than generating a DEDUP-1 representation. We discuss a sketch of the algorithm for generating a DEDUP-2 representation in Section 3.3.3.

3.3 Preprocessing & Deduplication

In this section, we discuss a series of preprocessing and deduplication algorithms we have developed for constructing the different in-memory representations for a given query. The input to all of these algorithms is the C-DUP representation, that has been extracted and instantiated in memory. We first present a general preprocessing algorithm for the BITMAP representation for multi-layer condensed graphs. We then discuss a series of optimizations for single-layer condensed



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

Figure 3.5: 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).

graphs, including *structural* deduplication algorithms that eliminate duplication (i.e., achieve DEDUP-1 representation). In contrast with Section 3.2.3, here we focus on describing algorithms for *how* to generate the representations that we described there. 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).

3.3.1 Preprocessing for BITMAP

Recall that the goal of the preprocessing phase here is to associate and initialize 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.

3.3.1.1 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 to V that is 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 so, 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 maintaining 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. The pseudo-code for BITMAP-1 can be found in Algorithm 1.

3.3.1.2 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 $Out(V_1)$ 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 $Out(V_1), ..., Out(V_n)$ that covers their union. Unfortunately, the set cover problem is not only NP-Hard, but is also known to be hard to approximate.

3.3.1.3 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 $|Out(V_i)|$. It adds a bitmap for u to V_i, and sets all of its bits to 1; all nodes in $Out(V_i)$ are now considered to be *covered*. It then identifies the virtual node V_j with the largest $|Out(V_j) - Out(V_i)|$, 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 (which are part of the first layer of virtual nodes). 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. We consistently 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.

It's important to note that here 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 3.6 is unreachable by every u_s).

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. The pseudo-code for BITMAP-2 can be found in Algorithm 2.

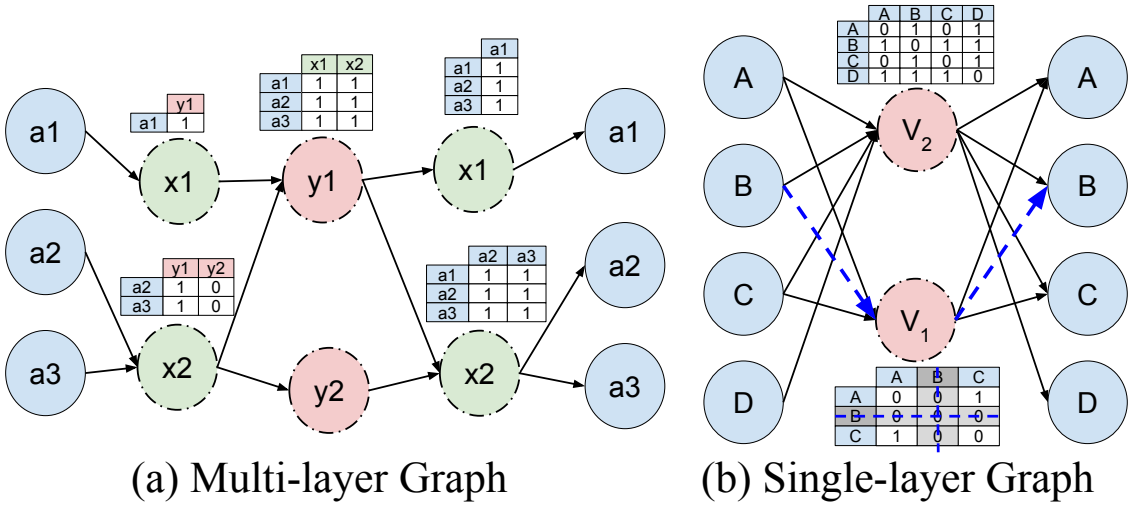


Figure 3.6: Using *BITMAP*s to handle duplication; the dotted edges (corresponding to columns or edges with all 0s) are removed.

3.3.2 Deduplication for DEDUP-1

The goal with deduplication 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 deduplication for multi-layer graphs to future work.

3.3.2.1 Single-layer Condensed Graphs

The theoretical complexity of this problem for single-layer condensed graphs is the same as the original problem considered by Feder and Motwani [51], which focuses on the reverse problem of *compressing cliques* that exist in the expanded graph, by finding cliques and connecting all vertices in the same clique to a virtual node and removing the edges among them. Although the expanded graph is usually very large, it is still only $O(n^2)$, so the NP-Hardness of the deduplication problem is the same. However, those algorithms presented in [51] are not applicable here because the input representation is different, and expansion is not an option. We present four algorithms for dealing with this problem.

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

Naive Virtual Nodes First: This algorithm deduplicates 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 $In(V) \cap In(R_i) \neq \phi$; these are the virtual nodes that all the real nodes in $In(V)$ point to (*other* than V). 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 $|Out(V) \cap Out(R_i)| > 1$, 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 $r \in Out(V) \cap Out(R_i)$ at *random*, and choose to either remove the edge $(V \rightarrow r)$ or $(R_i \rightarrow r)$, depending on the in-degrees 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 $In(V)$, 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 deduplication 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 deduplicated, 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 deduplicate it individually. Figure 3.7 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 deduplication 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.

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 $E = \{(u_s, x) | x \in \cup_{V \in \mathcal{V}} Out(V)\}$. 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 $X = \cup Out(\mathcal{V}')$ 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 $Out(V) \cap X$ – otherwise there would be duplicate edges between u and those nodes. Then, for any $a, b \in Out(V) \cap X$, 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 $r_i \in Out(V) - Out(V) \cap X$, 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 $O(n_r * d^5)$. The pseudo-code for the Greedy Real-Nodes-First algorithm can be found in Algorithm 4.

Greedy Virtual Nodes First Algorithm: Exactly like the naive version above, this algorithm deduplicates the graph one virtual node at a time, maintaining a deduplicated partial graph at every step. We start with a graph containing only the real nodes and no virtual nodes, which is trivially deduplicated. 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., $|Out(V) \cap Out(V_i)| \geq 2$). Let $C_i = Out(V) \cap Out(V_i)$, denote the real nodes to which both V and V_i are connected. At least $|C_i| - 1$ of those edges must be

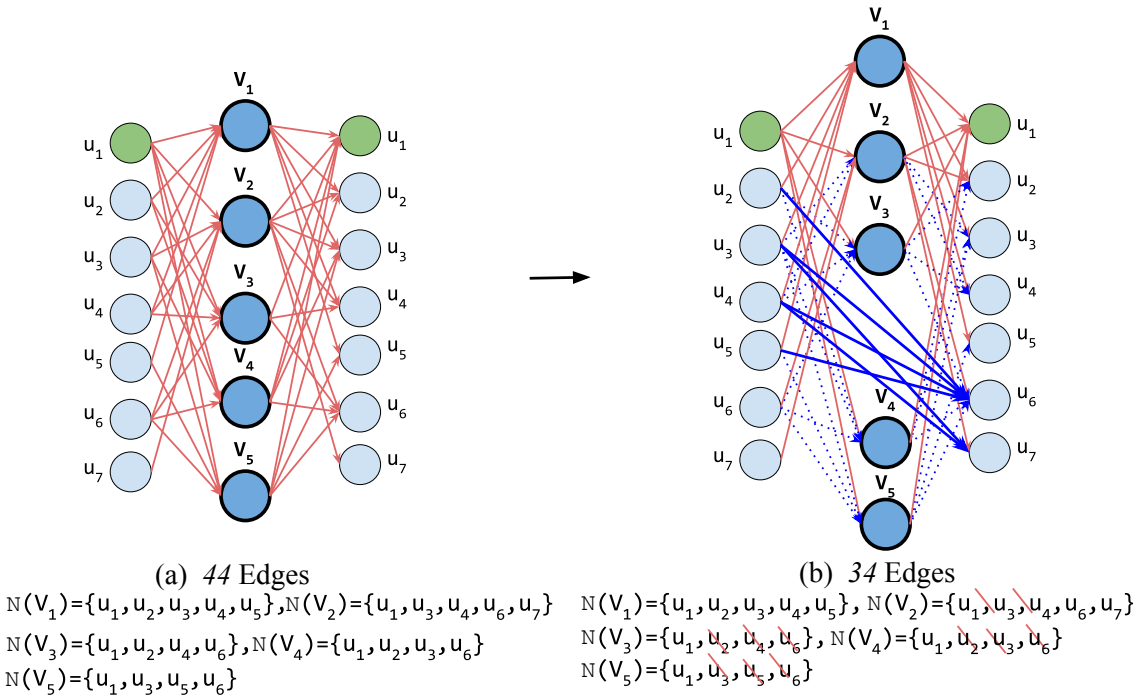


Figure 3.7: Deduplicating u_1 using the "real-nodes first" algorithm, resulting to an equivalent graph with a *smaller* number of edges.

removed from $Out(V)$ and $Out(V_i)$ 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 $Out(V_i)$ versus from $Out(V)$. 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 $Out(V_i)$ always yields a $benefit$ of 1, whereas removing it from $Out(V)$ 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: $O(n_v d(n_v d^2 + d))$. The pseudo-code for the Greedy Virtual-Nodes-First algorithm can be found in Algorithm 3.

We note that, these complexity bounds listed here make worst-case assumptions and in practice, most algorithms run much faster.

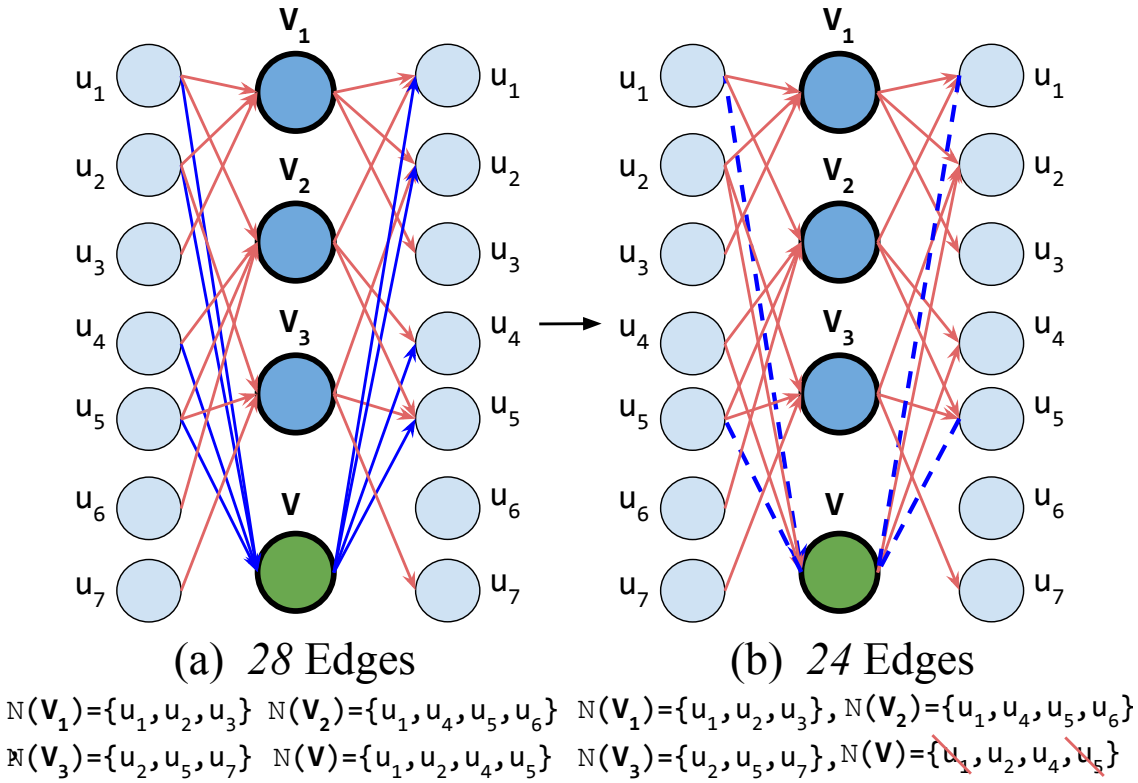


Figure 3.8: Deduplication using Greedy Virtual Nodes First.

3.3.2.2 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 $Out(V_1) \cap Out(V_2) \neq \phi$ and $In(V_1) \cap In(V_2) \neq \phi$, 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 simply 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 if possible (through expansion of all virtual nodes in all but one layer) and then using one of the algorithms developed above; note that the latter approach should only be considered if the expansion does not result in a space explosion.

3.3.3 DEDUP-2 Greedy Algorithm

Conducting deduplication for outputting the DEDUP-2 representation turns out to be significantly more challenging than DEDUP-1 and we only present one algorithm for doing so (a few other variants that we tried turned out to be too complex and inefficient, without any performance benefits). Because the Virtual

Nodes First algorithm reasons about a virtual node at a time, it turns out to be most amenable to adding such edges between virtual nodes. Figure 3.5 shows an example of the different outcomes after incrementally introducing a new virtual node V to a condensed graph containing only a single virtual node currently (V_1). Figure 3.5c shows the resulting graph after one applies our DEDUP-2 greedy algorithm to the condensed duplicated graph shown in Figure 3.5a. This graph now includes edges *between* virtual nodes. The addition of this new *type* of edge to the mix immediately makes deduplication substantially more complex, as *more invariants* need to now be checked in attempting to add the new virtual node into the deduplicated partial graph in order to maintain correctness.

Below we present an algorithm for adding a new virtual node V into a partially constructed, deduplicated graph. This algorithm does not add direct edges between real nodes; instead we introduce the notion of a *singleton* virtual node both for the purpose of implicitly adding direct edges as well as to deal with correctness issues. A *singleton* virtual node is one with only a single real node attached to it.

Step 1: Identifying Violations: We first identify the set of virtual nodes that overlap with V and may potentially lead to a violation. Let $\mathcal{V} = \{V_1, ..., V_n\}$ denote all the virtual nodes in the partial condensed graph constructed so far, that share at least 1 real node with V; these are all the virtual nodes that we need to check for violations when adding V. There are two types of violations that could potentially arise: (1) there exists a virtual node V_1 such that $|V \cap V_1| > 1$, (same as in all of the other representations and algorithms) as well as (2) there exists two *other*

virtual nodes V_1 and V_2 that are connected to *at least one common virtual node* C where $|V_1 \cap V_2| > 0$. More intuitively, violation (1) says there should be no overlap of more than 1 between any two virtual nodes, while (2) says that at any point in the graph, the virtual *neighbors* of any one virtual node should have zero overlap *with each other*. The reason (2) constitutes a violation comes up in iteration (`getNeighbors()`) in this representation, where that scenario would lead to the same real node being returned as a neighbor multiple times.

Step 2: Edges Between Virtual Nodes: Let $V_1 \in \mathcal{V}$ denote the virtual node with the highest overlap with V.

If V has a high overlap with V_1, then removing this violation by adding direct edges (as above) could result in the addition of many direct edges between the real nodes (as seen in the example in Figure 3.5b). Hence, if $|V \cap V_1| \geq 1$ we *split* both V and V_1 and create 4 different virtual nodes so as to correctly incorporate V's nodes into the partially deduplicated graph. These virtual nodes are: (1) $W_1 = V_1 \cap V$, (2) $W_2 = V_1 - W_1$, (3) $W_3 = V - W_1 - \cup N(V_1)$, (4) $W_4 = V - W_1 - W_3$, where $\cup N(V_1)$ is the union of real nodes in V_1's virtual neighborhood (some of these might be empty and would not be created).

To explain the intuition behind the above splits, we must explain how the algorithm works. The basic idea is to observe the current state of the deduplicated graph and see which virtual nodes need to be *split* in order to make way for correctly adding V; this results into W_1 and W_2. After that we need to keep track of which edges need to be maintained, while recursively *adding* in the portions of V that

could potentially cause violations (W_3 and W_4). The most important part in the implementation of this algorithm is to not actually add *any* edges between virtual nodes unless we are certain that these edges will not lead to any violations of the aforementioned invariants. To achieve this, we maintain a data structure `m` that includes all the edges that need to be added at any point in the execution, and after all the appropriate checks are made, only then are those edges physically added.

Processing each new virtual node V can be described in smaller sub-steps:

1. **Substep 1**: Since $W_1 = V \cap V_1$, it will need to be a separate virtual node that both V and V_1 will need to haves edges to. The intention of the first split is replacing V_1 with W_1 and W_2 where $W_1 \leftrightarrow W_2$. After this split, W_1 and W_2 also need to be connected to all the previous neighbors of V_1. This split can be applied immediately in the deduplicated graph as it does not alter the properties of the graph in any way. We keep track of the fact that W_1 and $V - W_1$ need to be connected after all checks are done by adding this potential edge in `m`.
2. **Substep 2**: For simplicity, let virtual node $W_3' = V - W_1$ which includes the rest of the real nodes that are *not* included in the initial split. We check if there are are any neighbors of V_1 that have *any* overlap with W_3', and if so, these nodes will constitute W_4, and the rest of the nodes will constitute W_3. If W_4 is not empty, we add the edge $W_3 \leftrightarrow W_4$ to `m` as well as $W_3 \leftrightarrow W_1$. If however there is a previous constraint in `m` for the virtual edge $W_3' \leftrightarrow W_1$, then that also needs to be split into two constraints inside `m`.
3. **Substep 3**: Recursively call the above on $V = W_3$ and then on $V = W_4$,

passing the same `m` into every call.

4. **Substep 4**: Physically add in all the edges that are described in `m` and clean up any virtual nodes that need to be deleted.

Complexity: The runtime complexity of this algorithm is hard to calculate precisely, but is upper bounded by $O(n_r * d^8)$. Please refer to Algorithm 5 for the pseudo-code.x

3.4 Experimental Study

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. We then compare the performance of the different deduplication algorithms, and 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.

3.4.1 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 3.2), 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 den-

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

sities. 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, which we sketch in Section 3.5.1.

3.4.1.1 Compression Performance

We begin with comparing the graph sizes for the different representations for each dataset. In addition to the in-memory representations presented in this chapter, we also implemented and compared against a prior graph compression algorithm, called *VMiner* (*Virtual Node Miner*) [49]. VMiner uses *frequent pattern mining* to identify *bi-cliques* in the graph, i.e., groups of nodes A and B, such that for $u \in A, v \in B$, there is an edge $u \rightarrow v$. It then repeatedly replaces such bicliques with *virtual nodes*; i.e., it adds a new virtual node C to the graph, adds an edge $u \rightarrow C, \forall u \in A$ and $C \rightarrow v, \forall v \in B$, and deletes all edges from A to B. It makes multiple passes through the graph, iteratively reducing its size. The final representation thus looks very similar to DEDUP-1, and is also duplication-free. VMiner has several parameters which we exhaustively tried out combinations of, for each of our datasets, (following the guidance in the paper) and picked the best. Note that using VMiner requires us to first expand the graph, which makes it infeasible

for several of the large datasets discussed in Section 6.2.

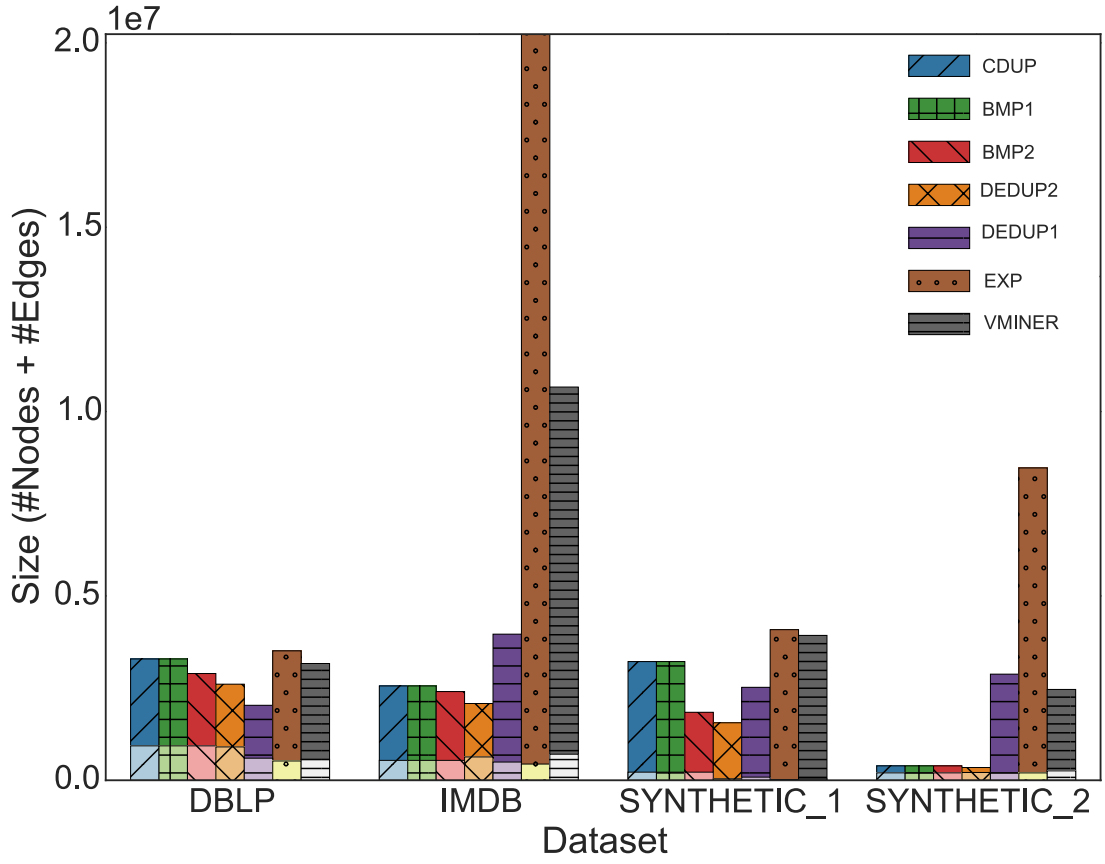


Figure 3.9: Comparing the in-memory graph sizes for different datasets; the bottom (lighter) bars show the number of nodes.

Figure 3.9 shows how the different algorithms fare against each other. For each algorithm and each dataset, we report the total number of nodes and edges, and also show the breakdown between them; the algorithm used for *DEDUP-1* was Greedy Virtual Nodes First, described in Section 3.3.2.1. When the average degree of virtual nodes is small and there is a large number of virtual nodes (as is the case with *DBLP* and *Synthetic_1*), we observe that there is a relatively small difference in the size of the condensed and expanded graphs, and deduplication (*DEDUP-1* and *DEDUP-2*) actually results in an even smaller footprint graph.

On the other hand, the *IMDB* dataset shows a 8-fold difference in size between EXP and C-DUP and over a 5-fold difference with all other representations. *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 *DEDUP-2* representation can be significantly more compact than *DEDUP-1*, while maintaining its natural, more portable structure compared to the BITMAP representations. As we can see, VMiner not only requires expanding the graph first, but also generally finds a much worse representation than DEDUP-1. This corroborates our hypothesis that working directly with the implicit representation of the graph results in better compression.

We also measured actual memory footprints for the same datasets, which largely track the relative performance shown here, with one major difference being that BITMAP representations perform a little worse because of the extra space required for storing the bitmaps. We report memory footprints for larger datasets in Section 3.4.2.

3.4.1.2 Graph Algorithms Performance

Figure 3.10 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 (BFS)* 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 2.3.1, 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 BFS 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.

We also ran a comprehensive set of microbenchmarks comparing the performance of the basic graph operations against the different representations. Those results can be found in Section 3.4.3, and as can be seen there, BFS and PageRank both follow the trends of the micro-benchmarks in terms of differences 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 DEDUP-1 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 DEDUP-1 and BITMAP-2 typically perform better; they feature a smaller number of virtual neighbors per real node than representations like C-DUP and BITMAP-1, and sometimes DEDUP-2 as well.

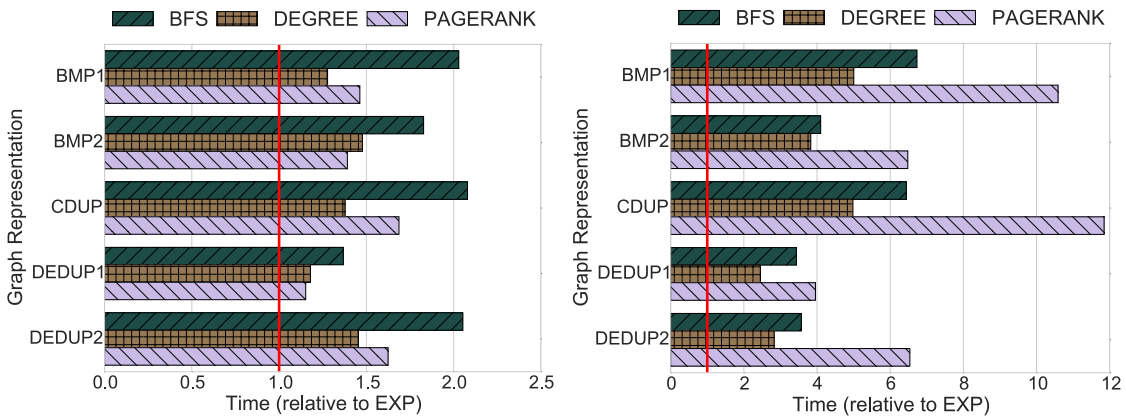


Figure 3.10: Performance of Graph Algorithms on Each Representation for the DBLP dataset (left) and for the Synthetic_1 dataset. The vertical red line represents EXP.

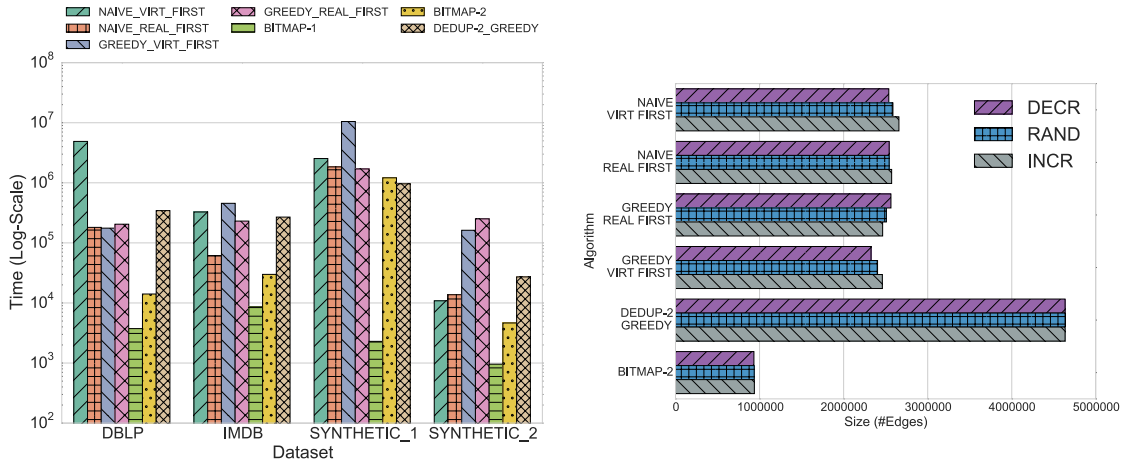


Figure 3.11: Deduplication Performance Results (a) Deduplication time comparison between algorithms. Random (RAND) vertex ordering was used where applicable, (b) Small variations caused by node ordering in deduplication.

