Systems for Graph Extraction from Tabular Data

by

Nafisa Anzum

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Connections amongst real-world entities provide significant insights for numerous real-life applications in social networks, semantic web, road maps, finance, among others. Graphs are perhaps the most natural way to model such connections in application data. However, in many enterprises, an application data is still primarily stored in an RDBMS in a tabular format and users extract graphs out of an RDBMS and store them in specialized graph processing systems. As a result, many users face two major challenges before conducting any graph analysis. First, extracting graphs from an RDBMS requires building an ETL pipeline, which can require a significant amount of time. Second, keeping the extracted graph in the graph processing system, such as a graph database management system (GDBMS), in sync with the original data in the RDBMS requires developing additional non-trivial synchronization code. In this thesis, we study and address these two challenges and present two software systems, *GraphWrangler* and *R2GSync*, that we have developed to solve these challenges. GraphWrangler is an interactive system that streamlines the ETL pipeline. Users connect to an RDBMS using GraphWrangler and with several simple interactions, such as dragging and dropping of rows and columns and drawing edges on the screen, they describe table-to-graph mappings. This way, users can describe the graphs they would like to extract without writing any custom scripts. In addition, GraphWrangler allows user to immediately visualize their tables in the form of a graph. Our second system, R2GSync, uses the mappings of an extracted graph and maintains a consistent, i.e., in sync, copy of this graph in a GDBMS as updates happen to the original RDBMS from which the graph was extracted. Querying the extracted graph inside the GDBMS requires a new querying functionality inside the GDBMS that we call *edge views*. We describe our implementation of edge views and several optimizations to make queries that contain edge views more efficient.

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor, Professor Semih Salihoglu, for his constant support and guidance, especially during a very difficult time of my life. This thesis would not have been possible without his relentless motivation and inspiration. Semih introduced me to the world of data systems and has been consistently present every step of the way in my learning process. He has taught me everything I know about good research work, presentation, and academic writing. His research group has a collaborative learning environment and he ensures a good vibe among his students. I am blessed to be a part of his research group and to have the opportunity to continue my learning under his care for my Ph.D.

I want to thank my friend and colleague, Amine Mhedhbi, who selflessly helped me whenever I needed and taught me a lot of things about good research and academic writing. His valuable insights and feedback played a big part in bringing my thesis to fruition. Furthermore, I want to thank my friend, Aida Sheshbolouki for all the help and support she provided in the last two years.

I also want to express my appreciation to my thesis readers, Professor Daniel Vogel and Professor Xi He, for taking out valuable time from their busy schedules to read my thesis and provide their valuable feedback.

I would like to thank my parents, Nazmum Nahar and Matiur Rahman, for their unconditional love and support. I am eternally grateful to them for teaching me the importance of good education, for putting me through the best schools throughout my life, and always believing in me to go further in life. Furthermore, I want to thank my soul sister, Johra Moosa, for giving me constant emotional support and making my life at Uwaterloo comforting.

Last but not least, I would like to thank my husband, Mohammad Rashidujjaman Rifat, for always being there for me and loving me unconditionally. His constant support and motivation kept me going and helped me to have a positive mindset in every circumstance. He has been the biggest source of happiness in my life, which motivates me every day to work harder to achieve my goals.

Dedication

This is dedicated to my parents and my husband.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are perhaps the most natural data structure to represent interconnections among entities in app data from many domains such as social networks, road networks, semantic webs, and communication networks. In *graph database management systems (GDBMSs)*, application data is modeled by nodes representing the entities and by directed edges representing the relationships between entities. GDBMSs manage data for a very wide range of analytical applications [33]. For example, Twitter searches for diamonds in their follower network for recommendations [14], clique-like structures in social networks indicate communities [30], and cyclic patterns in transaction networks indicate fraudulent activity [32]. For such applications, users rely on GDBMSs that are optimized for detecting complex subgraph patterns [26].

Despite the popularity of graph data analytics among users, GDBMSs are rarely the main system of record in enterprises. In a user survey, Sahu et. al., [33] revealed that enterprises usually maintain two separate databases, one for transactional purpose, which is the main system of record, and another for analytical applications, which is read-only. GDBMSs are typically secondary systems supporting read-heavy analytical applications such as fraud detection or recommendations, whereas RDBMSs maintain the transactional data. Therefore, the graph data stored in GDBMSs are often replicated and extracted from RDBMSs [33].