3.4.1.3 Comparing Deduplication Algorithms

Figure 3.11a shows the running times for the different deduplication 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 deduplication 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 3.11b 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 across various datasets, and recommend using the random ordering for robustness.

3.4.2 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 3.3). Datasets *Layered_1* and *Layered_2* are synthetically generated multi-layer condensed graphs, while *Single_1*, *Single_2* are standard single-layer condensed graphs (see Section 3.5.2 for details on how these datasets were generated). At this scale, only the C-DUP, BITMAP-2, and EXP are typically feasible options, since none of the deduplication algorithms (targeting DEDUP-1 or DEDUP-2) run in a reasonable time.

Comparing the memory consumption, 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 preprocessing phase (Section 3.2.2), 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

BITMAP 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 deduplication 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 deduplication 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.

3.4.3 Microbenchmarks

We conducted a complete set of micro-benchmarks to evaluate the performance of various graph manipulation operations. Figure 3.12 and Figure 3.13 show 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 3,000 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. *DEDUP-2* is usually least performant here because of the extra layer of indirection that this representation introduces. *DEDUP-1* 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 C-DUP, DEDUP-1 and DEDUP-2 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 DEDUP-1 and DEDUP-2, that employ virtual nodes, we need to remove a smaller number of edges on average in the removal process. DEDUP-2 is most interesting here because a real node is always connected to only 1 virtual node, therefore the removal cost is constant.

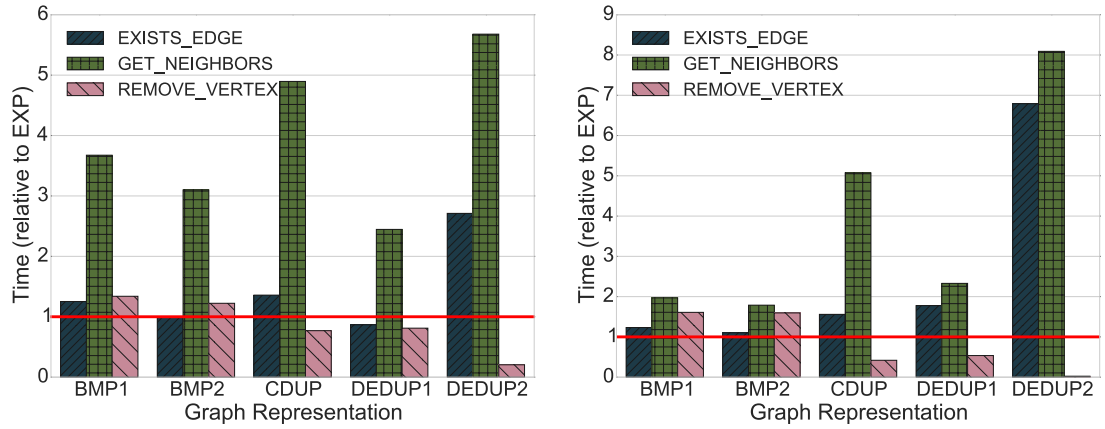


Figure 3.12: Microbenchmarks for the real datasets (a) DBLP and (b) IMDB.

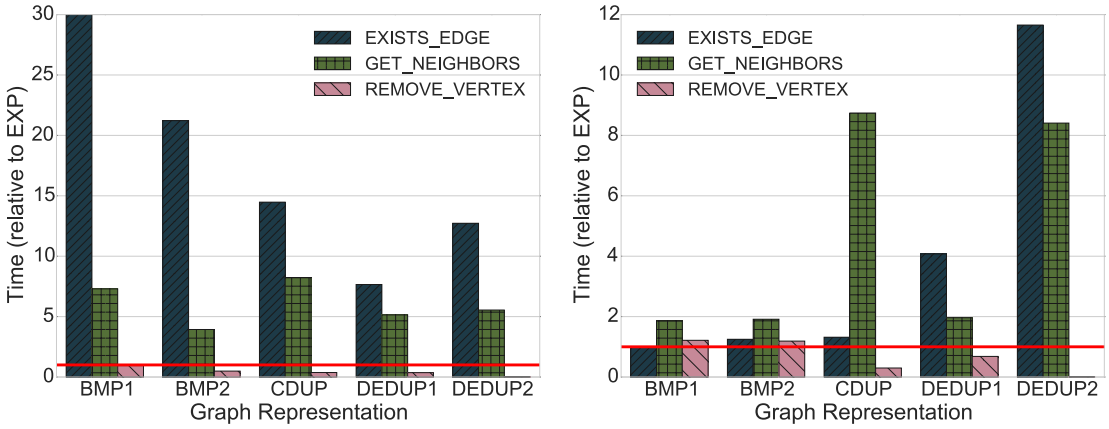


Figure 3.13: Microbenchmarks for synthetic datasets (a) Synthetic_1 and (b) Synthetic_2.

3.4.4 Integration with Apache Giraph

The wide array of representations we propose are significantly more memory-efficient than storing the entire graph (i.e., EXP representation). Further, the C-DUP representation is the easiest and fastest to obtain from the relational database (Table 3.1) and is usually the most memory-efficient. Past work on graph compression aims to compress the expanded graph into one with a smaller memory footprint. These techniques require the initial storage of the entire expanded graph, which is

then processed and compressed, potentially requiring memory even larger than the expanded graph itself. This would however defeat the purpose of our system since its aim is to efficiently extract and operate on top of graphs in relational datasets without requiring extraction and storage of the expanded graph. Our representations are quite generic and can be ported to other graph processing systems with varying degrees of difficulty. In Table 3.4, we showcase the results of running *Degree, PageRank* and *Connected Components* on three representations on a prototype port to Apache Giraph [25], for a diverse set of synthetic datasets created using our generator (see Section 3.5.1). The datasets *S1-2* were created by maintaining the number of real and virtual nodes static, and incrementally increasing the average *size* of each virtual node. For datasets *N1-2* we maintained the average size of each virtual node static, but increased the number of real and virtual nodes. We also ran these experiments over the co-actor graph extracted from portion of the *IMDB* graph. (Table 3.1).

The setup can be seen in Figure 3.14, which showcases the steps we took in order to leverage our representations using Giraph. The porting process was very straightforward–only a single `BMPWritable` datatype needed to be implemented in order to efficiently store a BitSet (used in the BITMAP representations) into HDFS.

We implemented these three algorithms as efficiently as possible for each representation. As seen in Table 3.4, our BITMAP representation almost always outperforms EXP and DEDUP-1, while requiring up to an order of magnitude less memory. Specifically, when using BITMAP, *Connected Components* received a speedup due to the fact that it is *duplicate-insensitive* and can be run directly over C-DUP.

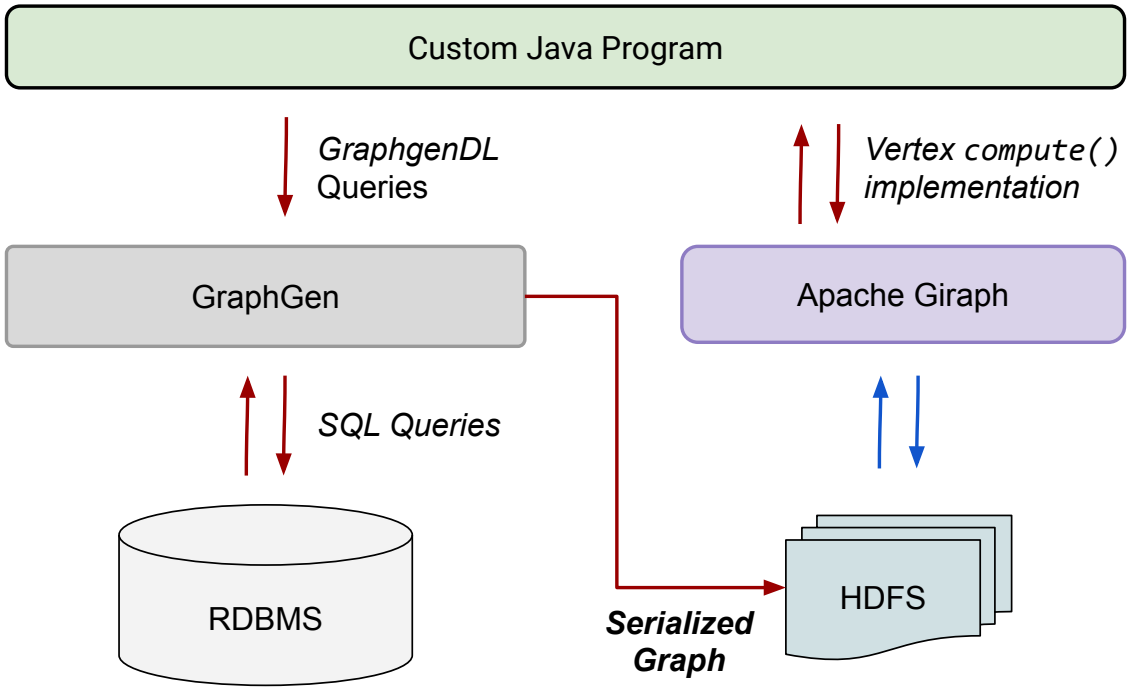


Figure 3.14: Porting GraphGen Representations to Apache Giraph.

Running *PageRank* and *Degree* requires a deduplicated graph, and further, correct execution over DEDUP-1 and BITMAP requires *twice* the number of supersteps. By implementing *message aggregation* at each virtual node, we were able to decrease the number of messages that need to be passed per superstep to only $2 * e$, where e the total number of edges. This resulted in a *speedup* over EXP for the larger datasets. Even though DEDUP-1 was not able to achieve a significant compression over EXP for these datasets, it still outperformed EXP on the larger datasets.

It's interesting to observe the different performance trends seen in the *IMDB* graph. Here we can see that DEDUP-1 ends up being the best alternative in terms of both memory consumption and running times while BITMAP often ends up being second or third in comparison. This skew in the results comes from the difference between the number of nodes in these datasets. The BITMAP representation ends

up having nearly *twice* the number of nodes than EXP, and also shows a substantial difference from DEDUP-1; this difference is due to the *virtual nodes* we need to store for both DEDUP-1 and BITMAP. Also, the fact that we see memory consumption for BITMAP being close to or surpassing EXP in many situations is due to the fact that storing more *nodes* (and in the case of BITMAP, several bitmaps associated with each node), incurs a larger overhead than storing more *edges*. Also, in comparison with the other datasets, the difference in the number of edges between the representations is significantly smaller, which also plays into the fact that BITMAP does not provide as big of an advantage as for the previous datasets. Nevertheless, in situations where a lot of messages need to be sent and received (like when running *PageRank*), the benefits of BITMAP already start showing up, with BITMAP being on par with DEDUP-1, and better than EXP.

One of the fundamental issues that have to be dealt with regarding ports like this is the fact that, some vertex centric algorithms assume direct access to a node's immediate neighbors and therefore assume that calculating the degree for each node is trivial and fast. In the case of our representations, while calculating the degree is trivial, there is no direct access to the immediate neighbors at each node, and therefore the degree cannot be computed on the fly when the vertex-centric framework is used. When the degree needs to be used continuously in a vertex-centric program (e.g., PageRank), it needs to be pre-computed and stored as a vertex property once before the computation begins, otherwise an entire extra superstep would be needed every time only to compute the degree before continuing with the next iteration of the program.

3.5 Experimental Setup

In this section we describe the experimental setup in terms of the way the datasets that we used were generated. We also include details on the database schema used for some of the real datasets which we experimented on, as well as provide examples of the query extraction SQL generated by our system which efficiently extracted the C-DUP representation of these graphs.

3.5.1 Generation of Small Synthetic Datasets

We briefly describe our algorithm for generating small synthetic datasets for the detailed experiments. We needed the ability to generate 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. However, we could not use any of the existing random graph generators for this purpose; this is because we need the graphs in a condensed representation. Instead, we built a synthetic graph generator based on the Barabàsi–Albert model [55] (also called the *preferential attachment model*). that takes as input: the number of real nodes (n_1), the number of *virtual nodes* (n_2), as well as the mean m and the standard deviation sd that define the normal distribution from which we draw the random sizes (degrees) of the virtual nodes.

We sketch the algorithm here:

1. Add all real nodes into the graph at once and *generate* all virtual nodes and

their sizes by sampling the (m, sd) normal distribution.

2. **Initial Splits:** For every virtual node vn, *split* vn into s_1, s_2 with probability relative to its size.
3. **Initial Batch Random Assignment:** Add 15% of the virtual nodes to the graph, and attach real nodes to them at random.
4. **Random or Preferential Attachment:** For each v_i with size x in the remaining virtual nodes, if v_i was derived from a *split*, then with probability 35%, randomly assign real nodes to v_i. Otherwise, randomly select a real node r in the graph that currently has a degree of $d(r) \geq x$. Let s the set of r's neighbors. Assign probabilities to each neighbor s_i as such: $P_i = d(s_i)^2 / \sum d(s_i)^2$. Until $d(r) = x$, remove a real node s_i from s with probability *counter-proportional* to P_i. Real nodes with a high degree, and therefore high P_i value, are more likely to remain in s and thus be attached to v_i.
5. **Cleanup:** Merge the virtual nodes that derived from splitting in step 3, back into their one original virtual node.

This algorithm can be used to generate a graph with similar degree distributions as those generated by the commonly-used *preferential attachment model* [55], while also preserving the local densities typically seen in real-world networks (which the naive preferential attachment model does not preserve [56]).

3.5.2 Generation of Large Datasets

The Layered_1, Layered_2, Single_1, Single_2, datasets are synthetically generated multi-layer and single-layer datasets. Both Layered_1 and Layered_2 have the same layer structure as the TPCH example, as shown in Figure 3.3. The way these were generated was by generating database tables while adjusting the cardinality of the join condition attributes for those tables.The tables were created by randomly generating values in a range of integers (uniformly distributed). More information about the generated datasets can be seen in Table 3.6. The numbers in column joinSelectivities show the selectivity of each join that would be required for creating the full graph. In Layered_1 for instance (since the join structure is the same as TPCH), there are 3 joins that are required across 2 generated tables. Let those tables be A, B; the joins required here were $A \bowtie B$ which had selectivity 0.05, $B \bowtie B$ with a selectivity of 0.1 and again $B \bowtie A$ with selectivity 0.05. The definition of *selectivity* that we use here for a particular join on an attribute a of a table A is $selectivity = distinct\_a/|A|$, where $distinct\_a$ the distinct number of unique values of a.

3.5.3 Database Schemas and Generated SQL

We experimented on various real world datasets, some of which include DBLP, IMDB, and TPCH. The DBLP database includes authors and their publications to conferences, the IMDB database includes information about movies, the actors that acted in them as well as directors, crew, etc. The TPCH database includes

information about Customers and Orders they have placed, as well as information about said orders, suppliers, etc. Figure 3.15 shows subsets of the schemas for these databases. It's important to note that looking at these schemas, users can intuitively come up with various types of graphs that could be extracted and analyzed from these datasets; e.g.,looking at the TPCH Schema (Figure 3.15c), one can see that LineItem has an attribute for the supplier (`LineItem.suppkey`), and therefore, since there also exists a Supplier relation, a graph of suppliers could also be extracted. For the purposes of our experiments, we have extracted the graphs described in Section 3.4 (the IMDB and DBLP graphs in Section 3.4.1 and the TPCH graph in Section 3.4.2).The extraction queries in our Datalog DSL and the resulting SQL queries for these graphs can be seen in Figure 3.16

3.5.4 Discussion: Choosing a Representation

Our experimental evaluation illustrates the pros and cons of the different representations, which leaves us with the question of *which representation should we choose given a particular setting?*. Note that, in several of the experiments, we did not use the preprocessing step (Step 6 in Section 3.2.2) to allow us to more properly compare the different representations. In practice, however, our system *always* uses the one-time preprocessing step, and we further suggest that the graph be expanded if the memory increase is not substantial, e.g., less than 20% (the size of the expanded graph can be calculated relatively quickly from the C-DUP representation). If expanding the graph is not an option (i.e., it doesn't fit in memory), then the

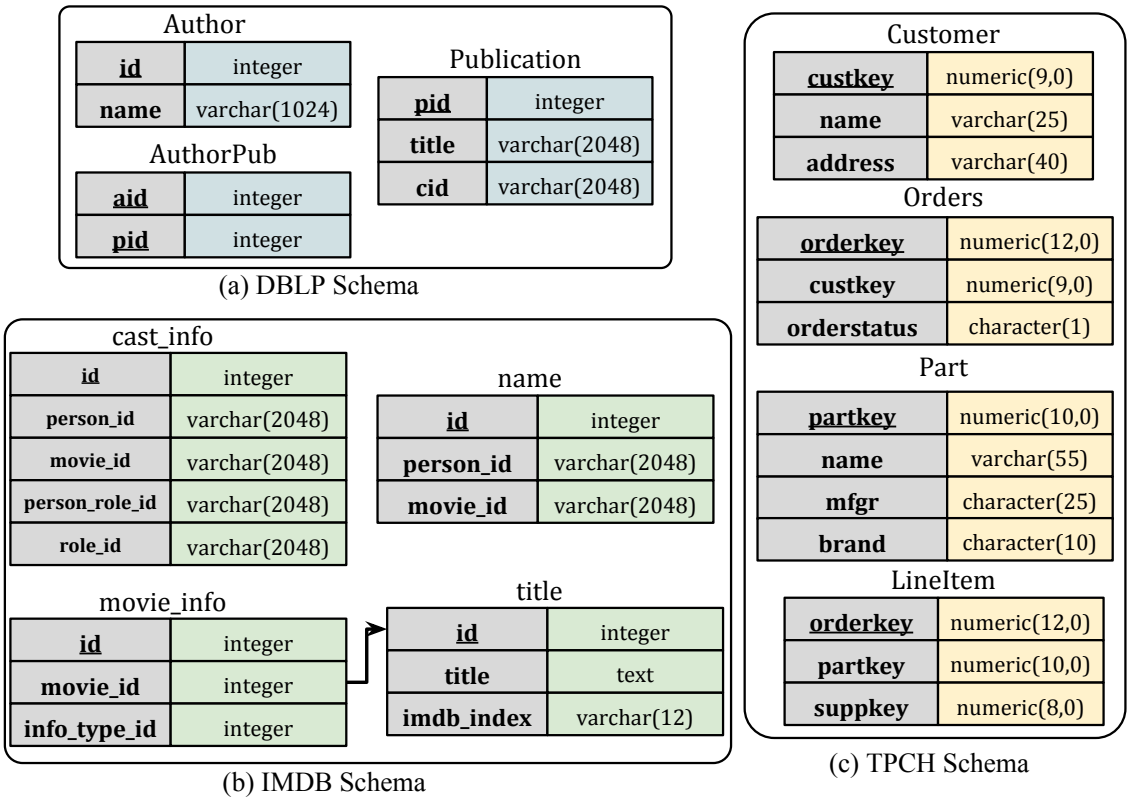


Figure 3.15: Database Schemas: If not explicitly shown, foreign key constraints for each attribute (if any) refer to the the primary key attribute in a different table with the same name.

system needs to choose between C-DUP, BITMAP-2, DEDUP-1, DEDUP-2. These representations are better in different settings, and thus the choice comes down to the use-case. For graph algorithms that don't touch a large portion of the graph, C-DUP is the best option (e.g., *breadth-first search*). BITMAP-2 is preferred for more complex graph algorithms that might make multiple passes on the graph (e.g., *PageRank*). On the other hand, DEDUP-1 and DEDUP-2 should be used if multiple graph algorithms need to be run over a period of time, to amortize the cost of constructing those; in those cases, it might even be a good idea to store the deduplicated graphs back into the database, with the caveat that changes to the

```
Nodes(ID, Name):-Author(ID, Name). →
select A.id as ID, A.name as Name from
Author;

Edges(ID1, ID2):- AuthorPub(ID1, PubID),
AuthorPub(ID2, PubID).

Query 1 (edges _s → virtual nodes)
select distinct A.aid as X1,A.pid as pubID
from AuthorPub A
Query 2 (edges virtual nodes → _t)
select distinct B.aid as X2,B.pid as pubID
from AuthorPub B;
```

(a) DBLP **[Q1]**

```
Nodes(ID,name):- name(ID,name).
 → select distinct A.id as ID,A.name as name from
name A;

Edges(ID1,ID2):-
cast_info(_,ID1,movie_id),cast_info(_,ID2,movie_id).

Query 1 (edges _s → virtual nodes)
select distinct A.person_id as ID1,A.movie_id as
movie_id from cast_info A;

Query 2 (edges virtual nodes → _t)
select distinct B.person_id as ID2,B.movie_id as
movie_id from cast_info B;
```

(b) IMDB

```
Nodes(ID, name) :- Customer(ID, name).
 → select distinct A.id as ID,A.name as name from name A;

Edges(ID1, ID2) :- Orders(orderId1, ID1),Lineitem(orderId1,part),Orders(orderId2,
ID2),Lineitem(orderId2,part)

Query 1 (edges _s → virtual nodes, assign join to the database)
select distinct A.custkey as ID1,B.partkey as part from Orders A,Lineitem B where
A.orderkey=B.orderkey;

Query 2 (edges virtual nodes → _t, assign join to the database)
select distinct C.custkey as ID2,D.partkey as part from Orders C,Lineitem D where
C.orderkey=D.orderkey ;
```

(c) TPCH **[Q3]**

Figure 3.16: The SQL generated from the system for a few of the graphs we used in our experiments.

underlying relations would require updating the graph. Our system allows making these choices easily and on a per-algorithm basis.

3.6 Summary

In this chapter, 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 deduplication 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 deduplication algorithms that we have developed are of independent interest, since they result in a compressed representation of the extracted graph.

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 3.3: Comparing the performance (running times in seconds, and memory consumption in GB) of C-DUP, BITMAP, and EXP on large datasets; the table also shows the time required for bitmap de-duplication (DNF $\rightarrow$ *did not finish* in reasonable time).

Data Set	Repr	Degree		ConComp		PageRank	
		time	*mem*	*time*	*mem*	*time*	*mem*
S1	EXP	61	237	90	202	**245**	526
	DEDUP1	54	227	**81**	171	311	484
	BMP	**50**	**134**	82	**72**	256	**156**
S2	EXP	**294**	2,879	498	2,869	3,287	9,164
	DEDUP1	335	2,582	460	2,573	3,049	8,126
	BMP	311	**186**	**335**	**163**	**812**	**293**
N1	EXP	142	1,109	241	1,088	1,456	3,389
	DEDUP1	141	926	483	901	1,317	2,874
	BMP	**131**	**219**	**149**	**150**	**469**	**377**
N2	EXP	268	2,710	593	2,690	4,493	8,432
	DEDUP1	312	2,216	495	2,194	3,726	6,892
	BMP	**257**	**479**	**280**	**347**	**824**	**691**
IMDB	EXP	**78**	586	**193**	749	861	1178
	DEDUP1	85	**553**	194	**594**	**802**	**764**
	BMP	146	952	291	1038	807	1185

Table 3.4: Experiments on Giraph showing the running *time(s) / memory(MB)* for different representations and algorithms.

Dataset	Repr	All Nodes	Virt Nodes	Edges
S1	EXP	50,000	0	19,921,854
S1	DEDUP1	50,100	100	14,959,692
S1	BMP	50,100	100	96,066
S2	EXP	50,000	0	373,092,320
S2	DEDUP1	50,100	100	334,148,178
S2	BMP	50,100	100	463,692
N1	EXP	80,000	0	138,689,052
N1	DEDUP1	84,000	4000	114,007,180
N1	BMP	84,000	4000	1,585,536
N2	EXP	140,000	0	346,369,202
N2	DEDUP1	150,000	10,000	281,084,734
N2	BMP	150,000	10,000	3,972,972
IMDB	EXP	503,483	0	33,066,098
IMDB	BMP	925,846	422,363	6,824,494
IMDB	DEDUP1	620,222	116,739	18,088,768

Table 3.5: Descriptions of the datasets used for experiments with Giraph.

Dataset	Nodes	Edges	Join Selectivities
Layered_1	1,299,990	3,999,884	$0.05 \rightarrow 0.1 \rightarrow 0.05$
Layered_2	1,498,692	3,999,908	$0.2 \rightarrow 0.1 \rightarrow 0.2$
Single_1	1,245,532	2,000,000	0.25
Single_2	10,010,000	20,000,000	0.01
S1	50,100	96,066	0.002
S2	50,100	463,692	0.0004
N1	84,000	1,585,536	0.00025
N2	150,000	3,972,972	0.0001

Table 3.6: Selectivities of synthetically generated multi-layer and single layer datasets. The nodes and edges sizes shown here are of the C-DUP representation of these graphs.

Chapter 4: Analyzing Collections of Graphs in RDBMSs

In this chapter we showcase how GraphGen can enable users to express and efficiently extract well-defined *collections of graphs*. We discuss the interesting scenarios where graph collections are useful, and focus on the multi-query optimization problem that comes up when attempting to efficiently extract a graph collection from an RDBMS. We study how GraphGenDL's graph collection features in combination with a *query rewriting* technique can enable efficient extraction of collections of graphs from RDBMSs.

4.1 Graph Collections

Leveraging datasets comprised of *collections* of graphs is an increasingly common trend, seen in various complex analyses. Some examples include: jointly analyzing different types of relationships in social networks [57, 58], analyzing evolving networks over time [56, 59, 60], or conducting data mining and machine learning tasks over graphs [61–63]. "Routing tables" [1] over a set of graphs have also been used in graph databases to guide certain queries to the "most appropriate" graph. The common thread in many of these works is the need for *independent access* to distinct graphs within a set.

Another interesting scenario is *what-if* analysis– the process of changing a value(s) in a dataset to see how the outcome is affected as a result. In the context of graphs, this examines the *different forms* a graph could take *if* a specific change (within a range) is made to the graph. Any simple change, e.g., a certain vertex being removed, could result in a new graph with entirely different characteristics. A set of such changes based on some parameter would therefore yield a *collection* of graphs.

There has been much work on efficiently storing and analyzing a set of *graph snapshots* [64–67]. Unfortunately, it is rarely the case that graphs in a collection of interest are *explicitly* stored as such inside a database, especially if the original data is structured differently in the form of a relational schema. Therefore, gaining access to these graphs often requires a daunting ETL process, as well as a lot of back-and-forth between the user and the database. Given a base graph G, there are two ways to currently delineate *distinct* graphs within G: (a) load all database tables that contain the data required to extract the graphs, and write scripts to manually separate each tuple into its appropriate graph, or (b) conduct a set of queries to the database, one for every graph in the collection. Option (a) *re-invents the wheel*, by creating an in-memory query processing engine from scratch to connect tuples together, sort and aggregate them into nodes and edges. Option (b) requires a large number of queries to the database, expensive data transfers through the network and much *repeated work*.