Keeping replicas of relational data in a graph format has two challenges for enterprises that motivate the research in this thesis. First, since data is not readily available in the graph format, most graph users go through a cumbersome *extraction, transformation, load (ETL)* pipeline. Second, maintaining the GDBMS alongside the primary data storage requires keeping two concurrent systems in sync. In this thesis, we study and

address these two challenges. Specially we develop two software artifacts: (i) GraphWrangler that streamlines the ETL pipeline by allowing users interactively extract graphs out of relational tables, and (ii) R2GSync that keeps the transformed graph in the GDBMS synchronized with the source RDBMS. We also propose several modifications and optimizations to GDBMSs to support queries over the graphs extracted from RDBMS through R2GSync.

In the rest of this introductory chapter, we first give an overview of the functionalities of our GraphWrangler system in Section 1.1. Next, we briefly discuss the RDBMS to GDBMS synchronization problem and give an overview of our R2GSync system . Finally, we outline the rest of the thesis in Section 1.3.

1.1 GraphWrangler: Streamlining Graph Visualization and ETL

A recent survey [33] of users of graph technologies [33] revealed two interesting aspects of how graph technology is situated in the software stack of enterprises:

1. The primary copy of the data stored in graph-processing software, such as a GDBMS, an RDF system, or a graph visualization software, is often stored in an RDBMS. To get the data into the GDBMS, users first extract the graph data from relational tables, transform the data into a graph structure, and finally, load the transformed data into the graph software. Developing this ETL pipeline is a time-consuming, resource-intensive, and costly task. Usually, users write scripts to extract the desired records to represent the set of entities and relationships for their applications from relational tables. These records are then transformed into a graph structure, before being loaded into a graph software. Often users go through a cycle of writing script to extract a specific graph, manual debugging to ensure the correctness of the script, visualizing a sample of the extracted graph to ensure the graph structure that is most appropriate for their application. This whole process may take several iterations, making it troublesome and time-consuming.

2. Many users visualize their graphs, i.e., graphs are a form of visualization for relational data.

To streamline this ETL process and allow users to get a graph view of the tabular data they store in the RDBMS, we developed an interactive system named *GraphWrangler*

(GW). GW lets users directly connect to their RDBMS and provide a tabular view of their data. Through visual interactions, such as dragging and dropping of rows or columns, users transform the tables into a set of nodes and edges. Users' actions are internally expressed as rules that transform tabular data into a node, edge, or a property on a node or an edge. These rules are expressed in the system's *data transformation language* which is a subset of SQL that consists of select, join, and aggregation queries. Within GW's data transformation language, users can express different types of node and edges. Each node and edge type correspond to the result of an SQL query.

Actions that create nodes and properties map to a single transformation rule that correspond to aggregations from a single table. For creating edges, GW adopts the *Predictive Interaction framework* [15]: users draw an edge between two nodes on the UI and the system makes predictions about the rule connecting the nodes. For example, two nodes representing customers Alice and Bob could be connected because they bought the same item or they live in the same city. Internally, these rules correspond to different ways to join the tuples that Alice and Bob were created from. These rules are presented to the user in a human-readable format and the user selects one of the rules.

After defining the nodes and edges through the interactive system, GW auto-generates a script that extracts the whole graph from the RDBMS. GW currently supports wrangling graphs out of MySQL and generates scripts to import graphs to GraphflowDB [21], a prototype in-memory graph database that is developed at the University of Waterloo. Thus GW solves the problem of complex ETL pipeline providing two main objectives: (i) immediate graph view on tabular data; and (ii) easy and streamlined way of transforming data into GDBMS for advanced analytics.

1.2 R2GSync: RDBMS-GDBMS Synchronization

The script that is generated by GW is useful for bulk extraction and loading of a graph database from a relational database. One-time data loading may be enough for applications that need to process snapshots of the database in relatively infrequently intervals. Many applications, however, require keeping a fresh copy of the relational data in a graph processing system. A straightforward technique for keeping two systems in sync is to periodically repeat the ETL process and load fresh snapshots of the graph into the GDBMS. However this is not adequate for many applications that require a snapshot after every transaction. This is especially the case for applications in fraud detection where detecting fraudulent patterns immediately as transactions occur is critical. For example, an e-commerce website's fraud detection application could require searching for a clique of

users buying similar items immediately after the last transaction forming the clique is committed, to ensure that these transactions are aborted quickly.

In the next part of this thesis, we study the problem of keeping an RDBMS, which is the main transactional data source, in sync with a GDBMS that stores and serves a graph data that is extracted from the RDBMS in a setting where the graph copy needs to be fresh very frequently, e.g., upon each update to the RDBMS. We first describe three possible designs for a system that keeps an RDBMS and GDBMS in sync, the pros and cons of each design, and the node and edge types each design can support. We then describe our implementation of one of these designs that we call *R2GSync*.

R2GSync supports a subset of, but perhaps the most commonly used, node types that GW supports. These are aggregation-free node types where each row of a table corresponds to a node. For edges that are results of multi-hop joins between two nodes, R2GSync supports a set of "raw" nodes and edges that may not be visible to the application developers but are maintained to avoid incrementally maintaining results of these joins as edges in the GDBMS. Instead of querying raw edges, the application developer queries *edge views*, which are virtual edges that correspond to the edge type expressed in the data transformation language of GW. We then describe our implementation of edge views inside GraphflowDB and two optimization opportunities that GDBMSs have when evaluating queries over edge views: (i) pushing down distinct aggregation operations; and (ii) pushing down early intersection operators for cyclic queries that involve the intersections of two edge views. We experimentally demonstrate that query performance can improve significantly through these two optimizations.

1.3 Thesis Outline

The outline of this thesis is as follows:

- In chapter 2, we investigate the graph ETL pipeline and describe our proposed interactive tool, GraphWrangler, which aims to streamline this cumbersome process.
- Chapter 3 discusses the different approaches for incremental maintenance of the GDBMS and presents the implementation details of the R2GSync system for this purpose.
- In chapters 4, we discuss the related work.
- Chapters 5 discusses conclusion and future work.

Chapter 2

GraphWrangler

In this chapter, we explain the functionalities and technical details of our interactive data transformation tool, GraphWrangler (GW). We start in Section 2.1 by providing an overview of the main functionalities of GW. We describe GW's predictive interaction framework in Section 2.2. GW adopts several strategies to optimize the interaction speed, while interacting with large dataset, which is explained in Section 2.3. Finally, in Section 2.4, we describe the implementation of the GW system in detail.

Throughout, we assume a database consisting of relations $R_1, ..., R_n$, and $cols(R_i)$ are the columns in R_i's schema. Our running examples use a database of three relations with the following schemas: Product(PID, PName), Customer(CID, CName, City), and Order(OID, CID, PID, Amount). The Product relation contains all the products of the system, where each product is uniquely identified by the primary key, PID. The information of each customer is stored in the Customer relation having CID as the primary key of the relation, CName expressing the name of the customer, and City indicating a place, where the customer lives in. Finally, the Order relation holds every purchase information made by each customer. Here, PID and CID in the Order table are foreign keys of Product and Customer tables respectively.

2.1 System Functionalities

GW helps users interactively extract graphs out of the relational tables. Adopting the terminology of "wrangling" [20], used for extracting tables from other sources, we refer to this ETL processes as *graph wrangling*. Figure 2.1 shows the main interface of GW.