The example below illustrates the specification of a set of *ego-graphs*, one for every author from a co-authorship graph. The co-author graph in the example below

(in Listing 4.1) is an example of what we have been calling a *hidden* graph since an expensive join is required in order to compute it. Note that an ego-graph in this case consists of the author and all of their direct neighbors.

```
CREATE GRAPHVIEW COLLECTION ego-graphs
WHERE X IN RANGE (Author(C))
  Nodes(C,name):- Author(C,name), C = X.
  Nodes(ID,name):- Authorpub(C,p), Authorpub(ID,p),
     Author(ID,name).
  Edges(ID1,ID2):- AuthorPub(ID1,p),Authorpub(ID2,p).
```

Listing 4.1: Extracting a set of ego-graphs over a graph of co-authors. Note that SQL can be used for any Nodes/Edges statement instead of our Datalog syntax. Please see Section 2.2.2 for more on the syntax and language structures for graph collections

In this chapter, we extended the GraphGen language (GraphGenDL) with simple constructs for specifying a variety of graph collections like the aforementioned example (see Section 2.2.2 for details on the syntax). Next, we discuss how these constructs can unlock the ability to conduct *what-if analysis* over hidden graphs without requiring manual ETL.

4.2 What-if Analysis

As previously mentioned *what-if* analysis, in the context of graphs, is the process of *changing* a graph structure through *additions* and/or *deletions* of nodes/edges in the graph. Each *separate* graph that is output after making a change (or set of changes) in the original base graph, contributes to the collection of graphs. Our language enables users to do this with `MINUS` or `PLUS` keywords.

The user can write: "W $S, f(C, X)$ `WHERE X IN RANGE (`r`)`", where $W = \{$`MINUS,ADD`$\}$, and $S \in \{$`Nodes(), Edges()`$\}$ is a nodes/edges schema. These con-

```
[Q1]
CREATE GRAPHVIEW COLLECTION co-author-peers
WHERE X IN RANGE (1989, 1991, 1)
  Nodes(ID, name, C) :- Author(ID, name, C), year(C)=X OR
      year(C)=X-1 OR year(C)=X+1.
  Edges(ID1, ID2) :- AuthorPub(ID1,pub), AuthorPub(ID2,
     pub).

[Q2]
CREATE GRAPHVIEW COLLECTION customers-order-snapshots
WHERE orderDate IN RANGE (Lineitem(_,_,_,orderDate))
  Nodes(ID,name) :- Customer(ID, name).
  Nodes(ID,price) :- Part(ID,_,_,_,_,_,_,_,price).
  Edges(ID1,ID2,C) :- Lineitem(oId,ID2), Orders(oId,ID1,C
     ), C=orderDate.

[Q3]
CREATE GRAPHVIEW COLLECTION ego-graph-two-hop
WHERE X IN RANGE (Author(C))
  CoAuthors(A,B):- AuthorPub(A,p),Authorpub(B,p).
  Nodes(C):- Author(C), C=X.
  Nodes(ID):- CoAuthors(C,ID), Author(ID), C=X.
  Nodes(ID):- CoAuthors(C,a), CoAuthors(a,ID), Author(ID)
     , C=X.
  Edges(ID1,ID2) :- CoAuthors(ID1,ID2);
```

Listing 4.2: `Q1` queries a co-author graph for every year X, which contains only authors that were born a within a year of X, `Q2` specifies one bipartite graph of customers, connected to parts that they bought, for each date that customers ordered parts. `Q3` Extracts the full set of two-hop ego-graphs over the co-authors graph (direct neighbors, and their neighbors)

structs specify a set of changes based on a range of values and therefore output a graph for each such change. The tagging predicate f associates the change with certain nodes/edges.

The resulting graphs from `MINUS` queries represent the state of a base graph *if* a set of nodes/edges were removed. The result of the query shown below contains graphs x_i that represent the state of a network of servers *if* node with `ID` $= x_i$

were to be *removed*. Having access to these graphs independently would allow us to analyze them individually and predict to what degree the system is susceptible to network partition failures.

```
CREATE GRAPHVIEW COLLECTION network-partitions
  Nodes(ID, p) :- Servers(ID,p).
  Edges(ID1,ID2) :- MessageSent(ID1,ID2).
MINUS Nodes(C), C = X WHERE X IN RANGE(Servers(C))
```

Queries that include a `PLUS` statement represent a set of graphs that have a set of *additional* nodes and/or edges. An extra set of Nodes or Edges statements must be specified after a `PLUS` statement. We assume that each graph is *valid* after the addition of the new nodes/edges. If we therefore want to study the collection of graphs that result from adding *new edges* to the base graph, the source *and* destination nodes for each edge are assumed to exist (either inherently in the base graph or added explicitly by the `PLUS` operation). The example below returns a set of friendship graphs, each with an extra set of friendships between users that went to the same school.

```
CREATE GRAPHVIEW COLLECTION social-networking
  Nodes(ID, p) :- Users(ID,p).
  Edges(ID1,ID2) :- Friends(ID1,ID2).
PLUS Edges(_,_,schoolId) WHERE X IN RANGE(Users(_,_,
   schoolId)).
  Edges(ID1,ID2, schoolId) :- Users(ID1,_,schoolId),
     Users(ID2,_,schoolId), schoolId = X.
```

Having access to these graphs allows users to analyze which graph satisfies their connectivity criteria best and choose to recommend friends to the users in a personalized manner.

Please refer back to Chapter 2 where we have discussed details on the syntax

and the way users can interface with the extracted graph collection.

4.3 Extracting Graph Collections

As previously mentioned, the baseline approach for extracting a collection of graphs is to execute a *separate* query for every value x in the range of values r, thus generating *a separate query* for each graph.

The approach we take in GraphGen is to have the user specify *one* base graph G which *contains* all graphs within collection, and use a single SQL query to extract it. Given this approach, we tackle the following question: given G that *contains* a collection of graphs of interest, *how can we group the nodes/edges in* G *into their appropriate, distinct graphs*? We instead use a set of query *rewrite rules* which add *tags* (one tag for every value in the range r) to the nodes and edges in G, dividing them into their respective graphs in the collection specified. If e.g., a vertex is tagged with tags {`1, 2 ,3`}, that signifies this particular vertex exists in all three graphs `1,2,3` (each graph is identified by its tag value). Figure 4.1 shows the interactions with the database that are required for this process. The Nodes and Edges statements are first parsed (and translated to SQL if necessary). Next, we create a *logical query plan tree* for each statement. The query rewriting process changes the query plan tree accordingly, and outputs a new SQL query.

It's important to note that our tagging approach assumes the underlying database has some sort of `array_agg` implementation, and supports one of the SQL flavors that GraphGen can generate queries in.

4.4 Tagging Framework

We now formally describe the aforementioned rewrite rules. In terms of the *relational algebra* symbols used below, π is projection, ρ is rename, $\bowtie_\theta$ is theta-join, $⟗$ is full-outer join, and γ is aggregation.

Let $S_N = \{N_1, N_2, ..., N_n\}$ and $S_E = \{E_1, E_2, ..., E_n\}$ be the set of nodes and edges statements respectively specified by the user. Let $R_{nodes} = N_1 ⟗ N_2 ⟗ ... ⟗ N_n$, and $R_{edges} = E_1 ⟗ E_2 ⟗ ... ⟗ E_n$. Also, let $f(c, x)$ be the *tagging predicate*, where c references the *value* of a column C in the result of any tuple $s \in S_N \cup S_E$ that contains column C, and $x \in r$, where r is the user specified value range. Let $R_S, R_{S'} \in \{R_{edges}, R_{nodes}\}$ and $R_{S'} \neq R_S$. We denote the `ID` attribute(s) in each case as $i \in \{$`ID, (ID1,ID2)`$\}$.

4.4.1 Rule 1 (Tagging)

Given a *logical query plan tree* of R_S, let R_C denote the relation that contains the attribute C. This is the relation that we are going to initially *add tags* to, using f. We first generate a *temporary table* T, that simply includes the *list of values* in range r, as a single column $T.X$. We then traverse the logical query plan tree and apply the transformation:

$R_C \rightarrow (R_C \bowtie_{f(\texttt{R.C,T.X})} T)$

We essentially *join* the relation that includes C with T, using f as the join condition (this is a cross product filtered by predicate f). This sets a tag at every joining tuple in R_C. Afterwards, we recursively return up to the root of the tree

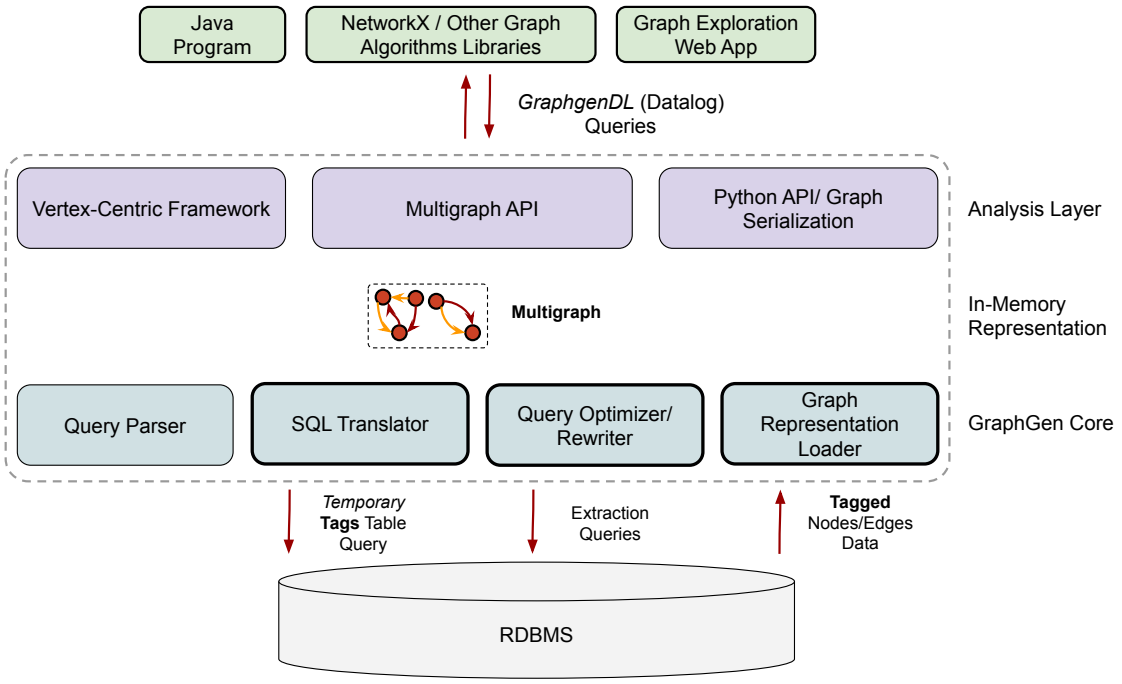


Figure 4.1: The query is parsed, rewritten by altering the logical query plan, and then executed against the database, aiming to push as much of the computation required for the extraction, to the database.

to find the appropriate projection node, and we apply the following rule: $\pi_{i,p} \rightarrow \gamma_{i,\texttt{array\_agg}(X)}(\pi_{i,p,\texttt{T.X}})$

In other words, we include the tag in the projection, and aggregate all tags into a single array associated with each element that was tagged. An example of the result of R_{nodes} after applying this rule can be seen in Figure 4.2a, where each node contains a list of tags that indicate which graphs in the collection that node is a part of. In this particular example, we have a graph for each year, so because author with `ID=1` was born in 1991, it is part of graphs for years $\{1991, 1990\}$

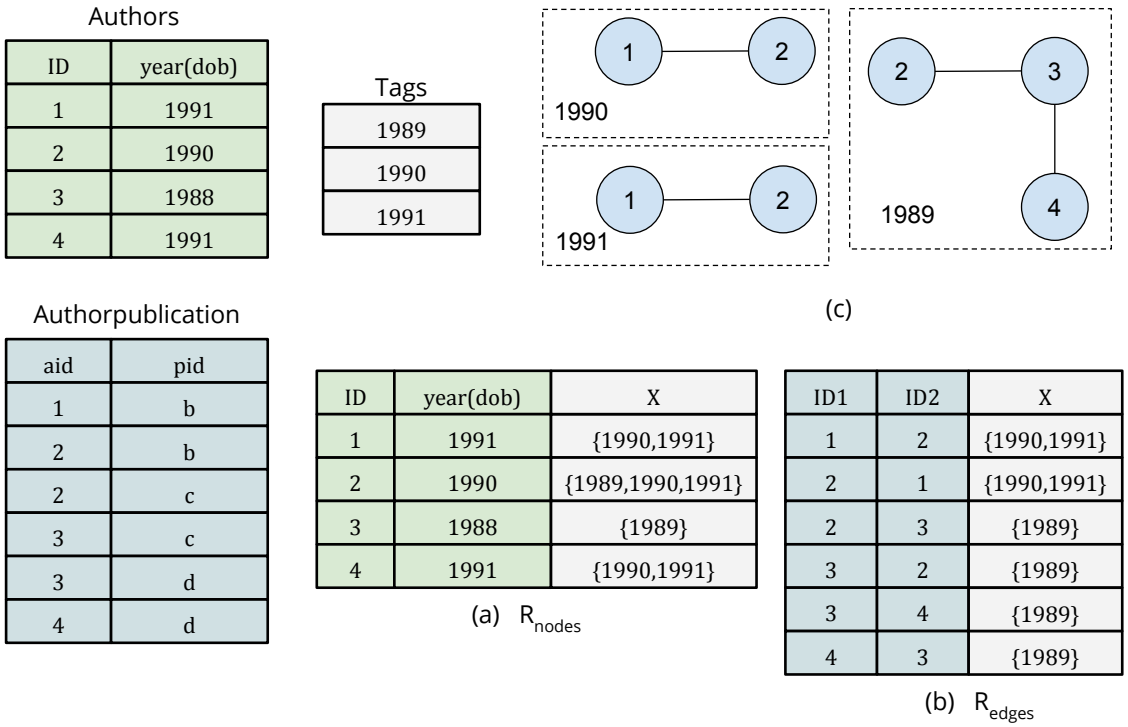


Figure 4.2: Data sample for query `Q1` in Listing 4.2 (a) The state of R_{nodes} after Rule 1 has been applied. (b) The state of R_{edges} after rewrite rule 2 has been applied. (c) Visual representation of each graph in the collection.

4.4.2 Rule 2 (Tag Propagation)

After executing Rule 1, if only one of R_{nodes}, R_{edges} is tagged, we need to appropriately *propagate* that tags to the rest of the elements. As we discuss in detail later, we *combine* tags in one of two ways $\kappa \in \{\cup, \cap\}$ depending on whether they are being propagated from nodes to edges, or vice versa. Tags are combined in the end of the query in order to maintain correctness when propagating them from one element to the next. Vertices need to exist in the *union* of graphs that all of their incident edges exist in, where as edges should *only* exist in the *intersection* of graphs their incident vertices exist (since an edge cannot exist without *both* its

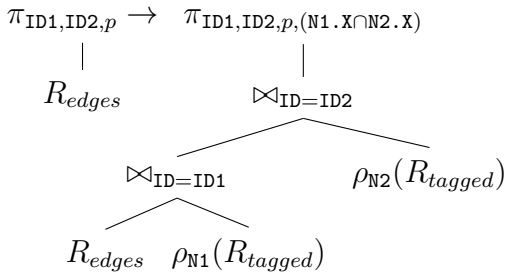
source and destination nodes).

If R_{nodes} is tagged and R_{edges} is not, for every edge $e_i = (n_s, n_d) \in R_{edges}$ we will have $tags(e_i) = tags(n_s) \cap tags(n_d)$. The intuition behind this is that an edge only exists in the graphs that *both* the source and destination nodes are a part of. When both R_{nodes} and R_{edges} are tagged, no tag propagation is necessary. An example of the result of R_{edges} after applying Rule 2 can be seen in Figure 4.2b.

More generally, we rewrite the query as follows:

$$\pi_{i,p}(R_S) \rightarrow \pi_{i,p,(\texttt{F1.X}\ \kappa\ \texttt{F2.X})}(\rho_{\texttt{F1}}(R_{S'}) \bowtie_{\texttt{ID1=ID}} R_S \bowtie_{\texttt{ID=ID2}} \rho_{\texttt{F2}}(R_{S'})) \tag{4.1}$$

Let R_{tagged} represent the tagged relation. Using Equation 4.1, if $R_{tagged} = R_{nodes}$, we rewrite:



The rewrite works exactly the same way if $R_{tagged} = R_{edges}$ except $\kappa = \cup$ in Equation 4.1. Intuitively, this is because nodes will exist in *all* graphs that their incident edges are a part of.

Tagging `MINUS` Queries: In the case of `MINUS` queries, the above rewriting rules would work the exact same way. A key difference here however is the *meaning* of these tags changes– the tags would now mean that the element exists in all graphs *except* the graphs with ids listed in the tag list.

Tagging `PLUS` Queries: On the other hand, `PLUS` queries work the same way as regular tagging queries (e.g.,`[Q1, Q2, Q3]`). Due to the assumption that all

required vertices are already part of the base graph, adding new sets of edges for every value in the range simply tags them with the id they exist in. All nodes/edges in the base graph definition are assumed to exist *in all graphs*, and therefore are logically tagged with all of the versions beforehand. The final edges relation will be: $R_{edges} = R_{tagged} \cup R_{edges}$. Note that `PLUS` queries are always going to add new edges since adding vertices (without any edges) is not interesting.

Query	# Tags	Single Query	Query-at-a-time	MQO
Q1	101	434	3,967	864
Q1'	101	523	9,444	6,526
Q2	126	2,898	134,832	13,825
Q3	2,577	1,171	345,103	256,445

Table 4.1: Query times are in *ms*. MQO refers to *Multi-Query Optimization* as it aims to mimic the approaches in past work which look for common sub-queries across queries, materialize those sub-queries and re-use them.

4.5 Preliminary Experiments

We ran four different types of graph collection queries over small subsets of the TPCH [32] and DBLP [31] datasets on a commodity laptop. For these preliminary experiments, the query rewrite rules were applied manually to each of those queries.

We compare our approach (which generates a single query for the nodes and a single query for the edges), with the baseline *query-at-a-time* approach of posing a query for every distinct value in the *range* specified in the graph extraction task. We also attempted to *simulate* the multi-query optimization (MQO) approach proposed in recent work [68] which looks for common sub-expressions shared across a set of queries, and materializes and re-use them. We did this by manually materializing

portions of the query that are not parameterized, and referencing them in every generated query. In particular, the query-at-a-time approach the query in Listing 4.1, would have to compute the `Edges` table for every single graph. However since that join does not contain any parameterized filter conditions, we materialized it once and re-used it at every generated query.

In Table 4.1, `Q1'` is the `MINUS` version of `Q1` and extracts graphs x_i, each of which contains the full co-author graph, *except* all authors that were born within a year of x_i. `Q2` queries snapshots over a bipartite graph of customers and parts they have ordered, and `Q3` queries two-hop ego-graphs over the co-author graph. All queries in the experiments are shown in Listing 4.2. We found that both the *number* and *size* of the graphs in the collection affect performance outcomes. Our technique is an order of magnitude faster in the `MINUS` query (`Q1'`) because it ended up querying substantially less data. The graphs in set `Q1'` are significantly *larger* than those in `Q1`. Tagging is over *100x* faster for `Q3`, which queries 2577 graphs. While MQO can provide benefits, it only does so when there is a portion of the query that can be shared. Even for those queries however (e.g., `Q3`), the benefits of MQO start becoming irrelevant when the *number* of graphs is large.

4.6 Summary

In this chapter we presented our preliminary work on enabling analysis of collections of graphs over RDBMSs. We proposed novel declarative language structures to GraphGenDL that allow users to specify a set of independently accessible

graphs. The intuition behind these language structures is that they enable users to specify a single base graph in normal GraphGenDL fashion, and then specify a parameterization which maps each element of that base graph into their appropriate, distinct, graph within a well-defined collection, based on a range of values and a mapping predicate function the query is parameterized over. We proposed a set of simple query rewriting rules which tag elements of the base graph appropriately, distributing them to the graph(s) they should be a part of. Our tagging query rewrite rules which output a single SQL query show promise as they significantly outperformed naively executing a separate query for each distinct graph, and also outperformed past multi-query optimization techniques by over an order of magnitude in a preliminary evaluation.

Chapter 5: Leveraging Graphs for Aggregate Query Processing

In this chapter we discuss how we can use graph representations of RDBMS data in order to execute multi-way join-aggregation queries, in a memory-efficient fashion. This work was inspired from the observation that the condensed representations presented in Chapter 3, essentially represent the join result in a latent representation, which can be used to enumerate the join result by simply traversing it. Another observation we made (as mentioned in Section 1.3.2) was that, while large-output joins result in an explosion in memory requirements, the results of aggregate queries can be orders of magnitude *smaller* than the join result. Many mission-critical, decision-support queries in standard BI analytics fit this description. We therefore asked the question: Could we devise a general algorithm that is able to compute *aggregations* over arbitrary *acyclic* joins, where the result is grouped by an arbitrary number of attributes from *any* combination of the relations involved in the join? This would enable the execution of join-aggregation queries by essentially only using memory bounded by the input RDBMS relations.

5.1 Overview

This section provides the reader with a more detailed overview of the problem we tackle, our techniques, and the contributions in this chapter.

5.1.1 Re-thinking Aggregate Query Processing

Traditionally, aggregate processing over conjunctive queries in RDBMSs has been done through the use of simple binary operators for executing joins, followed by a (typically separate) unary aggregation operator. The simplicity of these operators has proven invaluable throughout the development of modern RDBMSs. Each simple operator enables the optimization of a very specific operation with a concise set of parameters, inputs and outputs. This enabled simpler query optimization, since it is easier to create good cost models for simple operators than for complex ones.

Simplicity however often comes at the cost of performance. It is known that binary operators can lead to sub-optimal performance regardless of the query plan used [69,70]. The main drawback with binary join operators in RDBMSs specifically, is the generation of intermediate join results, potentially with materialization at the granularity of every join. Each individual join within a query plan may output an increasingly larger number of tuples, making the latter intermediate results unwieldy, especially as they start becoming larger than memory. Pipelining operators help in some cases, but enumeration of the full intermediate result set though joining every single tuple is necessary and cannot be easily avoided (e.g., bloom filters can help in

some cases to filter out results, but break the classical model as well). These issues become significantly more pronounced in analytics settings where the joins are often done on non-key attributes to derive higher-level insights (see examples below).

This has led to an increasing interest in the idea of *multi-way* database operators. Eddies was one example of such an operator [69], where the benefits of combining multiple operators into one came from the ability to choose different execution paths for different tuples. More recently, breakthroughs *in worst-case optimal* join algorithms [71] have shown that one can put tight bounds on the maximum possible number of tuples generated by a query, and then develop algorithms whose runtime guarantees match those worst-case bounds. These breakthroughs have led to a variety of different query operators that take a *multi-way* join approach over the traditional binary operator. The benefits seen by many of these proposed operators typically come from the fact that the operator takes multiple relations that are part of a large conjunctive query into account *simultaneously*. This allows for avoiding the materialization of large intermediate results [72], enables pruning out various portions of the computation based on complex conditions [73], or allows for exploiting more parallelism and fast set intersections toward the join result [74].

In this chapter, we focus on another very common combination of operators, namely a series of joins followed by a group-by aggregate. Our approach focuses on any query in which the join result is large, but the final post-aggregation result is small. To re-iterate Example 1.3.3 at a high level, we can look at the TPCH supply-chain dataset of items, orders, customers and suppliers. If we wanted to figure out the sum of `(order,customer,item)` records we are storing for each supplier

per zipcode in which that supplier satisfied orders, the result size post-aggregation would be bounded by the number of zipcodes times the number of unique suppliers. The size of the *join result* before we apply aggregation however can be very large. Another example is computing the aggregation of a set of paths between all pairs of nodes in a graph since the number of paths could be huge depending on the characteristics of the graph, whereas there is a limited number of node pairs. Example 1.3.3 and Example 1.3.4 showcase these cases in more detail.

At a high level, our operator works by loading in a compact representation of the underlying data in the form of a graph that we call a "data graph". This graph can then be traversed to yield the final aggregation result thus avoiding the materialization of intermediate join results.

5.1.2 The Join-Agg Operator

In this section we formally describe the general framework for efficiently answering queries like the one shown in Listing 5.1. Our framework views the join between a series of relations in the form of a *graph* structure of interconnected tuples that we call the *data graph*. For the sake of simplicity, we use the `COUNT(*)` aggregation function in our explanations and examples. We provide a discussion on how more standard aggregation functions can be supported using the same framework in Section 5.3.4.

Let $Q(R, G)$ be an aggregate query over a join between a set of relations R, where $G = \{g_1, g_2, ..., g_n\}$ is the set of group-by attributes of this query. For now,

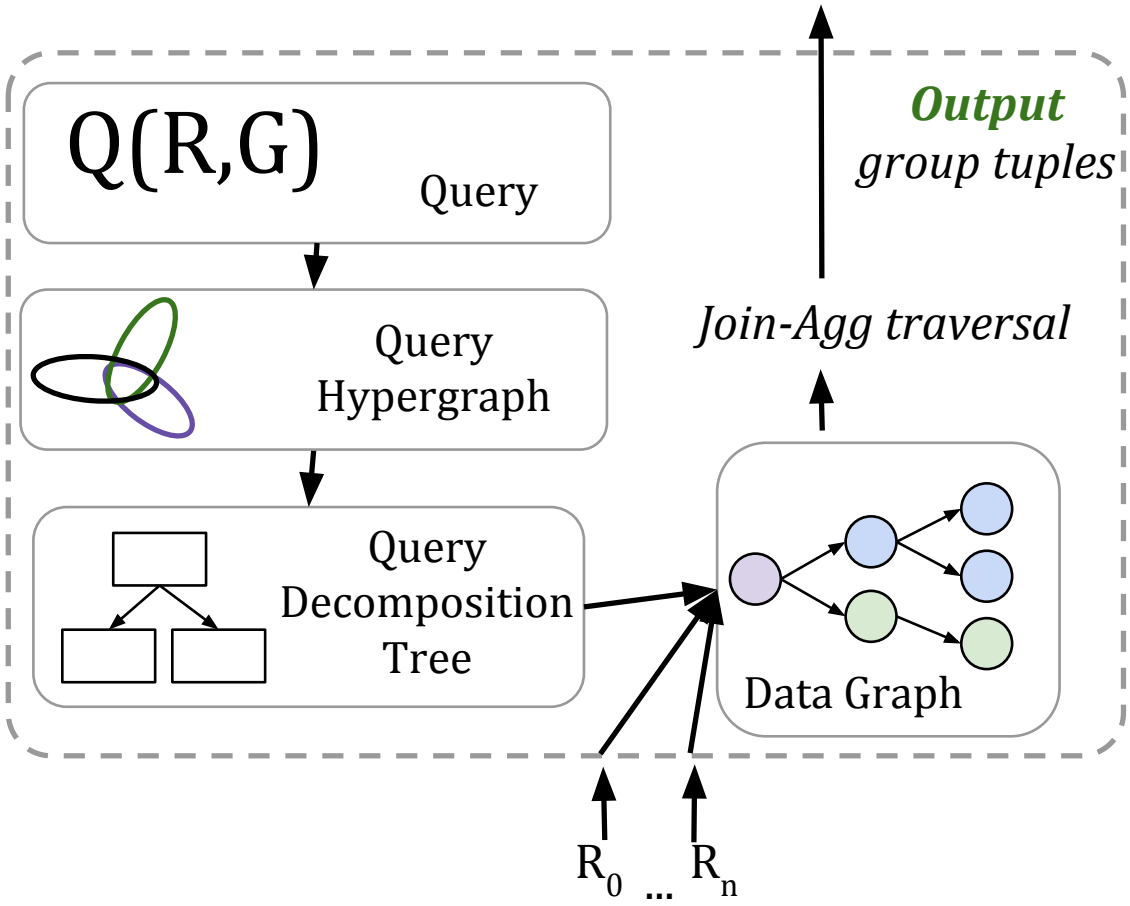


Figure 5.1: The inner workings of the JOIN-AGG operator.

```
SELECT A.a,B.b,C.c, COUNT(*)
FROM R1 A, R2 J,R3 B,R4 C
WHERE A.j1=J.j1 AND J.j2=B.j2 AND J.j3=C.j3
GROUP BY A.a,B.b,C.c;
```

Listing 5.1: Generic group-by query with *three* group-by attributes.

assume that we only need to count the number of tuples in each group (`COUNT(*)`). We assume without loss of generality that a group by attribute $g_i \in G$ corresponds to a single attribute in relation R_i, and that *none* of the g_i participate in a join condition. We also assume that all joins are natural joins. All of these can be relaxed easily through standard tuple-level transformations (e.g., if a group-by attribute participates in a join, we can (implicitly) create a copy of that column). As

mentioned earlier, we restrict our attention to **acyclic joins** in this dissertation.