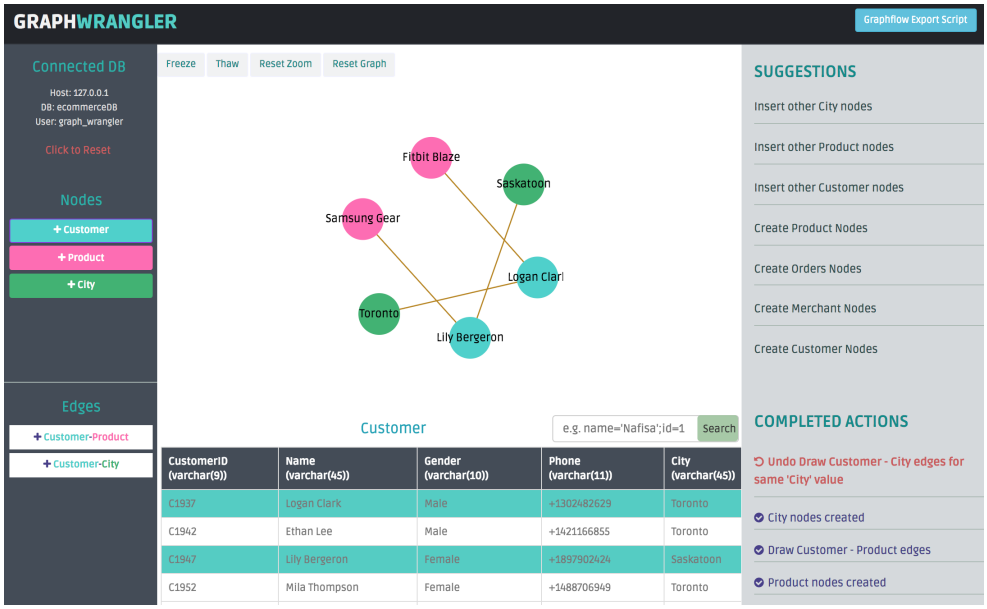


Figure 2.1: Main interface of GraphWrangler

Wrangled graphs in GW consist of a set of nodes, edges, and key-value properties on nodes and edges. The user starts interacting with the system by providing the authentication information of their RDBMS and a specific database name from which the user is interested in extracting graphs. GW then connects to that database and collects information about existing tables, their column names, data types of each column, the primary key of each table, and all of the foreign key relations. This information later used to guide users throughout the wrangling process.

GW provides three major functionalities: 1) wrangling nodes, 2) wrangling edges, and 3) wrangling node and edge properties, which in combination let users define a graph from the relational tables. Each functionality is associated with a set of interactions, which maps to GW's data transformation language. Figure 2.2 shows the functional overview of GW. GW also lets users visualize and explore a portion of their defined graphs on top of relational tables as shown in Figure 2.1. Finally, GW auto-generates a script that extracts the entire graph from the relational tables and transforms them into a graph structure.

2.1.1 Wrangling Nodes

Users can create a node by highlighting one or more cells of a single row (possibly the entire row) in a single table and dragging and dropping the cells on GW's node-link panel.

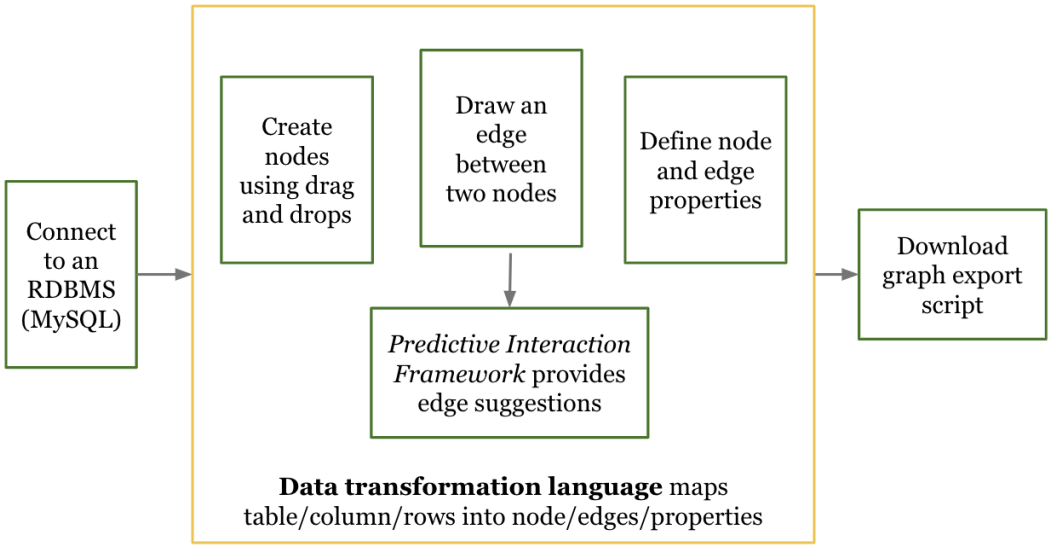


Figure 2.2: GraphWrangler overview

This action draws a single node v on the panel, which is internally associated with three pieces of information:

- *Node Type*: Each node v is of a particular *node type* NT_k, which is a pair $(R_i, cols(NT_k) \subseteq cols(R_i))$ indicating the R_i and R_i's columns that v was wrangled from. We use $v.rel$ and $v.cols$ to refer to R_i and $cols(NT_k)$, respectively.
- *Values*: This is a tuple $v.vals = (val_1, ..., val_t)$, where $t = |v.cols|$, indicating the values of $v.cols$ identifying v.
- *Lineage:* v's lineage $v.lin$ is the set of tuples from R_i with the same values as v for $v.cols$, i.e., $v.lin = \sigma_{v.cols=v.vals} R_i$.

Example 2.1: Suppose the user drag and drops an entire `Customer` *row* t_5=(`C5`, `Alice`, `Waterloo`) *to the panel. GW interprets this as the entire row of the* `Customer` *table representing a node* v*, so* v*'s type is* NT_1=(`Customer`, {C_{ID}, C_{Name}, `City`}), $v.vals$= (`C5`, `Alice`, `Waterloo`), *and* $v.lin$=$\{t_5\}$.

Example 2.2: If the user drag and drops only the `Waterloo` *cell in* t_5*, this is interpreted as each unique* `City` *value being a node. So* v*'s type is* NT_2=(`Customer`,{`City`}),

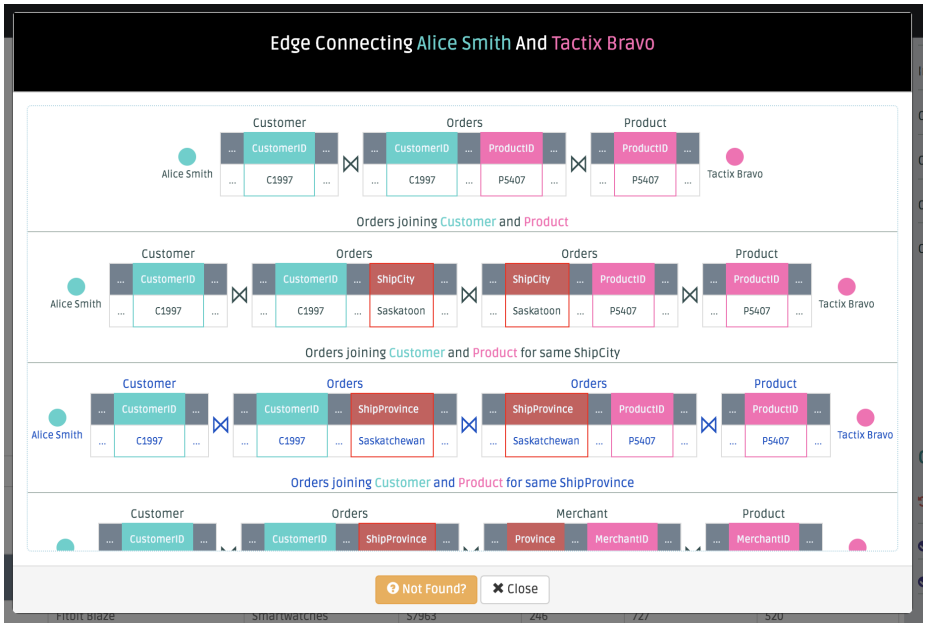


Figure 2.3: Edge type predictions.

$v.vals$=(`Waterloo`), *and* $v.lin$=$\sigma_{City=Waterloo}$`Customer`, *which includes* t_5 *as well as the other tuples whose* `City` *values equal Waterloo.*

Node types are given a human readable name by the users, e.g., "Customer" or "City" as shown in Figure 2.1. If the number of nodes of a particular type are small, users can click a button and extract all nodes of this type and display them in the panel. For example, creating all NT_2 City nodes could add `Kitchener`, `Toronto`, and `Guelph` nodes to the panel assuming only those cities appearing in the table.