We represent the overall join-aggregation query as a hyper graph $H(X \cup G, E_H)$ where X is the set of all attributes that take part in the *join conditions* between the relations in R and E_H represents hyperedges, containing one hyperedge e_{R_i} per relation R_i, i.e., $e_{R_i} = R_i \cap (X \cup G)$. Note that the only attributes that are relevant here are either join condition attributes, or group attributes– the result is a set of tuples that represent groups (grouped by G). Let $R_i.x$ denote attribute (or set of attributes) x from relation R_i.

For every $e_{R_i} \in E_H$, we partition the attributes of e_{R_i} into two disjoint groups (x_l, x_r). We describe the specifics in Section 5.2.2, but intuitively this is done in order to reduce the size of the *data graph* that we load into memory, while also capturing enough information to execute the query.

We propose a new database operator called Join-Agg that receives a set of input relations R and outputs a single set of result tuples, i.e., after the appropriate grouping and aggregation, as the output. The decision of whether to use the operator is made by the query optimizer in a cost-based manner; in essence, if at least one of the joins in the query is a *non-key* join or a join that may result in a large output compared to the input relations, then this new opeartor should be considered. When the operator is chosen, instead of conducting a series of binary joins as traditional RDBMSs do, we would instead go through each relation, and load each one into an in-memory *data graph* which is then traversed to output the resulting grouped tuples and their aggregate value.

Prior to the instantiation phase, the operator creates a **query hypergraph**

H that captures the joins in $Q(R, G)$. This query hypergraph is then turned into a **query decomposition tree**, which is traversed in order to transform each individual relation into a set of *edges* in the data graph. Based on the final decomposition, during the execution phase, the operator constructs the edges that correspond to each relation as the **data graph**. Finally, this in-memory data graph structure is used (and potentially re-used) to directly compute and output the grouped tuples.

The data graph paradigm proposed here is reminiscent of *factorized representation of conjunctive query results*, by Olteanu et al. [75], and the idea of a *tuple hypergraph* that can cover all tuples in a query result [76]. All of these provide compact representations of the underlying join result, especially in presence of large-output joins, with minor differences because of the specific goals behind their genesis. Our key contribution here is a novel way to use such a structure for computing *group by aggregates* efficiently over complex acyclic joins, by using an alternative approach to computing aggregates over a "factorized" representation. More information regarding the juxtaposition between our system with the related work can be found in Chapter 6.

5.1.3 Summary of Contributions

Our main contributions presented in this chapter are twofold; *first*, we propose a new *multi-way* database operator called Join-Agg, which enables the efficient computation of aggregation queries, *without materializing* any intermediate join results, by computing the *join* and *aggregation* simultaneously. We describe

a novel *general framework* for executing aggregation over conjunctive queries of arbitrary numbers of relations, and numbers of group by attributes that may be derived from any participating relation, by leveraging a *graph* representation of the underlying *data*. We restrict our formal development to **acyclic** queries – although our algorithms can be adapted to handle cyclic queries, systematically combining our data-level optimizations with the recent work on cyclic joins raises complex issues that are beyond the scope of this dissertation. We implement a prototype of the Join-Agg operator *outside* of the RDBMS and experimentally showcase the benefits of our operator over synthetic and real datasets.

Second, we provide a comprehensive complexity analysis of common example queries that benefit from our Join-Agg operator in comparison to executing them using the classical RDBMS model, or other less general techniques such as pre-aggregation [35] which only looks at reducing intermediate data size at the level of each individual join instead of looking at the join as a whole. We show that in terms of computational complexity Join-Agg is comparable or asymptotically better than those techniques, particularly in the general case of complex acyclic branching join queries. We also show that Join-Agg is overall *better* than those techniques in terms of memory complexity.

Section 5.2 describes the construction of the data graph representation, while Section 5.3 goes in-depth regarding the details of our algorithm which *traverses* that representation and finally yields the query results. Section 5.4 provides an in-depth complexity analysis of our technique, we expand on the details of our implementation in Section 5.5 and provide an extensive experimental evaluation in Section 5.6.

5.2 Data Graph Representation and Construction

In this section, we begin with describing how a **query decomposition tree** is constructed and how it is used to split the attributes of each relation into two groups that form the edges of the data graph. We then describe the basic representation of a **data graph** and explain how it is constructed by loading in relations from the underlying database.

5.2.1 Query Decomposition

A query decomposition of a hypergraph H is defined as a *tree* where each node corresponds to a hyperedge $e_H \in H$. We create a *pure* query decomposition of H where each node in the decomposition directly corresponds to a single relation. In this work, we focus on acyclic queries, i.e., queries for which there exists a *tree* decomposition [77]; in future work, we are planning to extend our approach to handle cyclic queries by combining it with recently proposed techniques for optimal worst-case join algorithms.

We construct the query decomposition tree using the standard *elimination* algorithm [78]. First, we note that, all of the relations that contain *at least one* attribute not present in any other relation must contain a group attribute; we'll call these **group relations**. We start with one of those as the root of the tree, and traverse the hypergraph in a breadth-first manner to construct a query decomposition tree. An example of such a decomposition can be seen in Figure 5.3d. Given H, to build the query decomposition tree, we can start from any group relation; here we

picked A. The hypergraph is then *traversed* in a breadth-first manner starting from A using a standard queue. We start by creating a root node in the decomposition tree for A – let that be the *current* node. Then, for every neighbor relation of A, if it has not been visited, we add it to the queue. We then pop the queue and add the popped relation as a *child* to the current node in the decomposition tree. Thereafter we continue with neighbor B which becomes the current node, it is added as a child node to A and so on until the queue is empty.

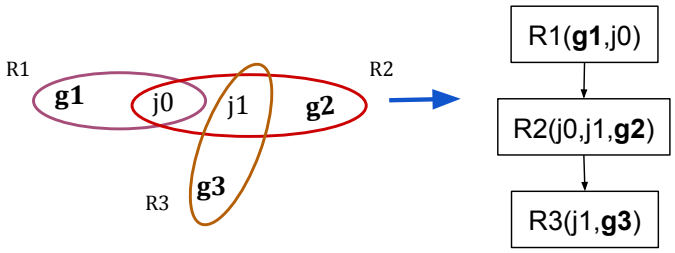
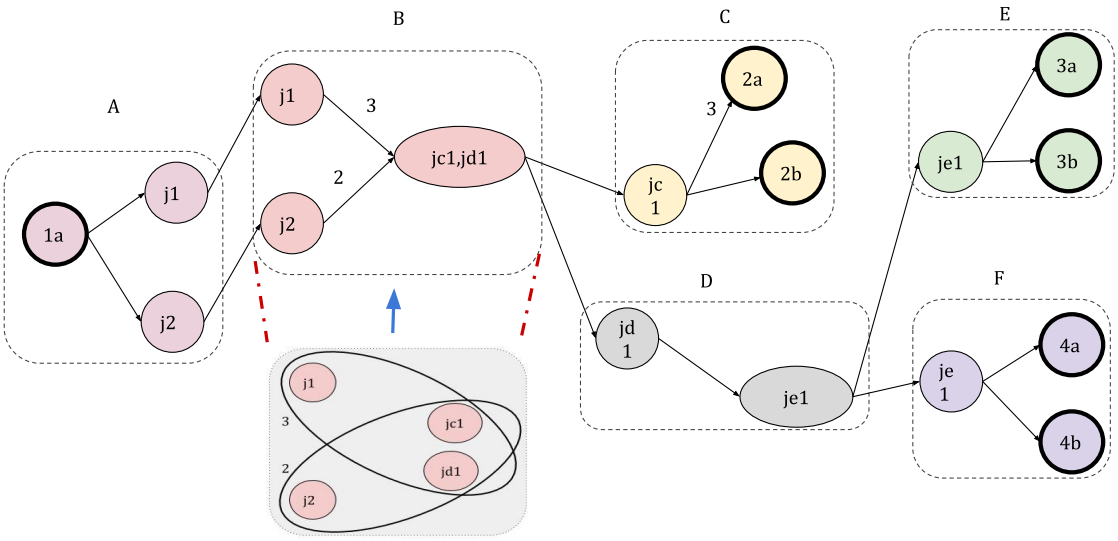


Figure 5.2: Derivation of a Query Decomposition tree from a Query Hypergraph.

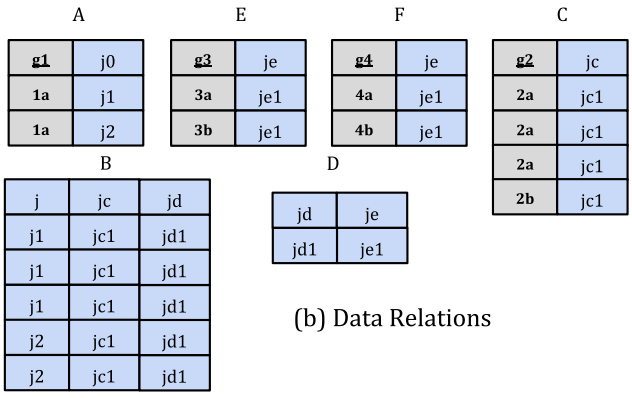
5.2.2 Splitting Attributes

As previously mentioned, the attributes of each relation are partitioned into a pair of attribute sets (x_l, x_r). This is done in order to properly view every relation as a set of *edges* for the data graph–an edge has two entities it connects, here it connects x_l and x_r. To do this, we simply traverse the query decomposition tree starting from the root. As we discuss in further detail later on, this splitting mechanism is a *data reduction* mechanism (similar to *pre-aggregation* [35]) for reducing the input data as much as possible before the query is executed.

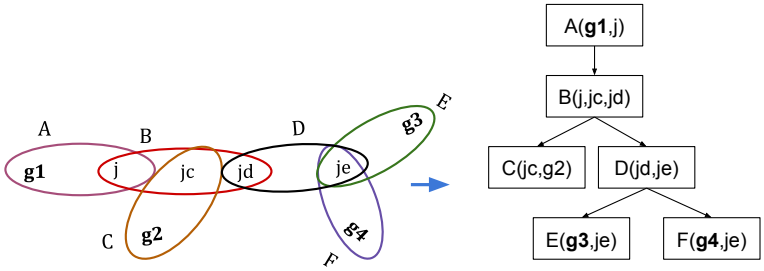
To construct these pairs, we traverse the decomposition tree in order to par-



(a) Data Graph



(b) Data Relations



(d) Query Hypergraph (c) Query Decomposition Tree

Figure 5.3: A data graph created by a set of joining relations (after projections have been applied). Relation B has multiple attributes as part of x_r, which merge into the multi-node `(jc1,jd1)`. In the relations involved in the join, we have four different group attributes g_i, one of which is a source attribute (g_1). Node `1a` is a *source* node, `2a, 2b, 3a, 3b, 4a, 4b` are all group nodes, and `(jc1,jd1)` and `je1` are both *branching* nodes. The rest are all *intermediate* nodes.

tition the set of attributes for each relation into a (x_l, x_r) pair. In a slight abuse of notation, let R_i also represent the *set of attributes* relevant to the query from relation R_i. We start the traversal at the root; let that be $R_S(g_0, a_1, ..., a_n)$, for which we set $R_S.x_l = \{g_0\}$. Afterwards, we set $R_S.x_r = \bigcap C_S$ where $C_S = R_S.children() \cup \{R_S\}$ is the *set of attributes* in the children relations of R_S in the decomposition tree, plus (union) R_S itself. Next, for every relation $R_i \in R_S.children()$, if R_i *is not* a group relation, we set $R_i.x_l = parent(R_i).x_r$, and again $R_i.x_r = \bigcap C_i$. If R_i *is* a group relation, we set $R_i.x_l = R_i \setminus \{g_i\}$ and $R_i.x_r = \{g_i\}$. As we will later describe, nodes created from group attributes need to be *sinks* in the DAG structure that we will be building. This same process is recursively executed on all relations in $R_i.children()$.

Below we provide a few standard examples of how the aforementioned algorithm is used to split the attribute sets.

Example 5.2.1. Looking at the decomposition tree in Figure 5.3c, a simple and common case is that of `D`. Because `D`'s parent relation is `B`, we have $x_l = B \cap D = \{jd\}$. Also, the children of `D` are `E,F`, therefore $x_r = D \cap E \cap F = \{je\}$. In the data graph constructed later, a single node will be created for each value in $\pi_{jc}(D)$ and for each $\pi_{je}(D)$.

Example 5.2.2. Again regarding the tree in Figure 5.3c, for relation `B`, its x_l value will be $B \cap A = \{j\}$, and $x_r = B \cap C \cap D = \{jc, jd\}$. This multi-valued x_r value means that a *multi-node* will be created for each value in $\pi_{jc,jd}(D)$.

Example 5.2.3. Looking at the decomposition tree in Figure 5.2, for relation $R2$, since it's a *group* relation (one that contains at least one group attribute), we *always*

split its attributes by setting the group attribute as $x_r = \{g2\}$,and $x_l = R2 \setminus \{g2\} = \{j0, j1\}$.

5.2.3 Data Graph Representation

Next, we formally define the data graph representation for a join-aggregate query. For a given query Q, a *data graph* $G_Q(V, E)$ captures the underlying data in the relations and the interconnections between those data elements. Let $n \in V$, and $e = (n_l, n_r) \in E$ the nodes and (directed) edges respectively that make up graph G_Q.

Relation & Node Types: At a high level, we partition the relations R in four (overlapping) groups: R_S, R_B, R_J, R_G, which dictates how the corresponding nodes are handled during execution of the JOIN-AGG algorith:

1. $R_G = \{G_0, G_1, ...G_n\}$ denotes the *group* relations in R, each containing a group by attribute g_i.

2. R_S is the *source* relation $R_S \in R_G$, which we choose to start the computation from. The algorithm we develop is based on a *traversal* of the data graph. As we describe in Section 5.3, nodes originating from the source relation are the **anchors of the traversal** and therefore do not get visited multiple times. As a result, no additional data needs to be maintained for them.

3. $R_B = \{B_0, B_1, ..., B_n\}$ a set of *branching* relations in R. A relation is marked as a branching relation if its corresponding node in the *query decomposition tree*: (a) has *more than one* child and therefore branches out the join execution, or

(b) is *not* a leaf node or the root node, **and** *is* also a group relation. In other words, either the tuples in these relations need to sequentially be joined with multiple tables, one at a time in the context of the overall conjunctive query, or there is a grouping attribute in the relation that needs to be separated out so that we can exploit *caching* effects (as discussed later). An example of a type (a) branching relation is D in Figure 5.3d, and a type (b) would be relation $R2$ in Figure 5.2.

4. $R_J = \{J_0, J_1, ..., J_n\}$ a set of intermediate relations in R. These are relations that only have exactly *one child and one parent* in the query decomposition tree, and are *not* group relations.

Consequently, there are four types of nodes in the data graph, each originating from its adjacent relation type: source nodes (n_s), group nodes (n_g), intermediate nodes (n_j), and branching nodes (n_b). Each of the aforementioned relations all portray a pair of attributes (x_l, x_r) that are relevant to the query. As discussed in Section 5.2.5, source nodes are always loaded from the x_l attribute in the *source* relation, while group and branching nodes from the x_r attribute in their respective relations. Group nodes are always *sinks* in the data graph, while we made the choice to always set branching nodes on the right hand side of the split for the sake of simplicity. All remaining nodes in the data graph are intermediate nodes.

Note that a relation can have *multiple types* (e.g., R_S is both a source relation and a group relation). Similarly, based on the specific query, a branching relation can also be a group relation if it satisfies the criteria for both relation types simul-

taneously. This would occur for instance if a relation includes a group attribute g_i, but *also* joins with more than two relations in the query described by a hypergraph H. The *nodes* derived from relations with multiple types naturally also inherit the same set of node types.

Attribute *splitting* (Section 5.2.2) enables us to conduct a *pre-aggragation* step in which we group tuples with the same values (after projection) into a single edge with a *multiplicity* value. For example, in regards to relation C in Figure 5.3, splitting attributes this way not only allows us to pre-aggregate $(2a, jc1)$, but to also only load in a single node for $jc1$.

5.2.4 Mapping Relations to a Data-Graph

We now formally describe how we map rows in the underlying relations to nodes and edges in the data graph. Let $\pi^*_{x_1,x_2,..,x_n}(R_i)$ denote the set of values of attributes $x_1, x_2, .., x_n$ in relation R_i, where π^* indicates bag semantics for the projection. Also, let $\pi_{x_j}(R_i)$ denote the set of unique values of the attribute x_j in relation R_i, and let X_i denote the set of attributes in relation R_i that take part in a join condition. We create the nodes in the data graph in two simple steps; for every relation $R_i \in R$:

1. We create a *hyperedge* for every (x_l, x_r) tuple in relation R_i, as is seen in Figure 5.3 (hyperedges with only 2 nodes are shown as regular directed edges). Every hyperedge describes a set of values from attribute sets $x_l \cup x_r$. A unique value in attribute set x_i corresponds to a single node in such a hyperedge.

2. For every set of nodes that appear in the same x_l or x_r, create a *multi-node* that includes *all* values in the intersection (also shown in Figure 5.3 for relation B). The result is a set of regular directed edges between nodes and/or multi-nodes. For all purposes moving forward, multi-nodes function exactly the same way as regular nodes in the data graph. In general, the node created from an attribute set x_i is simply n_i, and its set of values are denoted as v_i.

We now define the *edges* in G_Q. Let m_e denote the *multiplicity* of an edge $e \in E$. The multiplicity of an edge is a numeric value associated with each edge and is defined as the number of times the tuple that edge corresponds to exists in the relation. G_Q contains a directed edge $(n_l, n_r) \in E$ iff one of the following applies:

1. There exists a tuple $(v_l, v_r) \in \pi_{x_l,x_r}(R_i)$. If $A = \{(a,b) \in \pi^*_{x_l,x_r}(R_i) : a = v_l \wedge b = v_r\}$, the set of tuples in R_i with values (v_l, v_r) in R_i, the multiplicity $m_{(n_l,n_r)} = |A|$.

2. A tuple in R_i joins with one in R_j on attribute $x_{join} = X_i \cap X_j$, such that $v_{join} = v_r = v_l$. In this case, the multiplicity of the edge is always $m_{(n_l,n_r)} = 1$. An example of such an edge is $(n_{A.j1}, n_{B.j1})$ in Figure 5.3a.

For the sake of simplicity, and without loss of generality we can assume that any x_i, corresponds to a single attribute, i.e., relations only join with one another through single attribute join conditions. In practice, x_i can be a set of attributes, in which case v_i (the value for node n_i) would constitute of a bag of values and be described as a multi-node in the data graph. Formally, in that case we simply have that $(n_l, n_r) \in E : x_l \cap x_r \neq \emptyset$ where $x_l = X_i$ and $x_r = X_j$ and $v_l \cap v_r \neq \emptyset$.

5.2.5 Join-Agg Stage 1: Loading Data Graph

To summarize, the input to the overall load process is the hypergraph H. We initially need to partition all sets of relation attributes to (x_l, x_r) pairs, by first creating a *query decomposition* of H, and then using that decomposition to do the partitioning.

The data graph is then loaded into memory by simply scanning the input relations, sorting them, and creating nodes as described above. If there are any attributes in the input relations that don't participate in the query, we push down appropriate projections (without duplicate elimination) to the underlying database to minimize the amount of data transferred over the network.

5.3 Traversing The Data Graph

In this section, we describe our algorithm that computes the aggregated groups of such a query Q, by traversing a data graph G_Q. For the sake of simplicity we will focus on the query that *counts* the number of tuples in each group and discuss how it is generalized in Section 5.3.4.

The high level idea behind Join-Agg is it to traverse the data graph, which represents the underlying data being joined, starting for one source node at a time and maintain certain partial aggregate values (in this case, counts) *at all reachable group nodes* in each iteration. We can later *combine* these values in order to obtain the final aggregate value of each group, instead of materializing the join at any point. The way this happens at a high level is by propagating the counts along

the data graph, starting from each unique source node, to the group nodes, while keeping track of certain path information (which we refer to as *path-id*s) along the way. These path-ids allow us to figure out which counts are derived from which paths in the data-graph and enable us to properly combine them to compute the correct count for each group.

5.3.1 Definitions & Axioms

Before we formally describe our general algorithm for executing these queries over a data graph G, we enumerate a few core definitions and axioms for concepts that we be regularly reference in the algorithm description. The execution algorithm we propose revolves around traversing the data graph and maintaining certain information along the way in order to *directly* output the groups in the result.

Definition 5.3.1. A *rooted tree* (also formally known in the context of directed graphs as an *arborescence*), is defined as a directed subgraph that consists of a tree, with a single root node, therefore containing *exactly one* path between that root node and every leaf node.

Definition 5.3.2. Let $C(n_1, n_2)$ denote a *count* between nodes n_1, n_2. We conceptualize the traversal of the data graph as equivalent to *conducting joins* between the tuple that each element of the data graph represents, thus generating new tuples which we want to avoid materializing. A count represents the *number of tuples generated along all paths* between node n_1 to n_2. Any such path *cannot* include a branching node (n_1, n_2 may themselves be branching nodes, but there cannot exist

a branching node in any path between them). More formally, using Axiom 5.3.1 we have $C(n_1, n_2) = \sum_{i=0}^{k}(|n_1 \rightarrow N_{j_i} \rightarrow n_2|)$, where N_{j_i} the set of intermediate nodes in one of the k unique paths between n_1 and n_2.

Definition 5.3.3. Let $p = [v_{b_0}, v_{b_1}, .., v_{b_n}]$ denote a *path-id*. A path-id is a *list of branching nodes* found in a unique path from a source node s to a group node g_i. We maintain path-ids in order to logically re-construct all possible rooted trees which have s as their root, and include all group nodes $n_{g_1}, n_{g_2}, .., n_{g_i}$ in their leaves in order to compute the number of tuples within each group $(v_s, v_{g_1}, v_{g_2}, .., v_{g_i})$. Path-ids are unique identifiers for unique paths in the data graph, and are always paired with a path-id *count* described below.

Definition 5.3.4. A *path-id count* denoted by C_{p_i}, is defined as the count between two branching nodes n_{b_i}, n_{b_j} and is equal to $C(n_{b_i}, n_{bj})$, where path-id $p_i = [v_{b_0}, ...v_{b_i}, v_{b_j}]$. The path-id maintains information about the rooted trees this certain path is part of. The path-id count itself however represents the count between the *last two* branching nodes in the path-id (even though the path-id might include more than two branching nodes). In the case where $|p_i| = 1$ then $C_{p_i} = C(s, n_{b_j})$ where s a source node. The intuition here, thinking about this from a query processing perspective, is that we need to keep track of *how many tuples were generated* at the point where a relation joins with more than one other relation. Once we join with one of the relations, we need to go back and join with the rest one at a time. In order to do that properly (without actually materializing the join result) we need to know how many tuples were generated at that time in the query before it branches

off to multiple joins.

Definition 5.3.5. A *group-node count* denoted $c_i = C(n_b, n_g)$ is the count between the last branching node n_b of a path, and a group node n_g. Intuitively, a group-node count c represents the number of tuples generated by joining the tuples in the underlying relation that contain value v_b, with all intermediate tuples, and then also joining them with all tuples that contain v_g.

Definition 5.3.6. A *c-pair* denoted by $P = (p, c)$, is a pair consisting of a path-id and a group-node count. These pairs are recorded at every group node during the traversal of the data graph described in the algorithm in Section 5.3.2.

Axiom 5.3.1. Let $|n_1 \rightarrow N_j \rightarrow n_2|$, denote the number of tuples generated when there exists a path from n_1 to n_2 which includes a set of in-between (either branching or intermediate) nodes $N_j = \{n_{j_1}, n_{j_2}, ..., n_{j_n}\}$. By definition of the data graph (Section 5.2.3), there **must** exist tuples $t_1 = (v_1, v_{j_1}), t_2 = (v_{j_1}, v_{j_2}), ..., t_n = (v_{j_n}, v_2)$, (where a tuple t_i appears $m_{(n_l, n_r)}$ times in its corresponding relation), such that $\{(n_1, n_{j_1}), (n_{j_1}, n_{j_2}), .., (n_{j_n}, n_2)\} \in E$. The number of tuples generated is $|t_1 \bowtie t_2 ... \bowtie t_n| = m_{(n_1, n_{j_1})} * m_{(n_{j_1}, n_{j_2})} * .. * m_{(n_{j_n}, n_2)} = |n_1 \rightarrow N_j \rightarrow n_2|$, and is derived by taking the product of all edge multiplicities along the path.

5.3.2 Join-Agg Stage 2: Traversal and Multiplicities

Stage two of this algorithm traverses G_Q in a depth first fashion starting from each *source node*, properly keeping track of the cumulative edge multiplicity along the way, and finally setting the appropriate *c-pairs* at all reachable group nodes.