2.1.2 Wrangling Edges

Users can create edges that connect nodes by defining an *edge type* which expresses a way to join the tuples in the lineages of the nodes. Users define edge types through a predictive interaction with GW (explained momentarily). Formally, an edge type ET_k is an equi-join query Q and connects two nodes v_1 and v_2 if the output of Q is non-empty. Q is formally expressed as follows (jc stands for **j**oin **c**olumn, and in jc_{x_L} and jc_{x_R}, L and R stand for **L**eft and **R**ight):

$$v_1.lin \underset{jc_{v_1}=jc1_L}{\bowtie} R_{i1} \underset{jc1_R=jc2_L}{\bowtie} R_{i2} \cdots R_{it} \underset{jct_R=jc_{v_2}}{\bowtie} v_2.lin \tag{2.1}$$

Above: (i) $jc_{v_1} \in cols(v_1.rel)$, $jc_{v_2} \in cols(v_2.rel)$, and $jc_{x_L}, jc_{x_R} \in cols(R_{ij})$; (ii) t can be 0, meaning a direct join between the tuples in $v_1.lin$ and $v_2.lin$ on columns jc_{v_1} and jc_{v_2}; and (iii) R_{ij} are not necessarily distinct relations. In our implementation, edges are undirected but a direction to edges can easily be given e.g., the left node v_1 in Equation 2.1 could be the source.

Example 2.3: Suppose a user has created a node v_1 representing customer Alice, with lineage t_5=(`C5`, `Alice`, `Waterloo`), and a v_2 representing customer Bob, with lineage t_7=(`C7`, `Bob`, `Waterloo`). If the user wants to add an edge between v_1 and v_2 because Alice and Bob are from the same city, the formal edge type would be $ET_{SameCity}$=$v1.lin \underset{City=City}{\bowtie} v2.lin$, where both jc_{v_1} and jc_{v_2} are `City` and there is no other relation involved in the join (so t is 0 in Equation 2.1).

Example 2.4: Consider now adding an edge between Alice and Bob because they have ordered the same product. This can be represented by an edge type $ET_{Copurchase}$ (below `Cu`, `Or`, and `Pr` are respectively the `Customer`, `Order`, and `Product` tables):

$$v_1.lin \underset{Cu.C_{ID}=Or.C_{ID}}{\bowtie} Or \underset{Or.P_{ID}=Pr.P_{ID}}{\bowtie} Pr \underset{Pr.P_{ID}=Or.P_{ID}}{\bowtie} Or \underset{Pr.C_{ID}=Cu.C_{ID}}{\bowtie} v2.lin \qquad (2.2)$$

Predictive Interaction: Users define edge types interactively by drawing an edge between two existing nodes v_1 and v_2, and GW starts searching for different queries that can join $v_1.lin$ and $v_2.lin$. The valid edge type definitions GW finds are ranked and presented to the user in a human readable way, as shown in Figure 2.3. The user then selects one of these predictions to indicate the correct definition. We explain more about the prediction mechanisms in Section 2.2.

2.1.3 Wrangling Properties

GW supports two *node property types* and one *edge property type*. Users define properties through an interface by selecting a column from a drop-down list. For node aggregation and edge aggregation properties an aggregation function is also selected (explained momentarily). Figure 2.4 shows the interface for wrangling an edge property.

Node Column Properties

Any column value in $v.cols$ can be added as a property, e.g., in our first running example, we can give Alice the properties City=Waterloo or $\texttt{C}_{\texttt{Name}}$=Alice. Internally we keep the

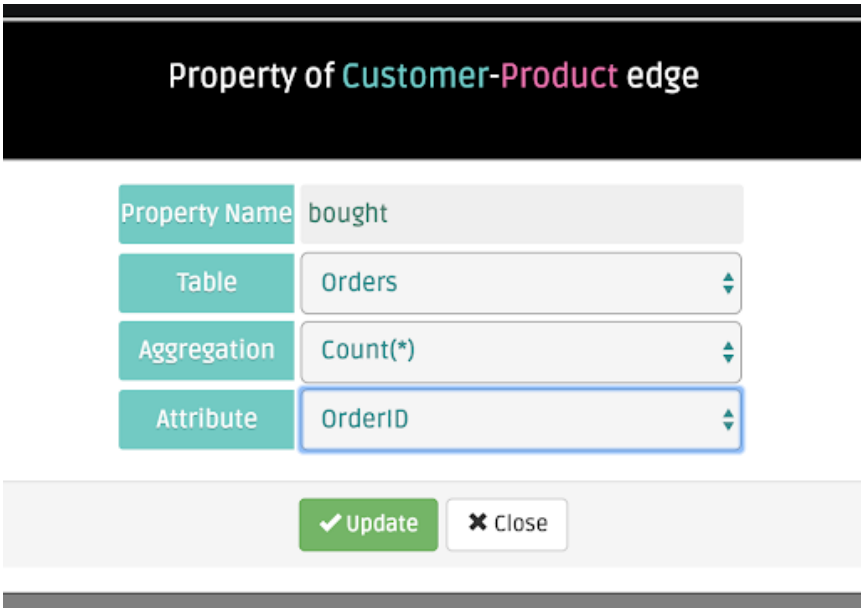


Figure 2.4: Interface for selecting an edge property

column name on which the property is defined.

Node Aggregation Properties:

These properties are the results of aggregation queries on the lineages of a node: $v.lin$. Node aggregation properties are internally expressed as a pair (`key`, γ_A, fn), where `key` is the string of the property, γ_A is an optional numeric-valued column to aggregate, and fn is an aggregation function. The value of the property corresponds to the result of the relational algebra expression $\gamma_{\text{v.cols},fn(\gamma_A)}v.lin$ (so we group by all columns). GW supports count, sum, max, and min functions. The simplest example counts the size of each node's lineage (so translates to a count star query). For other aggregations, γ_A must be a numeric-valued column. Consider our example where we wrangled `City` nodes from the `Customer` table. Suppose the table had a `Salary` column. We could add a node property for the total salaries of customers in each city with a node aggregation property (`Total Salary`, `Salary`, sum).

Edge Aggregation Properties

These are expressed the same way as node aggregation properties but the values are aggregations on the results of the joins defining the edge types. Edge aggregation properties are internally expressed as a set (`key`, γ_R, γ_A, fn), where `key` is the string of the property, γ_R is an optional relation that has aggregated column, γ_A is an optional numeric-valued column from relation γ_R to aggregate, and fn is an aggregation function. The value of the property corresponds to the result of the relational algebra expression:

$$\gamma_{fn(\gamma_R,\gamma_A)}(v_1.lin \underset{jc_{v_1}=jc_{1_L}}{\bowtie} R_{i1} \underset{jc_{1_R}=jc_{2_L}}{\bowtie} R_{i2} \cdots R_{it} \underset{jc_{t_R}=jc_{v_2}}{\bowtie} v_2.lin) \tag{2.3}$$

Similar to the node aggregation properties, the function fn can be count, sum, max, or min. Consider our example where we wrangled $ET_{Copurchase}$ edge between two customers Alice and Bob. We can set a "count" property on the $ET_{Copurchase}$ edge, which shows the number of times both customers bought the same product.

2.1.4 Other Interactions

Users can interact with GW in three other ways. First, users can type keywords or predicates, e.g., "name = Alice", into the search boxes above the tables to get specific rows (shown in Figure 2.1). Second, users can click on a node v_1 and an "Expand" button, which will wrangle all 1st degree neighbors of v_1 based on the existing node and edge types; further clicks expand the graph to higher degree neighbors. This enables users to quickly put a graph view on their relational data, and visualize and explore their tabular data as a graph. Third, users can automatically obtain a script to import all nodes and edges from the RDBMS to GraphflowDB [21], for more advanced graph querying. This streamlines the ETL pipeline of transforming data from relational tables into graphs that can be imported to a graph-specific software [33].

2.2 Predictive Interaction and Ranking

GW's predictive interaction framework discovers possible connections between two nodes and uses a ranking algorithm to show the strongest connections at the top. The predictive interaction framework is triggered when the user draws an edge drawn between two nodes. This action indicates the system that there might be a join connecting these two node

lineages. GW then searches for all possible relations between these two nodes that exist in the RDBMS and present them in the human-readable format as suggestions. The suggestions are presented based on a ranking algorithm, which is described in Section 2.2.2.

2.2.1 Searching of Edge Types

GW inspects the schemas of the tables in the database and generates a set of SQL queries that fit the join template in Equation 2.1. This process involves enumerating different column combinations that can join (e.g., have the same data type) in $v1.rel$, $v2.rel$ and other tables checking whether the output of the corresponding SQL query is empty or not issued with a `LIMIT 1` clause.