A depth first traversal starting from source node n_s to the rest of the group nodes consists of multiple different rooted trees, each of which ends up at a potentially different set of leaf nodes (group nodes). Every rooted tree that reaches *exactly one of each type* of group node, corresponds to a tuple (or set of identical tuples) in the result of the join $\mathcal{R}$. The purpose of this algorithm is to *count* the number of such rooted trees that each combination of group nodes has *in common*.

The result of the traversal step is a set of lists L, containing one list of *c-pairs* associated with each group node n_g reachable from n_s – let l_{n_g} denote each such list. Again, (p, c) denotes a *c-pair*, comprising of a path-id p, and a group-node count c. There is also a path-id count C_p associated with each unique path-id (we define the terms path-id, path-id count and group-node count in Section 5.3.1).

We now outline the process that traverses the data graph and sets the appropriate c-pair lists at every group node n_{g_i}. We start at a source node n_s, and conduct a DFS traversal. Let p_c denote the current path-id, c_c denote the current count, and n_c the current node being visited. Also let n_{c_i} denote the i'th neighbor of n_c, and $m_{(n_c,n_{c_i})}$ denote the multiplicity of the edge between them.

We now define a recursive *visit(n_c)* function: if n_c is a group node, record $(p_c, c_c) \rightarrow l_{n_c}$, and return. This is the base case of the recursion. If n_c is not a **group** node, for each $n_{c_i} \in out(n_c)$, if n_{c_i} is a **branching** node, update $c_c = c_c * m_{(n_c,n_{c_i})}$ with the current neighbor's multiplicity, append n_{c_i} to the path-id p_c, and reset $c_c = 1$. The reason we reset the current count is because we now need to keep track of the count along the *new path* since we encountered a **branching** node. If n_{c_i} has already been visited by this traversal (the traversal starting from n_s), simply

update that path-id count to $C_{p_c} = C_{p_c} + c_c$ and return. Next, recursively *visit* every $n_{c_i} \in out(n_c)$. This can be seen as a form of computation *caching*. If we've been through a path in a current traversal, we don't need to go through it again, whereas in traditional execution, this path would be computed multiple times (in the form of joining tuples).

5.3.3 Join-Agg Stage 3: Result Generation

Finally, we end up with a list l_{n_g} for every n_g reachable from n_s – let $N_{n_s} = [n_{g_0}, n_{g_1}, ..., n_{g_k}]$ denote this set of group nodes. We utilize these lists in order to generate the final result groups. The *intuition* behind this process is that a group $(v_s, v_{g_0}, v_{g_1}, ..., v_{g_k})$ in the final result will only have a non-zero count value iff there is *at least one* rooted-tree in the data graph with n_s as the root, and N_{n_s} as leaves. Every c-pair set during the *traversal* stage of the algorithm will contain a path-id that is part of such a rooted-tree. There is a count computed for every such rooted-tree. The goal of this stage of the algorithm is to use these c-pairs set at every n_g in order to re-construct all of the rooted-trees that contribute to the result, and finally compute the *sum* of all of their counts. That sum is equal to the size of the output group.

First, we separate the group nodes reached by the traversal into a set of *buckets*. The *combination* of all c-pairs found in all nodes in N_{n_s}, will result in the final count for the group $(v_s, v_{g_0}, v_{g_1}, ..., v_{g_k})$; if this count is non-zero, the group is output to the final result. We will now properly explain how this *combination* of c-pairs is

conducted.

We partition $n_g \in N_{n_s}$ nodes into $|R_G|$ *buckets*, one for each group relation $G_i \in R_G$, by adding a node into a bucket B_i if it was derived from group relation G_i. Let B denote this set of group node buckets. Next, for each bucket $B_i \in B$, we output a list of tuples F_i that we will combine in order to generate the final result. The way we output these tuples is the following: For each node $n_i \in B_i$, for every c-pair $P_i = (p_i, c_i) \in l_{n_i}$, we output a tuple $(n_i, (p_i, c_i)) \rightarrow F_i$, so that we keep track of which group node each c-pair in F_i came from. Let F denote the set of lists output from all buckets in this step.

Lastly, in order to construct and aggregate all distinct groups that are in the final output and their associated counts, we conduct a *prefix-join* (denoted as $\bowtie_\sim$) of the lists $F_i \in F$ on the *path-id* in a pair-wise fashion. In this prefix-join, two tuples match if their path-ids *share a common prefix*.

More specifically, let $\sim$ define a binary relationship between path-ids, that indicates they share a common path prefix. Let p_1, p_2 path-ids where l_1, l_2 are their respective lengths, and $l_1 \leq l_2$. We say that $p_1 \sim p_2$ iff $p_1[0..l_1] = p_2[0..l_1]$.

Therefore, for every tuple in $F_0 \bowtie_\sim F_1 \bowtie_\sim ... \bowtie_\sim F_i$, we compute a value that will *contribute* to a group in the final result. Say we're computing $F_1 \bowtie_\sim F_2$; Let a tuple $f_1 = (n_1, (p_1, c_1)) \in F_1$ and $f_2 = (n_2, (p_2, c_2)) \in F_2$. If $p_1 \sim p_2$ we output $f_3 = (\{n_1, n_2\}, (p_i, c_3))$ where $c_3 = C_{p_1} * C_{p_2} * c_1 * c_2)$ and $p_i = p_1$ iff $|p_1| < |p_2|$ or $p_i = p_2$ iff $|p_1| > |p_2|$ i.e., the path-id with the smallest length. We only multiply the result with the path-id count of each unique path-id, *once* – if e.g.,we joined $f_3 = (n_3, (p_i, c_3)) \bowtie_\sim f_4 = (n_4, (p_i, c_4))$ (a tuple with the exact same path-id), the

output tuple would be $f_5 = (\{n_3, n_4\}, (p_i, c_5))$ where $c_5 = c_3 * c_4 * C_{p_i}$.

For every iteration of the algorithm, we start from a source node n_s, we end up getting an F set of c-pairs, out of which we output all f_i tuples resulting from the prefix-join described above iff they have non-zero counts. After the end of step 3, we will have output all groups, that have any combination of values where every value comes from a different group relation.

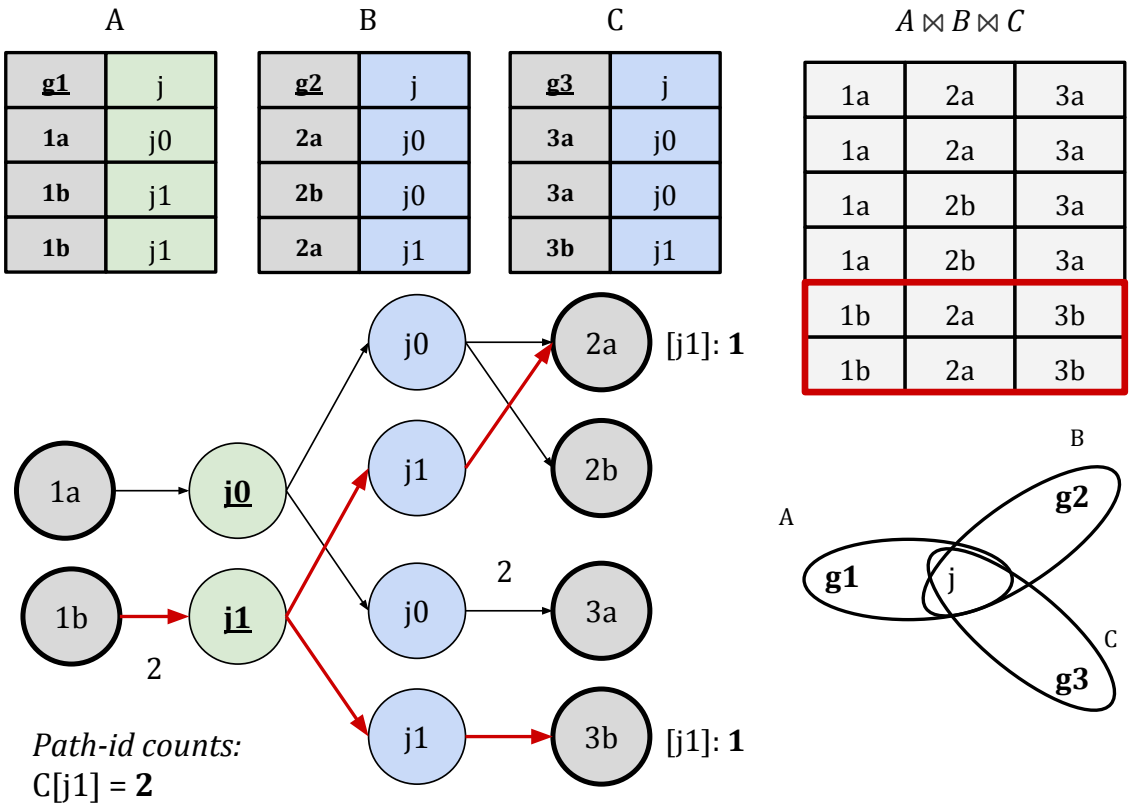


Figure 5.4: A rooted tree in the data graph corresponds to at least one tuple in $\mathcal{R}$ that contain the values at the root and the leaves of the rooted tree (the source node and the group node values).

Example 5.3.1. Consider the example on Figure 5.4, showing the data graph and the join result $\mathcal{R}$ for this query. The red arrows showcase an example of a *rooted tree*, with source node `1a` as its root. Every possible rooted tree in G_Q which includes one

of each type of group node in its leaves directly corresponds to a tuple in the join result $\mathcal{R}$. Since the idea is to *avoid* materializing $\mathcal{R}$, we instead traverse this graph, and set a c-pair at every group node every time it is visited, identifying the path that reached a given node by its unique *path-id*. Here we can see that for group node $2a$, its c-pair list is $l_{2a} = \{([j1], 1)\}$, and for $3b$ we have $l_{3b} = \{([j1], 1)\}$ accordingly. We will transform these lists into sets of tuples $\{F_1, F_2\}$ where $F_1 = \{(2a, ([j1], 1))\}$ and $F_2 = \{(3b, ([j1], 1))\}$. We compute the prefix-join $F_1 \bowtie_{\sim} F_2$, which outputs the tuple f_3 with the value $(2a, 3b)$ and the count $1 * 1 * 2$ (the count of f_1 times the count of f_2 times the path-id count for the path-id $[j1]$). Finally, $(1b, 2a, 3b), 2$ is output. There is such a tuple computed *for every rooted-tree in the data graph* that has $1b$ as its root, and $2a, 3b$ as its leaves. The sum of the counts for every unique group is equal to the size of the group in the final result.

5.3.4 Other Aggregation Functions

The list of basic aggregation functions supported by most SQL execution engines includes `COUNT`,`SUM`, `MIN`,`MAX` and `AVG`. We argue that our ability to execute `COUNT` without outputting individual intermediate results generalizes directly to the rest of these basic aggregation functions.

`SUM`: Firstly, `COUNT` can be thought of as a special case of `SUM`, if we assume that every single tuple in the group includes an attribute for which the value is always 1. If the value of such an attribute is not 1, while executing the query , we can simply keep track of the running sum of tuples, that include the attribute values being

summed over, instead of just the running multiplicity of generated tuples. The sum would then need to be multiplied by the count for a specific group, which would then output the correct result.

`MIN, MAX:` These two functions would only require keeping track of a single value and do not require maintaining counts at all.

`AVG:` This requires keeping track of the *sum* of the certain combination of attribute values that need to be averaged over, as well as the count that the current version of our algorithm is maintaining.

5.4 Complexity Analysis

Here, we provide a high-level analysis of the computational complexity of executing a join-aggregate query, with the goal of showing the asymptotic benefits of our approach. We make several simplifying assumptions for clarity. For any relation R_i that we use in the examples below, we make a *uniformity* assumption about all join condition attributes. Moreover, any join between relations in the below examples are natural joins. Again, let $\pi_g(R_i)$ denote the domain of values for attribute g in relation R_i. We assume that all relations R_i in any example are of a constant size $|R_i| = n$. Also, let $|\pi_{g_i}(R)| = a$ denote the number of unique values of a group attribute g_i and $|\pi_p(R)| = b$ the unique values of a join attribute. We assume they are uniform i.e. all group attributes have a domain of size a and all join condition attributes have a domain of size b.

We contrast the time and space complexity of the algorithm *traditional RDBMSs*

would use to compute join-aggregate queries, our JOIN-AGG operator, and an idealized pre-aggregation approach. We do this by choosing three simplified example queries. For the traditional RDBMS execution we assume that a *sort-merge join* operator is used to compute the join between any two joining relations and that *neither* of the joining relations have indexes on the join condition attribute. In most modern database systems, a *hash-join* operator of some sort would be chosen by the query optimizer, but only if the optimizer accurately estimates the amount of memory required for storing the hash-table given the amount of memory available. We discuss our relevant findings in regards to how PostgreSQL chooses a join algorithm in practice in Section 5.6.2. In terms of how *aggregation* is performed in the RDBMS, we assume again that a *sort-aggregate* operator is used instead of a *hash-aggregate* operator.

It's also important to note that we are loading in a node in the data graph for every unique value in each distinct relation. For the sake of simplicity we assume *set* semantics for any relation, i.e., we assume that every tuple in a relation is unique. Note that there are various optimizations possible for the first two examples below, which nonetheless don't change the asymptotic complexity of the algorithm.

Self-Join: Here, for a single relation $R(g, , p)$, the join-aggregate query is equivalent to doing a self-join on p, and a group-by on the two copies of the attribute g (shown in Figure 5.5a). More specifically, let the two copies of $R = \{R_1(g_1, , p), R_2(g_2, , p)\}$, and let $G = \{g_1, g_2\}$, giving us the JOIN-AGG query $Q(\{R_1, R_2\}, G)$.

Traditional RDBMS: A traditional RDBMS would compute the join and then

aggregate the results. Let $\pi_{g_1}(R_1) = \pi_{g_2}(R_2) = a$ and $\pi_p(R_1) = \pi_p(R_2) = b$. The **join** computation requires us to first sort both of the relations ($O(2nlogn)$), then compute the join between them and output all result tuples. The complexity of computing the join will be equal to the number of output tuples in the result of the join, which is $\frac{n^2}{b}$. Overall, the join process takes $O(2nlogn + \frac{n^2}{b})$ steps. The **aggregation** process that follows the join requires a sorting of the join result, giving us a total time of $O(\frac{n^2}{b}\log(\frac{n^2}{b}))$.

Join-Agg Operator: The **number of vertices** in the data graph $|V| = 2a+2b$. The **number of edges** in the data graph $|E|$ is at most $2ab$. At the **traversal stage** of the algorithm, we need to conduct a full *dfs* traversal of the graph for every *source* node, of which we have a here. A single *dfs* requires $O(|V| + |E|)$, therefore overall, we will have: $O(a * (|V| + |E|)) = O(a * (2a + 2b + 2ab)) = O(a^2b)$. Since there is no merging step for this query, the **result generation** requires a pass over the reachable group nodes, and there may be at most a^2 different results. Overall we have a time complexity of $O(a^2 + a^2b) = O(a^2b)$.

Pre-aggregation: For this query, we can do pre-aggregation on R by aggregating on $\{g, p\}$, thus reducing its size to at most ab. The total execution time then reduces to $O(a^2b\log(a^2b))$, and is thus comparable to the JOIN-AGG operator runtime. However, the maximum memory consumed by this approach at any point is $O(a^2b)$, whereas the JOIN-AGG operator consumes at most $O(ab)$ memory (i.e., memory equal to the size of the data graph).

Comparison: It is easy to see that, the JOIN-AGG operator performs better asymptotically than the traditional approach if $ab < n\log(n)$, i.e., if the number of

unique values of g and/or the number of unique values of p is small relative to the relation.

Chain Join: Next we consider a simple chain join between four relations, $Q = R_1(g_1, \_, p_0) \bowtie R_2(p_0, \_, p_1) \bowtie R_3(p_1, \_, p_2) \bowtie R_4(p_2, \_, g_2)$ (shown in Figure 5.5b), still maintaining 2 group attributes in total.

Traditional RDBMS: Computation of the **join** result is again dominated by the generation of the result tuples, and requires $\frac{n^4}{b^3}$ steps. For **aggregation**, we again would sort the join result and scan it to output the result groups, overall requiring $O(\frac{n^4}{b^3} log(\frac{n^4}{b^3}))$.

Join-Agg Operator: The **number of vertices** in the data graph $|V| = 2a+6b$. The **number of edges** in the data graph $|E|$ is at most $4ab$. Similarly as the above case, the **traversal stage** will take $O(a * (|V| + |E|)) = O(a * (2a + 6b + 4ab))$. Overall, we again have the total time complexity of $O(a^2b)$.

Pre-aggregation: With aggressive pre-aggregation over the input relations and all intermediate results after they are generated, the time complexity of the join-at-a-time approach can be reduced to $O(a^2b \log(a^2b) + ab^2 \log(ab^2))$. However, the memory consumption of this approach reaches $O(max(a^2b, ab^2))$ at various points during execution as intermediate results are materialized.

Comparison: As our experimental results also validate, the benefits of a single operator are clearly apparent here, with potentially very large gains coming from more careful and combined evaluation.

Chain Join w/ 4 Grouping Attributes: Next we consider a chain join between

four relations, $Q = R_1(g_1, \_, p_0) \bowtie R_2(p_0, \_, g_2, p_1) \bowtie R_3(p_1, \_, g_3, p_2) \bowtie R_4(p_2, \_, g_4)$, but with a total of 4 grouping attributes.

Traditional RDBMS: Since we assume the relation sizes and selectivities are unchanged, the total time complexity here remains $O(\frac{n^4}{b^3} log(\frac{n^4}{b^3}))$.

Join-Agg Operator: The **number of vertices** in the data graph $|V|$ is $O(n)$ here because there will be two sets of multi-nodes here, one for R_2 and R_3 each. The **number of edges** in the data graph $|E|$ is at most $O(\max(ab, n))$. Similarly as the above case, the **traversal stage** will take $O(a*(|V|+|E|)) = O(\max(an, a^2b))$ time. However, the result generation stage is more complex here as we have to maintain "paths" at the reachable group nodes and merge them at the end. Both the space and time complexity here is dominated by the number of different paths to the g_i nodes. In the worst case, there may be $O(\frac{n^2}{b})$ such paths per g_i node, giving us a total time complexity of $O(\frac{n^2}{b} \log(\frac{n^2}{b}))$ per source node. The overall complexity then is $O(a\frac{n^2}{b} \log(\frac{n^2}{b}))$, and the space complexity is $O(a\frac{n^2}{b})$.

Pre-aggregation: The pre-aggregation possibilities are somewhat limited here since the intermediate results contain a larger number of attributes, and thus have limited duplicity. If we assume there is no reduction due to pre-aggregration, then the time complexity here is similar to the traditional approach, giving us $O(\frac{n^4}{b^3} log(\frac{n^4}{b^3}))$ time complexity. However, another lower bound can be calculated using the similar worst-case assumption as above for the Join-Agg operator, where we assume all possible combinations of values exists in at least one join result, giving us a time complexity of $O(a^4b \log(a^4b))$, with a space complexity of $O(a^4b)$.

Comparison: The complexities of Join-Agg and pre-aggregation approaches

are very different in this case. The pre-aggregation approach may perform somewhat better if the number of unique values for the group attributes is small relative to the join attributes; however in that scenario, we expect the number of different paths to a g_4 node to be significantly smaller than b^3 (which assumes a worst-case situation that won't occur in practice). As above, we see that the JOIN-AGG space complexity is lower by a factor.

Branching Join: Next we consider a 5-relation branching query $Q = R_1(g_1, \_, j_1) \bowtie B(j_1, j_2, j_3, j_4) \bowtie R_2(j_2, \_, g_2) \bowtie R_3(j_3, \_, g_3) \bowtie R_4(j_4, \_, g_4)$, with a group by aggregate on four attributes from four different relations.

Traditional RDBMS: As above, the join computation time is dominated by the generation of the result tuples, giving us a total time of $O(\frac{n^5}{b^4} log(\frac{n^5}{b^4}))$.

Join-Agg Operator: The **number of vertices** in the data graph $|V| = 4a + 4b + n$ (since every tuple from B will be a different node). The **number of edges** in the data graph $|E|$ is at most $O(\max(n, ab))$. The **traversal stage** would again take $O(a * (|V| + |E|)) = O(\max(an, a^2 b))$ in total. The **result generation** however requires merging the lists of paths at each of the reachable grouping attribute nodes (g_2, g_3, g_4). It is easy to see that maximum number of different paths from a given source node to any of the destination grouping nodes (say a g_2 node) is n [1], thus giving us a result generation time of $O(n \log(n))$ per source node. Since this has to be done for each of the g_1 nodes, the total time for result generation is bounded by $O(an \log(n))$. Thus the overall complexity is $O(\max(a^2 b, an \log(n)))$, with a

[1]Note that this is because we have a single branching point in the data graph for this type of query. If we had x *recursive* branching points, this upper bound increases exponentially.

space complexity of $O(\max(n, ab))$. Unlike the bounds so far, we don't attempt to substitute n with a and b as the bounds become very loose in that case.

Pre-aggregation: The pre-aggregation possibilities are somewhat limited here (outside of the input relations) since the intermediate results contain a larger number of attributes, and thus have limited duplicity. The largest intermediate result we may generate here is $I(g_1, g_2, g_3, j_3, j_4)$, assuming we join R_1 with B followed by R_2, R_3, R_4 in that order (with aggressive pre-aggregation at every step). The size of $R_1 \bowtie B \bowtie R_2 \bowtie R_3$ can be estimated at $O(\frac{n^4}{b^3})$, and I is the result of projecting out j_1 and j_2 from that join result (and any other attributes from those relations that did not participate in the join). However, it is difficult to estimate the reduction in size from that projection. If b is sufficiently large compared to n (i.e., $b > \sqrt{n}$), then under uniformity assumptions, we expect minimal reduction in the size. Thus, in general, we expect the total time and space complexity of the pre-aggregation approach to be very high compared to the Join-Agg operator.

Comparison: Join queries with branching really illustrate the benefits of a holistic approach to executing such queries. The benefits over the traditional approach, even with aggressive pre-aggregation, come from the ability to avoid generating large intermediate results, and exploit "caching effects".

5.5 Implementation Details

The data graph we load into memory is stored in a data structure resembling a Compressed Sparse Row (CSR) graph representation. We store a list of all `Edge`

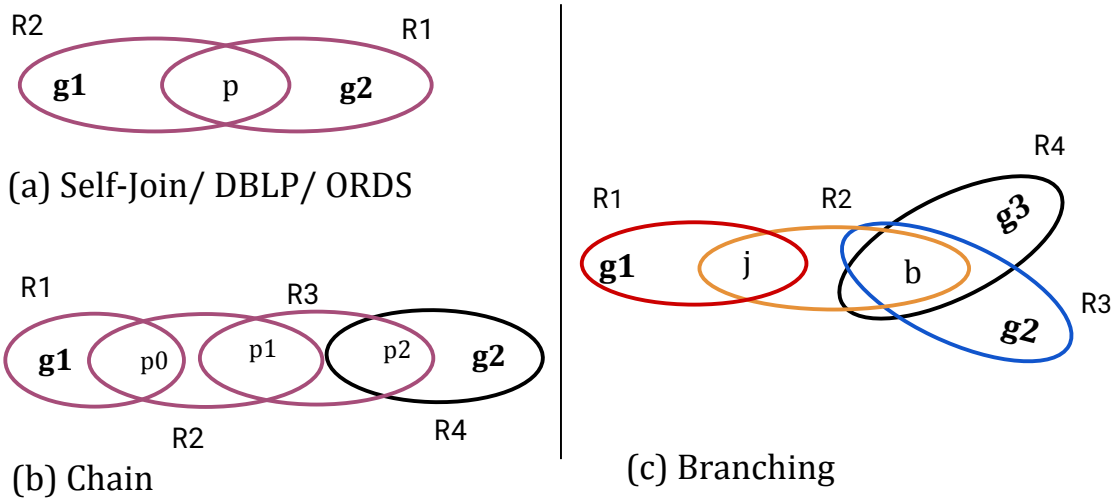


Figure 5.5: Hypergraphs of example queries

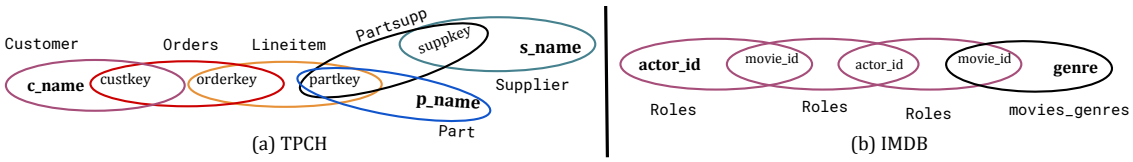


Figure 5.6: Hypergraphs of real world queries in the experiments.

objects in the graph called `outNeighbors`, and a list of `Node` objects. Each `Node` object contains the `List` of attribute values the node is comprised of (note that a `Node` can have values from multiple attributes in its relation if it is a *multi-node*). `Node`s also include one `Integer` for the *offset* value, and one `Integer` that stores the *number of neighbors* that particular node has. The offset value points to the outgoing edges that correspond to the particular `Node`; i.e., the outgoing edges of a node `n` would start at `outNeighbors[n.offset]` and end at `outNeighbors[n.offset+n.numNeighbors]`. `Node`s also include an `Integer` value representing the *type* of node (source, branching, group, etc.). Group nodes in particular, are assigned a unique `Integer` that references the relation they came from in this field. `Edge` objects store a reference to their outgoing neighbor `Node`, and a single `Integer`

value for their *multiplicity*. Path-ids are an integral part of the algorithm and are also stored as explicit objects, containing a `List` of branching node values, as well as an `Integer` for the path-id count.

Stage 1: Data Graph Loading: During the loading for the data graph, each relation is *sorted* by PostgreSQL, read in using a `JDBC` connector as a `List` of tuples, and each tuple is partitioned into its two (x_l, x_r) subsets. Duplicate tuples (after projection) in each relation are also *pre-aggregated* by PostgreSQL itself before being loaded into the data graph. A `HashMap` index is used to keep track of `Node` objects already loaded and access them in order to incrementally add each additional `Edge` to the data graph. For each relation being loaded, all children relations in the decomposition tree are subsequently loaded, as well as `Edge`s between `Node`s with overlapping values; these `Node`s intuitively map to joining tuples between the original relations. The CSR representation we use for our graph data structure is generally *immutable*, we therefore make sure to properly load in each `Node` and all of its `Edge`s entirely before moving on to the next one so as to never require to shift anything in the `outNeighbors` list.

Stage 2: Traversal: During the second step of the algorithm, the data graph is traversed in a *dfs* fashion starting from each source node. The `visit()` method is recursively called over the neighbors of the current source node, properly propagating the multiplicity as well as the path-id along the way. We keep track of the path-ids in each iteration inside of a `HashMap`, therefore a single hash-lookup is required to check if the current path-id has already been visited by this current traversal. If

so, we simply need to update its path-id count and continue with the next neighbor without continuing the traversal beyond that path since it has already been explored (for the current source node). This caching effect is one of the crucial optimizations that sets JOIN-AGG apart from other approaches such as *pre-aggregation* [35].

Stage 3: Result Generation: After a full traversal of the graph starting from a single source node concludes, we now have enough information to output all the groups that contain that source node as a value. First we separate the set of `reached nodes` into *buckets*, based on their *type*. If and only if *at least one* node from every relation in G was touched, do we take *any* further action in this stage.

Next we merge every c-pair in every node in each bucket into a single list of tuples ordered by path-id. We use a *k-way merge* algorithm to do this since c-pairs are all naturally sorted by path-id at the end of Stage 2. Next, for every list F_i generated by the previous step, we conduct a sort-merge join starting from the *smallest list* that contains path-ids of the longest size. We therefore sort the F_i lists first by path-id length (in a *decreasing* order), and then sort them by list size (in an *increasing* order). After the sort-merge join is completed, the result is sorted by the value of each output group lexicographically.

5.5.1 Pre-aggregation Implementation

In order to experimentally support our hypothesis described in Section 5.4, we implemented a simple in-memory database in Java which allowed us to manually describe query plans. We stored in-memory rows as Java `LinkedLists`, and stored

all values as `String` objects, as we did in the JOIN-AGG implementation, for the sake of consistency. We implemented a *hash-join* over two sets of tuples, *project* over a set of tuples, as well as a *hash-aggregate* group by operation over a set of tuples. We use the standard algorithms for hash-join. In particular, we create a `HashMap` on the join condition value for every tuple s_i in the *smallest* of the two sets of tuples, and probe that `HashMap` for every tuple l in the larger set, to generate all combinations l,s_i.

Optimizations: We included a few optimizations in order for our code to be as comparable as possible to a real in-database implementation. Firstly, we combined the project and hash-aggregate operators so that tuples are only read once, unnecessary columns are projected out, and the column is then hashed for aggregation in the same step. Moreover, due to the fact that each tuple's values are static (before it is joined), we compute the `hashCode()` of every tuple only once, upon its creation so that it doesn't need to be computed again when hashing the tuple (either at the join or aggregation step). At the hash-join stage, we allocate new memory for the output tuples only when outputting the join result. We store the pre-aggregated count at every stage in a separate field for each tuple.

5.6 Experimental Evaluation

We present an experimental evaluation over a series of synthetic and real datasets that showcase the benefits and trade-offs of the JOIN-AGG operator. We've generated 3 synthetic datasets for three types of queries described in Section 5.4,

the hypergraphs for which can be seen in Figure 5.5. We also present experiments on queries over TPCH [32] (using scale factor `SF=1`), DBLP [31], ORDS [79] and IMDB [80]. Each dataset is associated with a specific query, query hypergraphs for which are shown in Figure 5.6. Datasets DBLP and ORDS are both simple self-joins. Additional information about these datasets is shown in Table 5.1.

We implemented a prototype of the Join-Agg operator entirely in *Java*. We load the data directly from PostgreSQL into the JVM by using the `JDBC` connector. Our aim with this prototype is to showcase that the execution of aggregate queries over large-output joins can, in many situations, be evaluated *more efficiently* even *outside* of the RDBMS including the often substantial *overhead* of loading the data from PostgreSQL into the JVM. We advocate that a native version of Join-Agg implemented natively within an RDBMS itself in a lower level language would demonstrate an even wider performance gap in favor of Join-Agg. The main reason is that loading the data graph would naturally be significantly faster, because reading the data tuple-at-a-time using `JDBC` is a significant portion of the loading time overhead.

These experiments were all done on a single machine running Red Hat Enterprise Linux Server 6.9, with 64GB of RAM, and an Intel(R) Xeon(R) CPU E5-2430 0 @ 2.20GHz, using PostgreSQL version 9.4.18 and Java 8.

Query	Dataset	Selectivity	JoinR	Groups	Load(s)
Self-Join	S1	0.001	500 M	6.25 M	11.263
	S2	0.003	167 M	6.25 M	12.729
	S3	0.1	5.5 M	3.43 M	29.182
Chain	C1	0.1	837 M	5.04 M	15.203
	C2	0.3	64 M	1.71 M	20.198
	C3	0.5	23 M	1.04 M	22.048
Branch	B1	0.001/0.8	1.4 B	0.12 M	35.935
	B2	0.1/0.1	549 M	0.12 M	44.781
	B3	0.3/0.5	9.9 M	9.76 M	32.804
Real	TPCH		24 M	23.9 M	161.197
Queries	DBLP		105 M	87.8 M	253.536
	ORDS		59 M	7.50 M	20.881
	IMDB		4.4 B	13.1 M	138.956

Table 5.1: Characteristics about all synthetic and real datasets used in the experiments. *JoinR* shows the size of the join result before aggregation in Million (M) or Billion (B) tuples. *Groups* shows the number of groups output for each query in each dataset. *Load* is the total time required (in seconds) to load the data from PostreSQL to the in-memory data graph.

Dataset	Max Intermediate Result	JoinAgg Memory	PreAgg Memory
P1	987,285	0.097	0.259
P2	3,755,151	0.233	1.19
P3	13,414,963	0.547	3.3
P4	27,952,709	0.976	6.6
P5	45,762,103	1.1	>9GB
P6	66,326,006	1.3	>9GB

Table 5.2: Samples from the B2 dataset, the max memory consumption (max heap used in GB) when running Join-Agg or pre-aggregation respectively, as well as the size of the max intermediate result (in rows) that needed to be processed when using pre-agg.

5.6.1 Synthetic Datasets

The synthetic datasets that were used for studying the behavior on the example queries showcased in Section 5.4, were generated by pulling from a uniform distribution (using Java's `Random` class) of a certain set of values, based on the *selectivity* we wanted to emulate each time. We define the term *selectivity* as $s = |\pi_j(R)|/|R|$, where $\pi_j(R)$ the domain of unique values of attribute j in relation R. For each `S1,S2,S3` dataset, we generate a single relation $R(g, j)$ for which the join selectivity when joining with itself is roughly the one reported in Table 5.1. Similarly, for each `C1,C2,C3` dataset, we again generate a single relation with the specified join selectivity and use copies of that relation for each part of the chain–therefore all joins in the chain portray the same selectivity. For the `B1,B2,B3` datasets, there are two different selectivities specified, the first is for the join $R_1(g1, j) \bowtie R_2(j, b)$ and the second for the joins $R_2(j, b) \bowtie R_3(b, g_2)$ and $R_2(j, b) \bowtie R_4(b, g_3)$. Again, for each of the join condition attributes in each table, we generated each tuple by drawing from a uniform distribution of integers in the range $[0, s * |R|]$. Group attributes were generated the exact same way. The range that we used for generating the group attribute in each of these relations is roughly reflected by the number of output groups generated by the queries. For the sake of simplicity all generated relations are of size $|R_i| = 500,000$ tuples.

Dataset (groups/size)	S1 (6.25 M/80)	S2 (6.25 M/26)	S3 (3.4 M/1)
PostgreSQL	499 s	181 s	11 s
JOIN-AGG	38 s	28 s	33 s

Table 5.3: Experiment for the Self-join example.

Dataset (groups/size)	C1 (5 M/165)	C2 (1.7 M/37)	C3 (1 M/22)
PostgreSQL	512 s	65 s	18 s
JOIN-AGG	21 s	22 s	24 s

Table 5.4: Experiment for the Chain example.

Dataset (groups/size)	B1 (125 K/11 K)	B2 (125 K/4 K)	B3 (976 K/1)
PostgreSQL	1104 s	393 s	18 s
JOIN-AGG	136 s	226 s	55 s

Table 5.5: Experiment for the Branching example.

Dataset (groups/size)	TPCH (23 M/1)	DBLP (87 M/1)	ORDS (7.5 M/7)	IMDB (13 M/340)
PostgreSQL	71 s	172 s	95 s	3422 s
JOIN-AGG	248 s	384 s	31 s	1156 s

Table 5.6: Experiment for queries over real datasets.

5.6.2 Tuning PostgreSQL

We evaluate the performance of Join-Agg by comparing it to running these queries directly over a state of the art RDBMS; PostgreSQL. One of the database parameters that proved crucial for these queries is `work_mem`, which specifies the amount of memory every distinct query operator can utilize within a single query. In a data warehouse setting, given the specifications of the server machine we used, `work_mem` would normally be set to around `256MB`. Setting `work_mem` to a very high value is generally not recommended because it increases the risk of the database running out of usable memory very quickly as *multiple* user queries are executed

simultaneously. JOIN-AGG on the other hand only asymptotically needs as much memory as is required to store the data graph, per query, thus enabling multiple such queries to practically be run simultaneously and efficiently whereas PostreSQL would need to use slow methods (e.g.,use `SortMerge` Joins and `GroupAggregate` for aggregation).

Nevertheless, to showcase the *best possible* performance we could get out of PostgreSQL on this specific machine, we set `work_mem` to `10GB`. This essentially allowed the PostgreSQL query planner to mostly choose the `HashAggregate` operator instead of `GroupAggregate` which can be orders of magnitude slower, depending on whether the `Sort` phase happens in memory or on disk. The query plan generated by PostgreSQL when running these aggregate queries, showed that it always chooses to use `SortMerge` Joins, and `GroupAggregate`, when it estimates the value of `work_mem` isn't high enough to fit the hash-table based on the estimated number of output groups.

We also observe that PostgreSQL is completely unable to estimate the number of tuples in the result set, and uses the *same cardinality estimate* as the result of the join, for estimating the number of groups in the result. Anecdotally, we estimate this is as the primary reason PostgreSQL may choose to use `GroupAggregate` and `SortMerge` joins, to ensure that the query will not run out of memory instead of trying to use operators that require hashing, which are faster but require significantly larger amounts of memory.

5.6.3 Join-Agg Performance Analysis

We studied the three basic types of queries that constitute the baseline for most join-aggregation queries over a database. Our overall conclusion was that Join-Agg can make a huge difference in query execution time for a query, as that query outputs *larger groups*. The larger the size of the groups in the output, the more there is to gain from Join-Agg. In cases where the output is comprised of small groups (i.e., of size 1), Join-Agg portrays comparable performance to the traditional approach when taking into account the fact that a large portion of the execution time in Join-Agg is taken *loading* the data out of the database.

Table 5.3, showcases the performance of a join-aggregation query over a single *self-join*. We can see that Join-Agg performs over to an order of magnitude better than PostgreSQL when we have a relatively large group size and the gap between the two systems closes as that average size leans towards 1. This makes sense since outputting many groups of size 1 indicates the intermediate result is close in size to the final result, thus materializing it is mostly inevitable. Similar behavior is seen for the *chain* example shown in Table 5.4. Note that when we have multiple non-key joins in a row as is the case with this example, the selectivities of those joins don't need to be absurdly low for Join-Agg to have a substantial difference in performance. This is because the intermediate results keep expanding as non-key joins progress resulting in the output of a very large set of tuples that then need to be aggregated.

In Table 5.5 we can see that for datasets `B1,B2,B3`, where we have three

group attributes from different relations, showcase a similar performance trend as the other examples. In the datasets that output large groups, Join-Agg performs up to an order of magnitude better whereas the performance of `B3` which outputs groups of size 1 on average, is comparable to PostgreSQL. Particularly for dataset `B1`, we have a very low selectivity (0.001) join for $R_1 \bowtie R_2$ whereas the other joins portray a high selectivity (0.8). We can see that even a single low-selectivity join in this complex query, can result in a huge ($1.4B$ tuples) intermediate output which Join-Agg helps to avoid materializing.

In the real datasets we experimented with, showcased in Table 5.6, we observe results consistent with the synthetic datasets. The `DBLP` (rf. Figure 5.5a) and `TPCH` (rf. Figure 5.6a) queries output very small groups, causing the time for loading the data graph to dominate the computation. The dataset `ORDS` [79] is a typical *market basket* dataset of invoices that contain multiple items. We are querying all item pairs and counting *how many times* they were bought together. `IMDB`, is graph pattern counting query over the *IMDB* movie graph as seen in Figure 5.6b. This query counts the number of paths between an actor and a genre, two hops away from that actor, i.e., even genres of movies that co-actors of theirs played in. For both of the latter queries the groups portray a higher average size and the benefits of Join-Agg start becoming apparent.

5.6.4 Pre-aggregation Performance Analysis

To experimentally validate our hypothesis in regards to how using pre-aggregation stacks up against our approach, we sampled the `B2` dataset – incrementally taking a larger sample. Information about the samples can be seen in Table 5.2.

Figure 5.7 showcases the difference in memory requirements between Join-Agg and pre-aggregation. We can see that in the case of pre-aggregation, as the size of the largest intermediate result required for the query *after* using aggressive pre-aggregation at every stage of the join increases, the maximum amount of memory required to complete the query increases rapidly. The memory required when it comes to executing Join-Agg however increases slowly since it only has to do with the size of the input data in combination with the largest amount of c-pairs that need to be stored at a single iteration (after we process any one source node).

Figures 5.8 and 5.9 showcase the computation time required for the execution of the branching query shown in Figure 5.5c. Due to the fact that our pre-aggregation implementation is relatively simple and done in Java (as discussed in detail in Section 5.5), a large portion of the computation comes down to garbage collection time. If however we only look at the amount of time spent doing actual computation as shown in Figure 5.10, we can clearly see the gap in performance between the two techniques, as was expected based on the complexity analysis in Section 5.4.

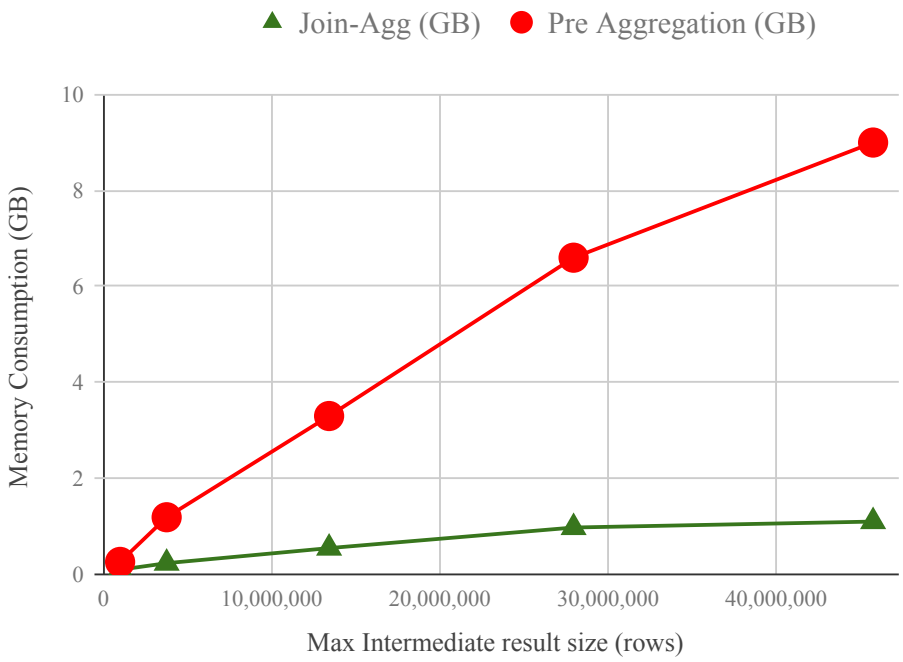


Figure 5.7: Maximum memory consumption (max heap used), at any point during execution. Each value in the y-axis represents the largest intermediate result we needed to store when using pre-aggregation at every stage of the join.

5.7 Summary

In this chapter, we proposed a multi-way database operator called Join-Agg that enables the memory-efficient execution of aggregation queries over joins that output large intermediate results, by executing the query over a *graph representation* of the underlying data called the data graph. We presented a detailed complexity analysis comparing our approach to the traditional binary joins-based approach as well as an idealized pre-aggregation approach. Our experiments show that Join-Agg operator can be over an order of magnitude more efficient than the traditional approach for a wide variety of queries, even when implemented outside of the RDBMS. We advocate that multi-way database operators may be the answer

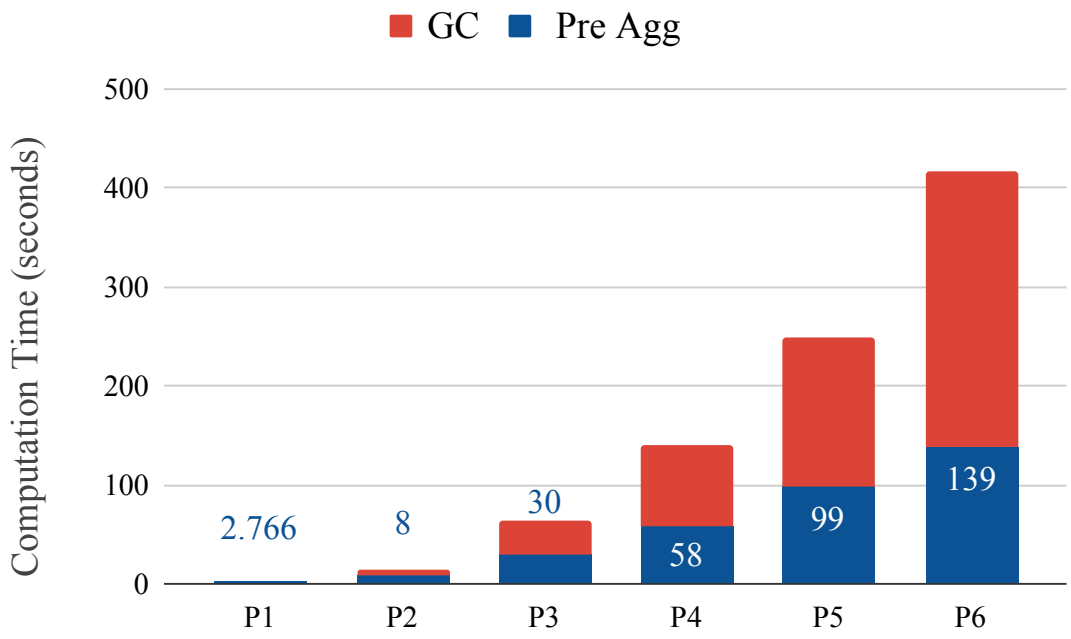


Figure 5.8: Total computation time spent when using pre-aggregation per sample, showing the portion of the computation time spent on garbage collection (GC).

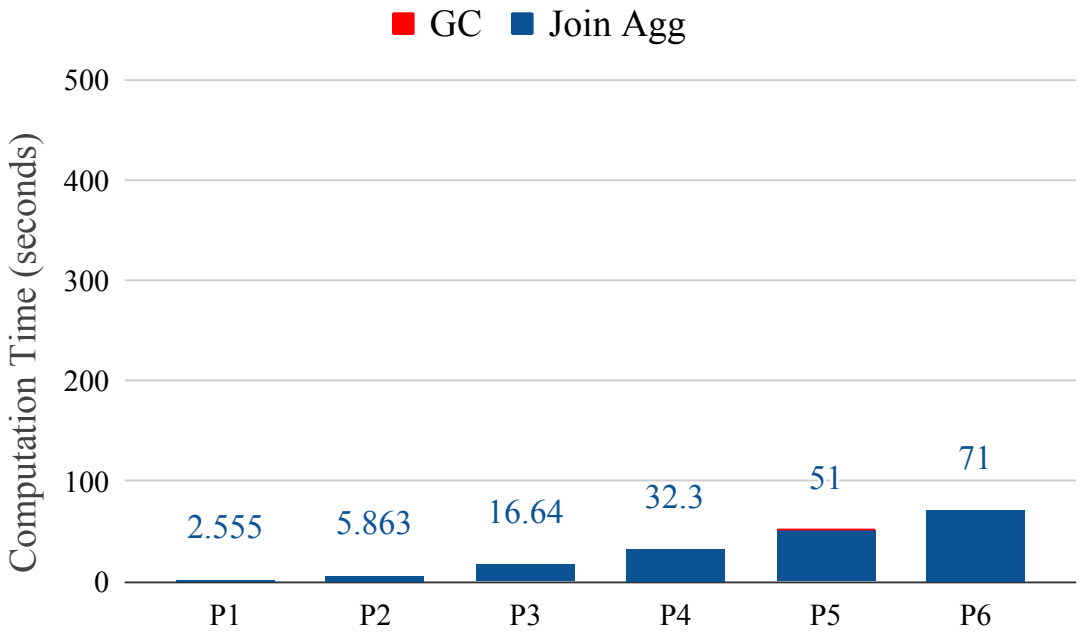


Figure 5.9: Total computation time spent when using join-agg per sample, showing the portion of the computation time spent on garbage collection (GC).

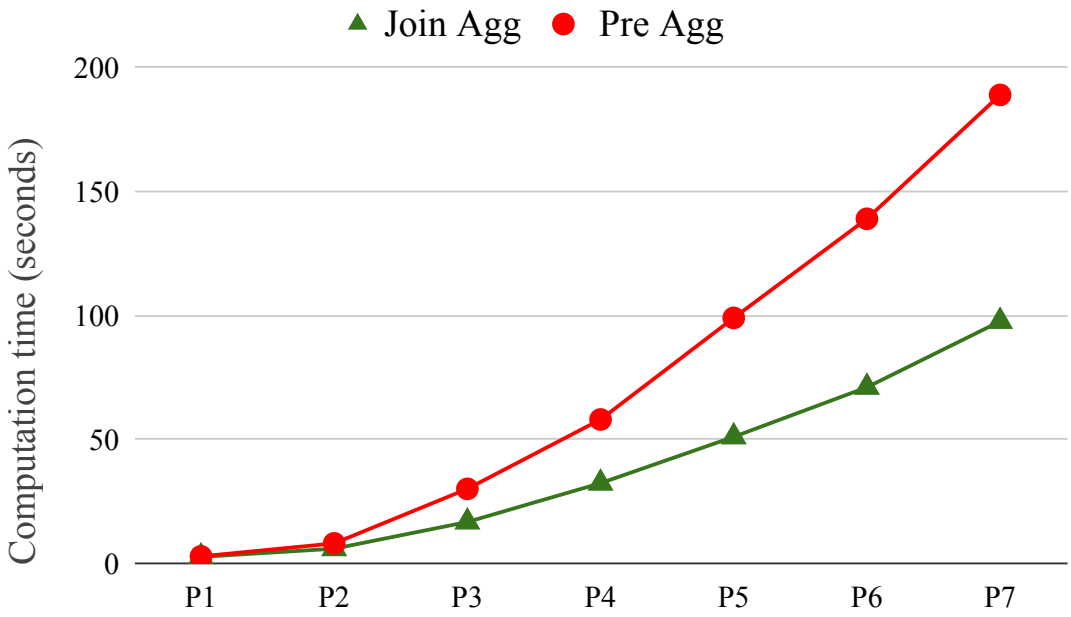


Figure 5.10: Only computation time (excluding GC time) for every sample dataset.

to dealing with the "non-normalized" data of the real world, which often leads to expensive non-key joins along with other aforementioned issues. They will enable users to continue leveraging RDBMSs for their OLAP analyses, requiring smaller amounts of resources.

Chapter 6: Related Work

In this chapter we discuss related work. We begin with enumerating the different types of graph storage systems, and graph analysis frameworks that have previously been considered, and how they approach the problem of conducting graph analysis over data in RDBMSs. We delve deeper into the connections and differences between our work on GraphGen and Join-Agg compared to past work on compressed graph representations as well as factorized databases. We also discuss work relevant to processing of graph collections as well as to processing of join-aggregate queries over RDBMSs.

6.1 Graph Data Management Systems

There has been much work on graph data management over the years, both on graph databases [81] and graph analytics systems [24]. This section focuses on the former.

Systems in this category include XML and RDF databases, as well as native property graph databases such as Neo4j [10], AWS Neptune [11], and OrientDB [12] to name a few. These systems are built from the ground up to revolve around the graph data model, and use specialized graph representations in their underlying stor-

age. They support SQL-like high-level query languages such as SPARQL, Cypher or PGQL, and also provide graph APIs like Gremlin or even direct access to the underlying graph, which is a necessity for expressing certain graph algorithms. Some systems (e.g., Neo4j) also offer a library of popular graph analytics algorithms to be used as black boxes. Most also provide support for ACID transactions. Migrating to this type of a system requires a complete buy-in into the graph data model which, as already discussed in Section 1.1, is usually not ideal since relational analytics still play a big role in most enterprises. Moreover, these systems are not as mature or scalable as most RDBMSs, which have been studied for many more decades. Users are also typically not interested in completely migrating their data over to a graph database if they aren't strictly dealing with graph-centric workloads.

Unlike graph databases, our work targets the scenario where the data resides in an RDBMS and migrating it to a graph database is not desirable. Nevertheless, there have been several recent pieces of work have also considered how to efficiently migrate a relational database to a graph database [28–30]. Table2Graph [29] 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 [30] is a MapReduce-based framework for extracting graphs from unstructured data through user-defined Java functions for node and edge specifications. GLog [44] is a declarative graph analysis language based on Datalog which is evaluated into MapReduce code towards efficient graph analytics in a distributed setting. Prior

work on translating schemas from one data model to another has considered more complex translation problems [82, 83]; for us, the translation (from GraphGenDL to SQL) itself is well-defined and straightforward, but the execution of the translated query and avoiding the generation of the final result, are the main challenges. This work typically focuses on creating the expanded graph most efficiently, and doesn't consider the possibility of generating a condensed representation like the ones we propose in GraphGen.

Lastly, several distributed graph analytics systems have adopted high-level declarative interfaces based on Datalog [43, 44, 84, 85]. Our use of Datalog is currently restricted to specifying which graphs to extract (in particular, we do not allow recursion). Combining declarative graph extraction (with systems such as GraphGen), and high-level graph analytics frameworks proposed in that prior work, is a rich area for future work. 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. Their use of declarative languages is aimed towards the specification of the graph traversal or algorithm to be executed over a given graph.

6.2 Graph Analytics Frameworks

There is a variety of systems developed in recent years with two main goals in mind: *simplifying* the process of writing graph analysis programs, and *executing* these programs efficiently on very large graphs. These *graph analytics frameworks*

are not concerned with transactional graph queries and expect a very particular graph format as their input. Most of the computation models for these systems are inspired by the Bulk Synchronous Parallel model [22]. Google's Pregel [23] is one of the systems that paved the way for multiple such "big graph" frameworks [24], later implemented in a variety of open-source and proprietary systems, one example of which is Apache Giraph [25]. Other systems in this space include GraphLab [26] and PowerGraph [27], that use a similar *Gather-Apply-Scatter* model with small variations in comparison to Pregel.

In order to benefit from these frameworks, users need to manually conduct the appropriate ETL in order to extract their graph of interest from an existing database, transform it into the appropriate input format, and write their graph algorithm which will then be executed by the framework. These computation models are also very particular and they do not provide direct access to the graph for arbitrary traversal– all traversals need to be altered in order to fit the computation model.

The work on graph analytics systems is largely orthogonal and complementary, as our techniques can be used for efficient extraction and in-memory representation in these systems (as we discussed in Section 3.4.4–implementing our condensed representations inside Apache Giraph).

6.3 RDBMS & Graph Analytics

Superficially, the most closely related branch of work to GraphGen 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 [4, 20], GRAIL [2], and SQLGraph [19], as even GraphFrames [86] show how to normalize 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 or queries to relational operations on those tables (see Figure 6.1). This is similar in spirit to the earlier work on storing semi-structured or XML documents in an RDBMS [13, 83]. EmptyHeaded [74] shows how worst-case optimal join algorithms may be used for efficient graph querying. On the other hand, G-SQL [87] and GQ-Fast [88] explore using graph processing engines to execute SQL queries efficiently, which is more closely related to our work on Join-Agg (see Chapter 5). GRFusion [89] materializes graph views in an in-memory graph data structure and uses it to conduct path queries, then combining it with relational operators to associate node and edge attributes to it. There has even been recent work in the programming languages community attempting to extend a relational query compiler in order to compile graph queries *posed in Datalog* into more efficient machine code that utilizes graph data structures [90].

However, those systems do not consider the problem of *extracting* different implicit graphs from existing relational datasets, and rather choose a relational schema for storing a given graph dataset. Further, those systems can typically only execute tasks (graph or XML queries) that can be mapped to SQL; while GraphGen 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 suitable for complex analysis tasks like

community detection, dense subgraph detection/matching, etc., which require *random access* to the graph, and cannot be efficiently, if at all, executed using basic SQL.

Aster Graph Analytics [91] and SAP HANA Graph Engine [92], also support specifying graphs within an SQL query, or using a custom definition language or script 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 only supports the vertex-centric API for writing graph algorithms. Our techniques could be used to reduce the graph memory footprints in those systems as well. IBM Db2 Graph [93] enable Gremlin queries over the Db2 RDBMS, by translating them into SQL. This only exposes the Gremlin language as an interface, and requires the user to provide a graph specification mapping (for which the syntax is not described).

Ringo [94] 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 large-output joins that are often necessary for graph extraction, or the alternate in-memory representation optimizations that we discuss here; it 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 [95], and we do plan to support Ringo as a front-end analytics engine for GraphGen.

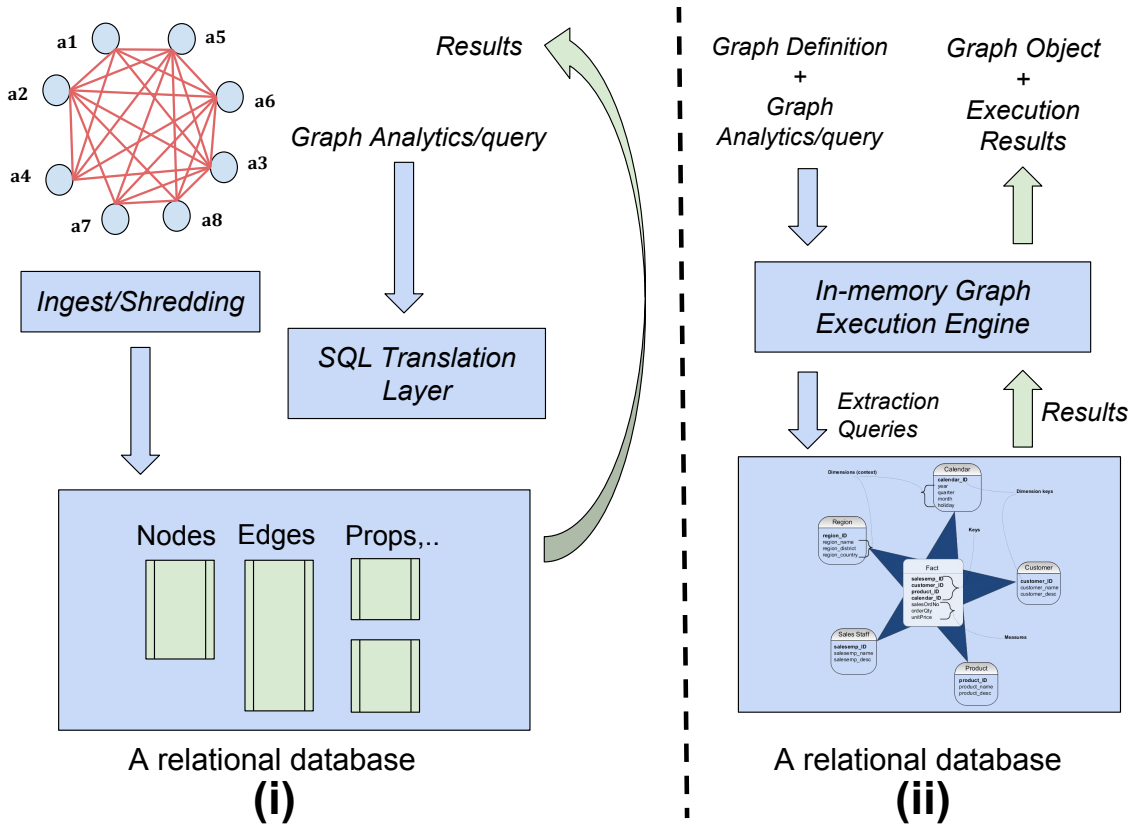


Figure 6.1: GraphGen (right) has fundamentally different goals than recent work on using RDBMSs for graph analytics (left).

6.4 Graph Compression

There has been a lot of work on graph, RDF, and XML compression, which can be roughly classified into [54]: (a) *succinct representations*, where the goal is to encode the graph with as few bits as possible [96–99]; (b) *structural compression*, where the graph structure is analyzed and changed to reduce the size of the graph [49, 51,100–102]; and (c) *lossy compression*, which aim to keep only sufficient information to answer specific classes of queries [103]. Our approach is complementary to the work on succinct representations and lossy compression, and can be seen as a form

of structural compression. Perhaps the most closely related work is VMiner [49], which identifies and exploits *bi-cliques* in the input graph to compress it losslessly (see Chapter 3.4.1.1 for more details). In a recent paper, Maneth and Peternek [104] present a technique to compress a graph through detecting repeating substructures; however, as with many of the graph or XML compression techniques, only certain types of queries can be executed against the compressed representation. The key difference between any of that work and our work on GraphGen is that: those techniques require us to first expand the graph before compressing it, i.e., they cannot operate on the implicit representation of the graph in the relational database; our approach aims to avoid the expansion step itself. We are therefore able to better utilize the structure in the data, which the expansion will remove (as our experimental results comparing to VMiner show). Further, we also support arbitrary graph operations on the compressed representations; a necessity for a general-purpose graph engine.

6.5 Multi-Query Optimization

Our contributions presented in Chapter 4 are mostly related to *multi-query optimization* (MQO), which is the attempt to generate an optimal combined evaluation plan by computing common subexpressions once, and then reusing them. Our goal however is to optimize a set of queries that are mostly identical (except for a single predicate). Recent work on multi-query optimization [68, 105, 106] has explored ways of optimizing query processing at a "global level" by attempting to

identify common sub-expressions across a set of queries and optimizing them in order to maximize re-use of portions of the computation across those queries, instead of attempting to find the local optimal way of executing each individual query. This technique however cannot be applied to our problem since there is no way to know what portions of the computation may overlap between each query that pertains to each seperate graph. Even if there was a portion of the query that was computed once and reused, the join condition dictated by the predicate f would be different for each query, and that join would need to be executed for each version separately.

6.6 Analysis Frameworks for Overlapping Graph Collections

There has been a lot of work on analysis frameworks for graph collections (e.g., graph snapshots over time) that aim at *optimizing analysis* of such collections mostly through exploiting overlaps between these graphs. Recent work on efficient multi-snapshot analytics [64] (SAMS) tackles the problem of automatically translating a graph algorithm in order for it to be run over a *set of snapshots* of a graph, while *overlapping computation* on portions of the graph that appear in multiple snapshots. Chronos [65] is a similar system, that portrays similar speedups for pull-based algorithms like *PageRank*, as PageRank matches the GAS (Gather-Apply-Scatter) model very well. For BFS (Breadth-First-Search) based algorithms, Chronos' vertex centric message passing slows it down in comparison to SAMS [64]. PED [66] looks at analyzing graph snapshots within a certain time window. Naturally, snapshots in the same time window tend to portray high overlap. PED takes

an edge *sampling* approach to computation instead. There has also been work on managing historical data for large *evolving* graphs [67], where a set of graph snapshots is stored in a distributed manner, with the goal of efficiently *retrieving* sets of snapshots of the graph and also leverage the fact that there may be large overlaps between them. Moreover, work on *temporal databases* [107], towards doing computations on certain time-based snapshots of datasets, discusses a simple language extension that enables specific types of queries to be easily computed over multiple snapshots of a dataset.

Our work in graph collections for GraphGen is orthogonal to this as we are concerned with properly obtaining each snapshot in the first place and efficiently loading them into an in-memory overlapping representation. Our work does *not* deal with how the user will later analyze the graphs.

6.6.1 Representing Graph Collections

Many of the aforementioned graph collection analysis frameworks use different in-memory representations for storing their graph collections. SAMS [64] uses a *vertex stack* which focuses on data locality, they also store multiple properties within every vertex, and a set of values for every graph instance/snapshot. In contrast, our approach aims mostly at minimizing memory required to store the graphs instead of optimizing for more time-efficient analysis. PED [66] stores graph collections as an "aggregate graph", where edges that belong to multiple different samples are simply *unioned*, which means that information about which edge is part of which sample is

lost.

6.7 Factorized Representation of Query Results

The condensed representations we propose (see Chapter 3) are similar to the notion of a "factorized representation of query results", where the goal is also to maintain the result of a query in a compressed form [75, 108]. This prior work proposes "schema-level factorizations" where the decisions about how to factorize a query result are based purely on the query, the relation schemas, and the functional and multi-valued dependencies that hold on the query result. On the other hand, our approach can be seen as exploring "data-dependent factorization". That prior work shows that the factorization of the result of an acyclic query without projections is *linear* in the size of the input database; however, for queries *with projections*[1], the storage requirements can be significantly higher.

For example, for the query that generates the *co-authors* graph, shown below, the schema-level approach entails expanding the graph and may require quadratic storage in the worst case.

Example 6.7.1. Let's examine the `Edges` definition (part of a graph extraction task) shown below:

```
Edges(ID1,ID2) :- AuthorPublication(ID1,PubID),
   AuthorPublication(ID2,PubID).
```

[1]Here we are referring to π projections in particular, where a `DISTINCT` is applied post-projection to eliminate duplicate data

The final query result here (i.e., the list of edges) is equivalent to the expanded graph, and our techniques in this paper are focused on avoiding the generation of that result altogether. Although this is also the focus of the work on factorized representations, their techniques work at the level of schemas and would not be able to avoid generating the full result. To elaborate, consider the following `Edges` definition that does not do the projection:

```
EdgesNP(ID1, PubID, ID2) :- AuthorPublication(ID1,
    PubID), AuthorPublication(ID2,PubID).
```

The query result for this query can be represented using the *f-tree* shown in Figure 6.2 (T1), and the size of the factorization (F1) is linear in the size of the joining relations, in this case, the size of the `AuthorPublication` relation. This factorized representation is, in fact, equivalent to C-DUP; both of these use explicit nodes to represent the different `PubID` elements. Another way to look at this is that, the query result here has a *multi-valued dependency*: `PubID` $\rightarrow\rightarrow$`ID1`, which these representations exploit.

However, both of these representations suffer from duplication since a pair of authors may share multiple `PubID`s. More specifically, although it is possible to generate the results of the second query with "optimal delay tuple enumeration" (Theorem 4.11 [108]), the same pair of authors may be generated multiple times with different `PubID`s. Projecting out the `PubID` attribute results in the *f-tree* shown in Figure 6.2 (T2). This *f-tree* is, however, equivalent to doing the join, removing the duplicates, and generating the full result, since for every author, we must list

out all of their co-authors (as shown in Figure 6.2 (F2)).

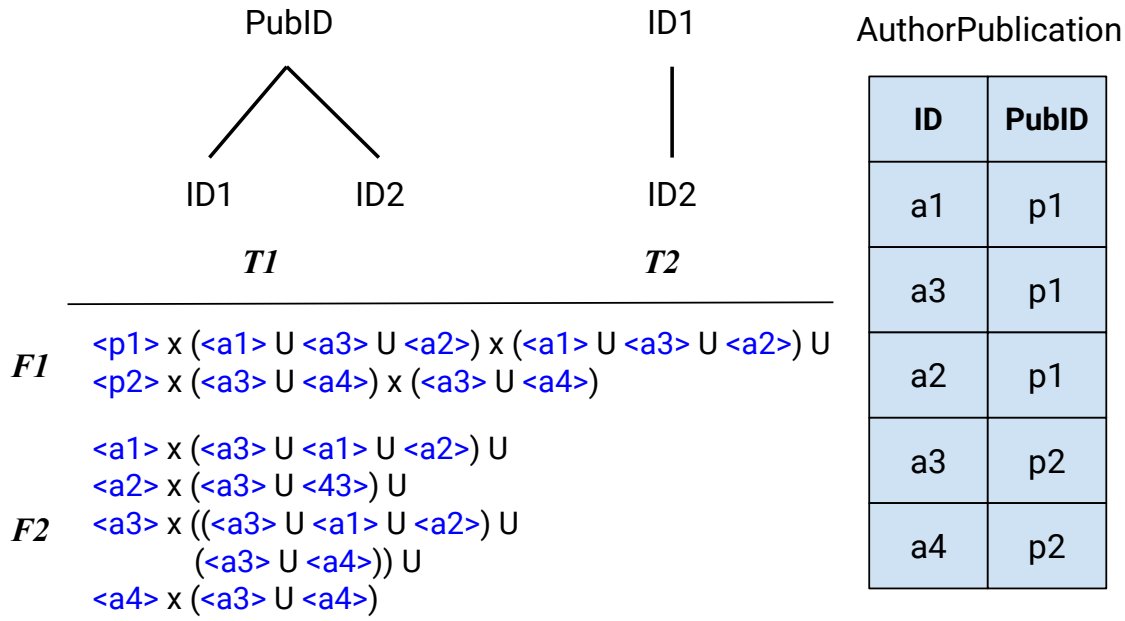


Figure 6.2: *T1* results in factorization *F1* (equivalent to C-DUP). *T2* results in factorization *F2* which is equivalent to the (expanded) graph.

The *data graph* paradeigm that we propose in Chapter 5 is also reminiscent of this factorized representation. Both representations aim at representing the underlying join while reducing the amount of data stored in order to do so. The data graph can also be connected to the idea of a *tuple hypergraph* which can cover all tuples in a query result [76]; it however serves a very different purpose.

Our main objective with the JOIN-AGG operator is to be able to compute *aggregations* over a representation like the data graph, especially in the case of complex acyclic joins. Several different works have considered the problem of executing group by aggregate queries against a factorized representation of a conjunctive query [109–113,113–115]. The key guarantees like constant-delay enumeration, however, do not extend to the kind of group by queries we focus on in this work, e.g.,

the "branching" query $R_1(g_1, j), R_2(j, b), R_3(b, g_3), R_4(b, g_2)$. Because all of g_1, g_2, g_3 (group by attributes) need to be present in the output, either (a) one of the other attributes needs to be eliminated (which requires generation of a large intermediate result), or (b) we have to iterate over all combinations of values for g_1, g_2, g_3 and compute the aggregate value for each combination (which can be prohibitively expensive if either the sizes or the number of group by attributes is large). In conclusion, the factorized representation *can* be used to compute the results of join-aggregate queries without materializing intermediate results, however *it is not able to do so in all cases* i.e., there exist conjunctive queries (like the one described in the example above) for which the factorized representation will need to *at least partially* materialize the intermediate result. Our approach with JOIN-AGG does not have that limitation.

6.8 Join and Aggregate Query Processing

Here we sketch some of the work in the general field of query processing that is related to our work on the JOIN-AGG query operator discussed in Chapter 5.

6.8.1 Worst-case Optimal Joins

Recent work on worst-case optimal joins [71,116–118] shows how to avoid large intermediate results during execution of multi-way join queries; we plan to integrate those techniques into our system as we generalize our work to allow cyclic extraction queries. However, for the class of queries considered in this paper (i.e., where the

edges are generated using a union of acyclic queries), those techniques do not provide any benefits over the classic Yannakakis algorithm [119]. Our challenge is that the final query result itself is too large.

In connection to our work on Join-Agg (discussed in Chapter 5), Joglekar et al., [120, 121] discuss how to generalize the work on worst-case optimal joins to aggregate queries. Their approach is largely complementary to [109] as well as our work. Recent work on FAQ [122] proposed a generalized way of viewing a very common type of aggregation query called a Functional Aggregate Query which they see parallels in multiple scenarios other than databases e.g.,matrix multiplication, probabilistic graphical models, and logic. The "InsideOut" algorithm proposed in FAQ however is not focused on executing SQL queries, like our work as well as the factorized databases work is aimed at doing. FAQ also assumes an optimal variable order, while this paper does not explore the benefits of choosing the optimal variable order (tree decomposition in our case).

6.8.2 Iceberg Queries

An *iceberg query* is a particular class of SQL queries, defined as an aggregate query, counting occurrences of target group instances of the `GROUP BY` clause columns, and filtering the results post-aggregation using a `HAVING` clause. These queries typically return a small fraction of the overall (potentially large) join result, (the tip of the iceberg). Iceberg queries are clearly a special case of the queries we're studying in this paper.

Fang et al. [123] propose a wide array of techniques for computing iceberg queries which focus on *minimizing the passes* done over the data (Disk I/O), being able to answer such queries in reasonable time, and doing so with a small amount of memory. The authors focus on combining two techniques: coarse counting (probabilistic counting), and sampling. These techniques may start causing issues as the final result increases in size. In a similar setting there has been work on efficiently computing the *iceberg* `CUBE` [124–126], which is largely orthogonal to ours, since this paper focuses on the general case of outputting all groups. Developing techniques for more efficient *iceberg* queries using our Join-Agg operator are delegated to future work.

Walenz et al. [73], presented a series of optimizations applicable to certain types of iceberg queries. The main focus of this work is to use formal methods towards *automatically identifying* whether a general SQL query would benefit from certain specialized optimizations for evaluating certain types of iceberg queries, as well as towards automatically using such optimizations during evaluation. The optimizations they consider involve pruning techniques based on memoization and complex non-equality join conditions. Given a general SQL query, their methods systematically identify if any optimization technique is applicable, and use it during execution of the query. Similarly to us, the authors implement and wrap the above optimizations into a custom database *join operator*. The work in this paper is largely orthogonal to ours since it mainly deals with complex join conditions, it does not focus on minimizing extra memory consumption during execution, and is more aimed at providing formal methods for automatically identifying queries that

would benefit from these specialized optimizations.

6.8.3 Similarity Joins

Work on *similarity joins* [127–130] uses various techniques to prune join computation. In a similarity join between two relations, (on a *string* join condition), a pair of tuples join if their join attribute similarity surpasses a threshold. This can be directly mapped onto the iceberg query problem where the aggregation function is `COUNT`. From this perspective, iceberg queries aim at finding the tuples in the result that have a certain number of join condition attributes in common, which surpasses a threshold. Similarity join techniques are almost exclusively signature-based (strings are collapsed into smaller signature sets). In a lot of these approaches, an "inverted index" is built beforehand, which in a sense resembles our in-memory graph structure. These join algorithms are however only studied for binary operations, similar to the self-join case.

6.8.4 Data Reduction Operators

There have been many early papers that observed this idea of being able to push aggregation past joins to reduce the amount of data that needs to be joined [131–134]. This type of work aims at re-arranging group-by operators in the logical query plan tree, moving them after or before joins accordingly. These techniques don't deal with avoiding materialization of intermediate results in situations when group by operators cannot be pushed down. Aggregation can only be

pushed down if it can be partially applied to a single relation, thus reducing that relation's cardinality. In the general case however when the query contains a series of group by attributes, each one coming from a different relation, there's no way to apply any complex aggregation to a single relation because the aggregation applies to the result.

Larson et al., describe techniques for doing *partial pre-aggregation* [35]. They describe a way to apply pre-aggregation to input relations when another aggregation is conducted on their join result. A simple hash table is used to aggregate tuples in the same relation, thus reducing the number of tuples joining with the next relation. As groups are pre-aggregated sequentially, if the number of pre-aggregated groups surpasses the memory capacity, partially pre-aggregated tuples are output to make room for new groups; therefore the pre-aggregation can be incomplete. Those same-group tuples will be aggregated later on at the final aggregation step. They also describe techniques to combine this pre-aggregation process with a join by pre-aggregating while reading the relation and joining the output partially pre-aggregated tuples with the tuples from the inner relation. These techniques however apply to a single binary join at a time, and as we show in Figure 5.7, Join-Agg provides substantial memory benefits than partial pre-aggregation especially when the two are combined and we use pre-aggregation before loading in the data graph.

As previously mentioned, the way we load the data graph into memory in our Join-Agg operator (discussed in Chapter 5) is reminiscent of these data reduction operators since we are pre-aggregating all relations to compute the multiplicity of each edge in the data graph. The creation of *multi-nodes* in the data graph can also

be seen as an even more *effective* form of pre-aggregation. For example in Figure 5.3, looking at relation B, we can see that any standard pre-aggregation operator would reduce the relation to at least 2 tuples with `jc1,jd1` appearing twice whereas we load a single `jc1,jd1` node. Our techniques are comparable with partial pre-aggregation in the case where there are no *branching* relations. As branching relations and multiple group by attributes are included in a complex join, our technique enables computation caching at the level of path-ids which can reduce the number of paths taken in the data graph during Stage 2 of the algorithm. The partial pre-aggregation technique has no means of skipping these paths and may require computing all of the joins associated with those paths potentially multiple times.

Chapter 7: Conclusions

In this chapter we summarize the things we learned from the work in this dissertation. In closing, we briefly discuss the results presented in past chapters, and reflect on the limitations of our approach on building an independent interface layer over database systems for graph analytics.

7.1 Leveraging Graph Representations of Relational Data

Relational databases still remain among the most widely used data management technologies. As their name suggests, RDBMSs contain various relationships between entities within. The importance of graph algorithms (which leverage these data relationships) is becoming increasingly apparent as these algorithms are used more and more ubiquitously. Connections between data are *also* used to compute joins that are crucial for traditional BI SQL reports, specifically for decision support queries, that often require *aggregations* over joins. Due to the iterative nature of graph algorithms, leveraging these relationships effectively as in-memory graphs requires time-consuming manual ETL and both types of ways to leverage data relationships are bound by the bottleneck of large-output joins.

The work presented in this dissertation can be boiled down to attempting

to leverage various in-memory graph representations of data stored in RDBMSs in order to:

- Allow the user efficient access to a certain set of interconnected entities in an RDBMS without them having to worry about changing their data analysis pipelines or storage layout/system choices.
- Efficiently execute aggregate queries over joins in RDBMSs without materializing intermediate results.

We enumerate a set of findings throughout this work, that could have only been discovered by attempting to build a system like GraphGen and investigate the power of similar graph representations.

Large-Output Joins: Combining various relations in an RDBMS via joins in order to form a set of graph edges can result in a space explosion–we call these large-output joins. In order to deal with these joins we store the graph in a condensed representation, and delay their evaluation until the point where it is actually needed (when the graph is traversed). This enables *exploration* of graphs that exist within RDBMSs without (a) having to wait to materialize the entire graph from the start, or (b) having to store the entire graph in memory at any one point in time. The trade-off for this is a relatively small traversal overhead for this representation. Moreover, current state of the art query planners are not very good at accurately predicting the sizes of join results apriori. This can result in long-running queries and manual investigation to figure out why the queries are taking so long.

Duplicates in Condensed Graph Representations: When one wants to as-

sociate two sets of entities in an RDBMS through joins, there are in many cases duplicate associations of those entities inherent in the RDBMS– this duplication is of course also inherent in the original condensed representation initially loaded from the RDBMS. We have developed a suite of techniques for both *structural* deduplication, and *bitmap* deduplication over the original condensed representation, for dealing with these duplicates.

Graph Collections: One interesting way of analyzing graph data is by directly comparing a variety of *distinct* graphs. We found that a simple declarative graph definition language like GraphGenDL is expressive enough to allow users to specify a variety of complex graph collections over their RDBMSs. Extracting these separate graphs in practice however requires executing multiple separate SQL queries against the database. This does not turn out to always be a tenable solution due to the amount of back-and-forth against the database as well as the amount of overlap between the data returned by each query–as collections of co-related graphs often overlap with each other. We found that by rewriting the query in order to tag each element and classify it into its appropriate graph, we can extract such graph collections efficiently with a single SQL query to the RDBMS. Moreover, these language constructs in GraphGenDL can open up users to *what-if* analytics over graphs, enabling them to analyze the different versions of a graph based on the possible value of a specific set of parameters. To the best of our knowledge, GraphGenDL is the first language to tackle the problem of expressing *what-if* analytics over graphs.

Join-Aggregate Query Execution: Currently, RDBMSs are still using classic

query processing techniques that roughly conduct a series of binary joins, materialize the intermediate results and then aggregate them. While pipelining joins might work in some cases, the end join result still needs to be materialized in full before it can be aggregated, otherwise hash-aggregation would need to be used, which is becomes very memory and CPU-intensive as the hash table grows in size. We found that by loading in a data-graph representation of the underlying join (similar to the notion of a factorized query result [113, 115]), we can avoid materializing the full join result while still being able to compute the aggregate result (which can be orders of magnitude smaller).

7.2 Limitations

While we believe the results and algorithms presented in this dissertation advance our understanding of graph querying and graph analytics, they also exhibit some limitations; we discuss a few of them here.

Loading In-Memory Representations: There is a cost to loading in-memory representations, associated with copying over the RDBMS tables into memory and creating edges between them. This process is the same as computing the join without materializing it. In most situations this would not cause an issue, however in cases where graphs are only as big as the underlying data, this process would be a considerable overhead. Nevertheless, this overhead cannot be avoided in any system unless the computation is pushed directly into the database, which is generally not ideal for doing analysis that requires multiple iterative traversals of the graph (given

the data is in an RDBMS and not a native graph store).

Single Machine: On a similar note, our approach makes the assumption that the graph to be analyzed fits in memory (in its condensed form). This assumption is based on the idea that the vast majority of graphs not only can fit on a big memory single machine [94], but it is also usually significantly more efficient to process them on a single machine [135].

Acyclic Queries: For both GraphGen and Join-Agg, we restrict our discussion to acyclic queries and do not delve into how these ideas would apply to cyclic queries.

Large Complex Schemas: While we have bet on Datalog and we believe its ability to express combinations of relations is the most intuitive and terse solution out there, it is not great when one is dealing with *wide* relations. For example, if relation `AuthorPublication` had 10 columns where we were only interested in the authorId (which is the first column) and the publication id which was the *seventh* column, the user would end up needing to write: `AuthorPublication(ID,_,_,_,_,_,pid)`, while being careful to count the number of underscores in-between. This can be ameliorated by adding syntactic sugar to make the process easier, or simply creating easy-to-use Graphical User Interfaces (GUI), that enable the user to simply point and click on their schema, generating the Datalog atom(s) automatically. Moreover, identifying potentially interesting graphs itself may be difficult for large schemas with 100s of tables. We make the assumption that the user that is interested in conducting graph analysis is well versed in the schema, as well as the Entity-Relationship (ER) diagram associated with it (i.e., the natural entities that

exist and how the can relate to each other inside the dataset).

Flavors of SQL: Even though SQL is a *standard* (as per its name), there are still a variety of different (albeit small) differences in syntax across RDBMSs. We use PostgreSQL as the underlying RDBMS in most of our work, and in some cases, our implementation utilizes PostgreSQL-specific syntax. We believe however that the differences between syntax choices are small enough that they can very easily be integrated into the SQL translation process which can output the appropriate, parseable SQL based on the input system.

7.3 Closing Thoughts

Graph analysis is still considered a relatively niche market as it does not constitute a large portion of analytics done in most enterprises. Wherever it *is* used however, it is irreplaceable. Moreover, there have been a slew of graph processing and graph database systems that have been commercialized in the past few years, as well as existing database systems that have added graph processing extensions to their systems. There have also been efforts towards a standardized graph query language [136]. Recent developments in artificial intelligence out of Google have discussed how graph neural networks (that are essentially multigraphs) might be the best way to model a real-world system [137].

The main source of motivation for us was to open up this world of graph algorithms to as many users as possible, and that is what we've attempted to do with this work. Users and developers alike are still in the process of trying to figure

out where graphs fit in the overall data pipeline, and what is the best way to interface with them. There is a plethora of opportunities on the horizon as the world comes to a consensus about these technologies, and we deeply believe the ideas behind GraphGen will have an important role to play in the way users conduct graph analytics in the future.

Chapter 8: Algorithm Pseudocodes

Here, we sketch the pseudo-codes for some of the different algorithms presented in various parts of the thesis:

Algorithm 1 BITMAP-1

```
1: procedure BMP1(graph)
2:     seen ← hashSet()
3:     for each real node rn in graph.vertices() do
4:         for each virtual node vn in rn.getOutNeighbors() do
5:             for each index, real node rn_2 in vn.getOutNeighbors() do
6:                 if rn2 ∉ seen then
7:                     set bit at index of bitmap vn.bitMaps.get(rn)
8:                     seen.add(rn2)
9:         seen.clear()
```

Algorithm 2 BITMAP-2

```
1: procedure BMP2(graph,ordering)
2:     srted ← graph.vertices.sortByDuplication(ordering)
3:     seen ← hashSet()
4:     for each real node rn in srted do
5:         virtSet ← greedySetCover(rn)
6:         for each virtual node v ∈ virtSet do
7:             for each index, real node rn_2 in v.getOutNeighbors() do
8:                 if rn_2 ∉ seen then
9:                     v.getBitmap(rn).setBitAt(index)
10:                    seen.add(rn_2)
11:                else
12:                    chosen ← false
13:                    for each bitmap bmp in v.getBitmaps() do
14:                        if bmp.getBitFor(rn) == 1 then
15:                            chosen ← true
16:                            break
17:                    if !chosen then
18:                        v.removeBitMapFor(rn)
19:                        removeEdge(rn,v)
20:            v.rebuildBitmapIndex()
21:        seen.clear()
```

Algorithm 3 Greedy Virtual Nodes First (DEDUP-1)

```
1: procedure VIRTNODESFIRST1(graph,ordering)
2:     processed ← hashSet()
3:     srtd ← orderVirtualNodes(ordering)
4:     for each virtual node v ∈ srtd do
5:         relevant ← getRelevantVNodes(v)
6:         moreDedup ←true
7:         while moreDedup do
8:             moreDedup ←false
9:             intersections ← getIntersections(v,relevant)
10:            for each s ∈ relevant do
11:                C_i ← intersections.get(s)
12:                if |C_i| > 1 then
13:                    moreDedup ←true
14:                    for each real node rn ∈ C_i do
15:                        R, V, DirectEdges ← maxBenefitRatio(rn)
16:            if R ≠Null then
17:                graph.removeEdge(R, V)
18:                addDirectEdges(R, V, DirectEdges)
19:            processed.add(v)
```

Algorithm 4 Greedy Real Nodes First

```
1: procedure RealNodesFirst(graph,ordering)
2:     V' ← hashSet()
3:     V'' ← hashSet()
4:     X ← hashSet()
5:     srtd ← graph.sortRealNodesByDuplication(ordering)
6:     for each real node rn ∈ srtd do
7:         initialize(V'',rn)
8:         while V'' ≠ ∅ do
9:             maxBenefitVNode ← getMaxBenefitCostRatioVNode()
10:            if maxBenefitVNode ≠Null then
11:                V ← hashSet()
12:                for real node rn_2 ∈ maxBenefitVNode.getOutNeighbors() do
13:                    V.add(rn_2)
14:                    graph.removeEdge(rn,rn_2)
15:                V'.add(maxBenefitVNode)
16:                V''.remove(maxBenefitVNode)
17:                VCapX ← V ∩ X
18:                for each pair a, b of a ∈ V − VCapX and b in VCapX do
19:                    if !existsEdge(a, b) then
20:                        graph.addEdge(a, b)
21:                for each real node s ∈ V do
22:                    if s.equals(rn) then
23:                        X.add(s)
24:                for each real node s in VCapX do
25:                    graph.removeEdge(s,maxBVNode)
26:            else
27:                for real node vn in V'' do
28:                    removeEdge(rn,vn)
29:        X.clear()
```

Algorithm 5 Greedy Virtual Nodes First (DEDUP-2)

```
1: procedure VirtNodesFirst2(graph,ordering)
2:     srtd ←orderVNodes(ordering)
3:     for each virtual node vn in srtd do
4:         constraints ← hashMap()
5:         ResolveVirtualNode(v,constraints)
6: procedure ResolveVirtualNode(v,constraints)
7:     relevant ←getRelevantVNodes(v)
8:     HV ←highestOverlap(v, relevant)
9:     W1 ←intersect(HV, v)
10:    w1 ←createVirtNode(W1)
11:    if W1 ≠ ∅ then
12:        W2 ← HV − W1
13:        if W2 ≠ ∅ then
14:            w2 ←createVirtNode(W2)
15:            addVirtualEdge(w1, w2)
16:        for each virtual node vn ∈ HV.getVirtualNeighbors() do
17:            addVirtualEdge(w1, vn)
18:            addVirtualEdge(w2, vn)
19:        W3' ← v − W1
20:        W3 ← W3'−NUnion(HV)
21:        if W3 ≠ ∅ then
22:            w3 ←createVirtNode(W3)
23:            addConstraint(constraints, w3, w1)
24:        W4 ← W3' − W3
25:        if W4 ≠ ∅ then
26:            w4 ←createVirtNode(W4)
27:            if W3 ≠ ∅ then
28:                addConstraint(constraints, w4, w3)
29:        if v exists in constraints then
30:            splitConstraintsFor(constraints, v)
31:        if w4 ≠Null then
32:            ResolveVirtualNode(w4, constraints)
33:        if w3 ≠Null then
34:            ResolveVirtualNode(w3, constraints)
35:    graph.removeVirtualNode(v)
36:    if initialCall then
37:        graph.addVirtualedgesIn(constraints)
```

Bibliography

[1] Amit Chaudhry. The graph technology buyer's guide. `https://neo4j.com/whitepapers/graph-technology-buyers-guide/`.

[2] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. The case against specialized graph analytics engines. In *Proceedings of the Conference on Innovative Data Systems Research*. ACM, 2015.

[3] Kangfei Zhao and Jeffrey Xu Yu. All-in-one: Graph processing in rdbmss revisited. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 1165–1180. ACM, 2017.

[4] Alekh Jindal, Samuel Madden, Malu Castellanos, and Meichun Hsu. Graph analytics using the Vertica relational database. In *Proceedings of the International Conference on Big Data*. IEEE, 2015.

[5] Knowledge base of relational and NoSQL database management systems. `https://db-engines.com/en/ranking`.

[6] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[7] Pablo Porras Millan. Visualization and analysis of biological networks. In *Silico Systems Biology*, pages 63–88. Springer, 2013.

[8] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[9] Michael Stonebraker and Joseph Hellerstein. What goes around comes around. *Readings in Database Systems*, 4:1724–1735, 2005.

[10] Neo4j: A scalable native graph database. `https://neo4j.com/product/`.

[11] Amazon. Amazon neptune fast, reliable graph database built for the cloud. `https://aws.amazon.com/neptune/`.

[12] OrientDB - the world's first distributed multi-model NoSQL database with a graph database engine. `http://orientdb.com/orientdb/`.

[13] Jayavel Shanmugasundaram Kristin Tufte Gang He and Chun Zhang David DeWitt Jeffrey Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the International Conference on Very Large Data Bases*. VLDB Endowment, 1999.

[14] Mihaela A Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient RDF store over a relational database. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 121–132. ACM, 2013.

[15] Daniel J Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the International Conference on Very Large Data Bases*, pages 411–422. VLDB Endowment, 2007.

[16] Roshan Punnoose, Adina Crainiceanu, and David Rapp. Rya: a scalable RDF triple store for the clouds. In *Proceedings of the International Workshop on Cloud Intelligence*, page 4. ACM, 2012.

[17] AllegroGraph. A high-performance persistent RDF store. `http://franz.com/agraph/allegrograph/`.

[18] Apache Jena. SDB - persistent triple stores using relational databases. `https://jena.apache.org/documentation/sdb/index.html`.

[19] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. SQLGraph: an efficient relational-based property graph store. In *Proceedings of the SIGMOD International Conference on Management of Data*. ACM, 2015.

[20] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Michael Stonebraker. Vertexica: your relational friend for graph analytics! *Proceedings of the International Conference on Very Large Data Bases*, 7(13):1669–1672, 2014.

[21] Titan: A distributed graph database. `http://titan.thinkaurelius.com/`.

[22] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[23] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 135–146. ACM, 2010.

[24] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, and James Cheng. Big graph analytics systems (tutorial). In *Proceedings of the SIGMOD International Conference on Management of Data*. ACM, 2016.

[25] Apache. Giraph. `http://giraph.apache.org/`.

[26] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. In *Proceedings of the International Conference on Very Large Data Bases*, volume 5, pages 716–727. VLDB Endowment, 2012.

[27] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Symposium on Operating Systems Design and Implementation*, volume 12, page 2. USENIX, 2012.

[28] Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. Converting relational to graph databases. In *Proceedings of the International Workshop on Graph Data Management Experiences and Systems*, page 1. ACM, 2013.

[29] Sangkeun Lee, Byung H Park, Seung-Hwan Lim, and Mallikarjun Shankar. Table2Graph: A scalable graph construction from relational tables using map-reduce. In *Proceedings of the International Conference on Big Data Computing Service and Applications*. IEEE, 2015.

[30] Nilesh Jain, Guangdeng Liao, and Theodore L Willke. Graphbuilder: scalable graph etl framework. In *Proceedings of the International Workshop on Graph Data Management Experiences and Systems*, page 4. ACM, 2013.

[31] DBLP dataset. `https://dblp.uni-trier.de/faq/How+can+I+download+the+whole+dblp+dataset`.

[32] `http://www.tpc.org/tpch/`.

[33] University of Toronto. Computer Systems Research Group and MH Graham. *On the universal relation*. 1980.

[34] Clement Tak Yu and Meral Z Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979.*, pages 306–312. IEEE, 1979.

[35] Per-Åke Larson. Data reduction by partial preaggregation. In *Proceedings of the International Conference on Data Engineering*. IEEE, 2002.

[36] Konstantinos Xirogiannopoulos, Udayan Khurana, and Amol Deshpande. Graphgen: exploring interesting graphs in relational data. In *Proceedings of the International Conference on Very Large Data Bases*, volume 8, pages 2032–2035. VLDB Endowment, 2015.

[37] Konstantinos Xirogiannopoulos and Amol Deshpande. Extracting and analyzing hidden graphs from relational databases. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 897–912. ACM, 2017.

[38] Konstantinos Xirogiannopoulos, Virinchi Srinivas, and Amol Deshpande. Graphgen: Adaptive graph processing using relational databases. In *Proceedings of the International Workshop on Graph Data-management Experiences & Systems*, page 9. ACM, 2017.

[39] Konstantinos Xirogiannopoulos and Amol Deshpande. Memory-efficient group-by aggregates over multi-way joins. *arXiv preprint arXiv:1906.05745*, 2019.

[40] Terrence Parr. Another tool for language recognition (antlr). `http://www.antlr.org/`.

[41] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys*, 48(2):25, 2015.

[42] Todd J Green, Molham Aref, and Grigoris Karvounarakis. Logicblox, platform and language: A tutorial. In *Proceedings of the International Datalog 2.0 Workshop*, pages 1–8. Springer, 2012.

[43] Jiwon Seo, Stephen Guo, and Monica S. Lam. SociaLite: Datalog extensions for efficient social network analysis. In *Proceedings of the International Conference on Data Engineering*. IEEE, 2013.

[44] Jun Gao, Jiashuai Zhou, Chang Zhou, and Jeffrey Xu Yu. GLog: a high level graph analysis system using mapreduce. In *Proceedings of the International Conference on Data Engineering*. IEEE, 2014.

[45] Yousef Saad. *Iterative Methods for Sparse Linear Systems*, volume 82. siam, 2003.

[46] Apache TinkerPop. Apache TinkerPop is a graph computing framework for both graph databases (OLTP) and graph analytic systems (OLAP). `http://tinkerpop.apache.org/`.

[47] The Gremlin graph traversal machine and language. `https://tinkerpop.apache.org/gremlin.html`.

[48] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

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

[50] Chinmay Karande, Kumar Chellapilla, and Reid Andersen. Speeding up algorithms on compressed web graphs. *Internet Mathematics*, 2009.

[51] Tom as Feder and Rajeev Motwani. Clique partitions, graph compression, and speeding-up algorithms. In *Proceedings of the Symposium on the Theory of Computing*, pages 123–133. ACM, 1991.

[52] Aidan Hogan, Marcelo Arenas, Alejandro Mallea, and Axel Polleres. Everything you always wanted to know about blank nodes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 27:42–69, 2014.

[53] Yodsawalai Chodpathumwan, Amirhossein Aleyasen, Arash Termehchy, and Yizhou Sun. Universal-DB: towards representation independent graph analytics. *Proceedings of the International Conference on Very Large Data Bases*, 8(12), 2015.

[54] Sebastian Maneth and Fabian Peternek. A survey on methods and systems for graph compression. *arXiv preprint arXiv:1504.00616*, 2015.

[55] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1), 2002.

[56] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *Transactions on Knowledge Discovery from Data*, 2007.

[57] Termeh Shafie. A multigraph approach to social network analysis. *Journal of Social Structure*, 16:0_1, 2015.

[58] Lars Backstrom and Jure Leskovec. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the international conference on Web Search and Data Mining*, pages 635–644. ACM, 2011.

[59] Jae-wook Ahn, Catherine Plaisant, and Ben Shneiderman. A task taxonomy for network evolution analysis. *Transactions on Visualization and Computer Graphics*, 20(3):365–376, 2013.

[60] Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. In *Proceedings of the International Conference on Knowledge Discovery and Data mining*, pages 611–617. ACM, 2006.

[61] Rakshit Trivedi, Bunyamin Sisman, Jun Ma, Christos Faloutsos, Hongyuan Zha, and Xin Luna Dong. Linknbed: Multi-graph representation learning with entity linkage. *arXiv preprint arXiv:1807.08447*, 2018.

[62] Mark Heimann, Haoming Shen, Tara Safavi, and Danai Koutra. Regal: Representation learning-based graph alignment. In *Proceedings of the International Conference on Information and Knowledge Management*, pages 117–126. ACM, 2018.

[63] Meng Wang, Xian-Sheng Hua, Richang Hong, Jinhui Tang, Guo-Jun Qi, and Yan Song. Unified video annotation via multigraph learning. *Transactions on Circuits and Systems for Video Technology*, 19(5):733–746, 2009.

[64] Manuel Then, Timo Kersten, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. In *Proceedings of the International Conference on Very Large Data Bases*, volume 10, pages 877–888. VLDB Endowment, 2017.

[65] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the European Conference on Computer Systems*, page 1. ACM, 2014.

[66] Wenlei Xie, Yuanyuan Tian, Yannis Sismanis, Andrey Balmin, and Peter J Haas. Dynamic interaction graphs with probabilistic edge decay. In *Proceedings of the International Conference on Data Engineering*, pages 1143–1154. IEEE, 2015.

[67] Udayan Khurana and Amol Deshpande. Efficient snapshot retrieval over historical graph data. In *Proceedings of the International Conference on Data Engineering*, pages 997–1008. IEEE, 2013.

[68] Tarun Kathuria and S Sudarshan. Efficient and provable multi-query optimization. In *Proceedings of the SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 53–67. ACM, 2017.

[69] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the SIGMOD International Conference on Management of Data*. ACM, 2000.

[70] Amol Deshpande. An initial study of overheads of eddies. *SIGMOD Record*, 33(1):44–49, 2004.

[71] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms:[extended abstract]. In *Proceedings of the SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 2012.

[72] Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *Proceedings of the International Conference on Database Theory*, 2014.

[73] Brett Walenz, Sudeepa Roy, and Jun Yang. Optimizing iceberg queries with complex joins. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 1243–1258. ACM, 2017.

[74] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. In *Proceedings of the SIGMOD International Conference on Management of Data*. ACM, 2016.

[75] Dan Olteanu and Jakub Závodnỳ. Factorised representations of query results: size bounds and readability. In *Proceedings of the International Conference on Database Theory*, pages 285–298. ACM, 2012.

[76] Ahmet Kara and Dan Olteanu. Covers of query results. In *Proceedings of the International Conference on Database Theory*, 2018.

[77] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: Questions and answers. In *Proceedings of the SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 2016.

[78] Robert E Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on computing*, 13(3):566–579, 1984.

[79] UCI Machine Learning Repository. Online retail data set. `https://archive.ics.uci.edu/ml/datasets/Online+Retail`.

[80] Oliver Schulte Jan Motl. Relational dataset repository. `https://relational.fit.cvut.cz/`.

[81] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1, 2008.

[82] Paolo Atzeni, Paolo Cappellari, Riccardo Torlone, Philip A Bernstein, and Giorgio Gianforme. Model-independent schema translation. *The International Journal on Very Large Data Bases*, 17(6):1347–1370, 2008.

[83] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semist- ructured data with STORED. In *SIGMOD Record*. ACM, 1999.

[84] Fan Yang, Jinfeng Li, and James Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *Proceedings of the International Conference on Very Large Data Bases*, 9(5):420–431, 2016.

[85] Yingyi Bu, Vinayak Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. In *Proceedings of the International Conference on Very Large Data Bases*. VLDB Endowment, 2014.

[86] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. Graphframes: an integrated api for mixing graph and relational queries. In *Proceedings of the International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2016.

[87] Hongbin Ma, Bin Shao, Yanghua Xiao, Liang Jeff Chen, and Haixun Wang. G-SQL: fast query processing via graph exploration. In *Proceedings of the International Conference on Very Large Data Bases*, volume 9. VLDB Endowment, 2016.

[88] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. Fast in-memory SQL analytics on typed graphs. In *Proceedings of the International Conference on Very Large Data Bases*, volume 10. VLDB Endowment, 2016.

[89] Mohamed S Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G Aref, and Mohammad Sadoghi. Grfusion: Graphs as first-class citizens in main-memory relational database systems. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 1789–1792. ACM, 2018.

[90] Ruby Y Tahboub, Xilun Wu, Grégory M Essertel, and Tiark Rompf. Towards compiling graph queries in relational engines. In *Proceedings of the SIGPLAN International Symposium on Database Programming Languages*, pages 30–41. ACM, 2019.

[91] David Simmen, Karl Schnaitter, Jeff Davis, Yingjie He, Sangeet Lohariwala, Ajay Mysore, Vinayak Shenoi, Mingfeng Tan, and Yu Xiao. Large-scale Graph Analytics in Aster 6: Bringing Context to Big Data Discovery. In *Proceedings of the International Conference on Very Large Data Bases*, volume 7. VLDB Endowment, 2014.

[92] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The graph story of the SAP HANA database. In Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mitschang, Theo Härder, and Veit Köppen, editors, *Datenbanksysteme für Business, Technologie und Web (BTW) 2037*, pages 403–420, Bonn, 2013. Gesellschaft für Informatik e.V.

[93] Yuanyuan Tian, Sui Jun Tong, Mir Hamid Pirahesh, Wen Sun, En Liang Xu, and Wei Zhao. Synergistic graph and SQL analytics inside IBM Db2. In *Proceedings of the International Conference on Very Large Data Bases*, volume 12. VLDB Endowment, 2019.

[94] Yonathan Perez, Rok Sosič, Arijit Banerjee, Rohan Puttagunta, Martin Raison, Pararth Shah, and Jure Leskovec. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the SIGMOD International Conference on Management of Data*. ACM, 2015.

[95] Jure Leskovec and Rok Sosič. SNAP: A general purpose network analysis and graph mining library in C++. `http://snap.stanford.edu/snap`, June 2014.

[96] Paolo Boldi and Sebastiano Vigna. The webgraph framework compression techniques. In *Proceedings of the World Wide Web Conference*, pages 595–602, 2004.

[97] Alberto Apostolico and Guido Drovandi. Graph compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.

[98] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. k2-trees for compact web graph representation. In *Intl. Symposium on String Processing and Information Retrieval*, 2009.

[99] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *Proceedings of the Data Compression Conference*, pages 403–412, 2015.

[100] Yasuhito Asano, Yuya Miyawaki, and Takao Nishizeki. Efficient compression of web graphs. In *International Computing and Combinatorics Conference*, pages 1–11, 2008.

[101] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *Proceedings of the International Conference on Very Large Data Bases*. VLDB Endowment, 2003.

[102] Christoph Koch. Processing queries on tree-structured data efficiently. In *Proceedings of the SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 2006.

[103] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query preserving graph compression. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 157–168. ACM, 2012.

[104] Sebastian Maneth and Fabian Peternek. Compressing graphs by grammars. In *Proceedings of the International Conference on Data Engineering*. IEEE, 2016.

[105] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Many-query join: efficient shared execution of relational joins on modern hardware. pages 1–24. Springer, 2017.

[106] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD Record*, volume 29, pages 249–260. ACM, 2000.

[107] Nikos Tsikoudis, Liuba Shrira, and Sara Cohen. Rql: Retrospective computations over snapshot sets. In *International Conference on Extending Database Technology*, 2018.

[108] Dan Olteanu and Jakub Závodnỳ. Size bounds for factorised representations of query results. *Transactions on Database Systems*, 40(1):2, 2015.

[109] Nurzhan Bakibayev, Tomáš Kočiskỳ, Dan Olteanu, and Jakub Závodnỳ. Aggregation and ordering in factorised databases. *Proceedings of the International Conference on Very Large Data Bases*, 6(14):1990–2001, 2013.

[110] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. AC/DC: in-database learning thunderstruck. In *Proceedings of the Workshop on Data Management for End-To-End Machine Learning*, page 8. ACM, 2018.

[111] Mahmoud Abo Khamis, Ryan R Curtin, Benjamin Moseley, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. On functional aggregate queries with additive inequalities. *arXiv preprint arXiv:1812.09526*, 2018.

[112] Mahmoud Abo Khamis, Hung Q Ngo, Dan Olteanu, and Dan Suciu. Boolean tensor decomposition for conjunctive queries with negation. In *Proceedings of the International Conference on Database Theory*. IEEE, 2019.

[113] Maximilian Schleich, Dan Olteanu, and H Ngo. A layered aggregate engine for analytics workloads. 2019.

[114] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 3–18. ACM, 2016.

[115] Dan Olteanu and Maximilian Schleich. Factorized databases. *SIGMOD Record*, 45(2), 2016.

[116] Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proceedings of the International Conference on Database Theory*, pages 96–106, 2014.

[117] Paraschos Koutris, Paul Beame, and Dan Suciu. Worst-case optimal algorithms for parallel query processing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 48. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[118] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2014.

[119] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the International Conference on Very Large Data Bases*, pages 82–94. VLDB Endowment, 1981.

[120] Manas Joglekar, Rohan Puttagunta, and Christopher Ré. Aggregations over generalized hypertree decompositions. *arXiv preprint arXiv:1508.07532*, 2015.

[121] Manas R Joglekar, Rohan Puttagunta, and Christopher Ré. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 2016.

[122] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. FAQ: questions asked frequently. In *Proceedings of the SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 13–28. ACM, 2016.

[123] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D Ullman. Computing iceberg queries efficiently. In *Proceedings of the International Conference on Very Large Data Bases*. Stanford InfoLab, 1999.

[124] Kevin Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cube. In *SIGMOD Record*, pages 359–370. ACM, 1999.

[125] Jiawei Han, Jian Pei, Guozhu Dong, and Ke Wang. Efficient computation of iceberg cubes with complex measures. In *SIGMOD Record*. ACM, 2001.

[126] Dong Xin, Jiawei Han, Xiaolei Li, and Benjamin W Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *The International Journal on Very Large Data Bases*, 2003.

[127] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. String similarity joins: An experimental evaluation. *Proceedings of the International Conference on Very Large Data Bases*, 7(8):625–636, 2014.

[128] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. Efficient similarity joins for near-duplicate detection. *Transactions on Database Systems*, 36(3):15, 2011.

[129] Jiannan Wang, Jianhua Feng, and Guoliang Li. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. In *Proceedings of the International Conference on Very Large Data Bases*, volume 3, pages 1219–1230. VLDB Endowment, 2010.

[130] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. Pass-join: A partition-based method for similarity joins. In *Proceedings of the International Conference on Very Large Data Bases*, volume 5, pages 253–264. VLDB Endowment, 2011.

[131] W Yan and Per-Åke Larson. Interchanging the order of grouping and join. Technical report, Technical Report CS 95-09, Dept. of Computer Science, University of Waterloo, Canada, 1995.

[132] Weipeng P. Yan and Per-Åke Larson. Eager aggregation and lazy aggregation. In *Proceedings of the International Conference on Very Large Data Bases*, pages 345–357. Morgan Kaufmann Publishers Inc., 1995.

[133] Weipeng P Yan and Per-Åke Larson. Performing group-by before join. In *Proceedings of the International Conference on Data Engineering*. IEEE, 1994.

[134] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *Proceedings of the International Conference on Very Large Data Bases*, volume 94, pages 354–366, 1994.

[135] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! But at what COST? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2015.

[136] Neo4j. New query language for graph databases to become international standard. `https://neo4j.com/press-releases/query-language-graph-databases-international-standard/`.

[137] Peter Battaglia, Jessica Blake Chandler Hamrick, Victor Bapst, Alvaro Sanchez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andy Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Jayne Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.