For databases with a large number of tables or columns, this search can be very expensive. To present predictions at interactive speeds, GW initially takes two steps: (1) GW issues the queries in increasing order of t and presents them as they are found. This guarantees that if there are edge definitions with small values of t, say t=0, the user always sees some initial suggestions. (2) When the lineages are small in size, GW keeps the lineages in its frontend and performs the search for $t = 0$, i.e., direct joins between $v1.lin$ and $v2.lin$, without querying the RDBMS. This is a common case, e.g., in our first running example both Alice and Bob have single-tuple lineages.

GW also converts joins to selections when performing the search for $t = 1$. Suppose our database has a `Friend(`C_{ID_1}`,` C_{ID_2}`)` table indicating the friendships between customers. Consider checking whether or not Alice and Bob can be connected because they are friends. This edge type involves 1 intermediate relation but can be checked with the query: Select * FROM Friends WHERE C_{ID_1}=C5 AND C_{ID_2}=C7, avoiding any joins. These steps allow GW to operate at interactive speeds on large tables, with millions of rows. Finally, GW currently limits the search to joins involving at most two tables. Users manually specify more complex edge definitions through a separate interface (by clicking the "Not Found?" button in Figure 2.3).

2.2.2 Ranking of Edge Type Definitions

Two nodes can be connected through multiple relations. Edge type definitions are ranked using three heuristics in this order:

1. Definitions with smaller t values rank higher.
2. Definitions with foreign key relations rank higher.

3. Definitions, whose join queries have more outputs rank higher, interpreted as stronger links between nodes.

The ranked edge type definitions are then presented before users showing the higher-ranked definitions at the top. A higher ranked edge definition indicates a stronger and shorter connection.

2.3 Achieving Interaction Speed for Large Datasets

For large datasets, GW's predictive interaction fails to generate suggestions within the user's interactive speed. Recall in Section 2.1.2 while defining an edge between two nodes, the predictive interaction framework generates a set of queries to the RDBMS to find all possible join between the nodes, from where users select the desired edge type. Searching for possible edge types from an interaction is a costly process. The strategies described in Section 2.2.1 keep the process within the interaction speed for small datasets. However, for datasets with large relations, executing a single join query having two relations can take several seconds. While searching for edge type, the system executes multiple of join-queries having different parameters and having multiple lengths of joins following the join template in Equation 2.1. Hence, the whole process of predicting edge definitions may take up to several minutes, making the interaction challenging for users.

To keep the predicting process within the interaction speed, we search for ways to avoid executing queries to the RDBMS after the user draws an edge between two nodes. While exploring with different datasets, we make two observations:

1. While defining an edge, the user interacts with a significantly smaller portion of the database, regardless of the database size. For example, while searching for possible connections between two Customers, Alice and Bob, GW explores the tuples related to these two customers only. The number of tuples in the Order table related to Alice and Bob is much smaller than the size of the Order table.
2. The interaction for defining an edge happens after the user draws the nodes between which the edge will be created. Therefore, the system can pre-compute the possible joins between available nodes before the user initiates the edge definition interaction.

These observations indicate that after a node creation, one of the next possible action is creating an edge involving the node. Therefore, as soon as a node is created, the system can query the database regarding the related tuples of the node and cache the data in

memory. This allows the system to search for possible edge definitions between nodes without executing any query to the database.

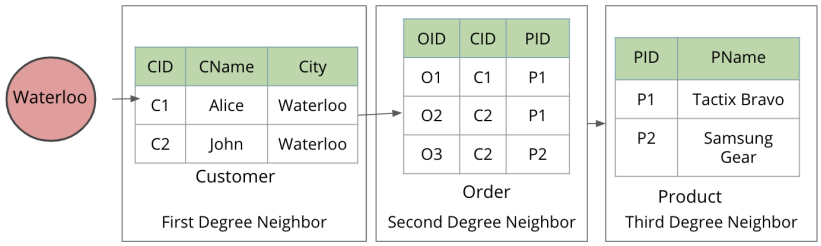
2.3.1 In-Memory Cache

GW maintains an in-memory cache to store related information regarding nodes. To efficiently use the available memory, we find out the minimum data that needs to be stored get from the RDBMS and store in the cache. Recall that, at the first step GW extracts information regarding available tables in the database, table columns, and primary key and foreign key relations. Now, this stored information is used to find out joinable columns between tables. For example, two table columns cannot be joinable if their data type is different. Primary key and foreign key relations are obvious joinable columns.

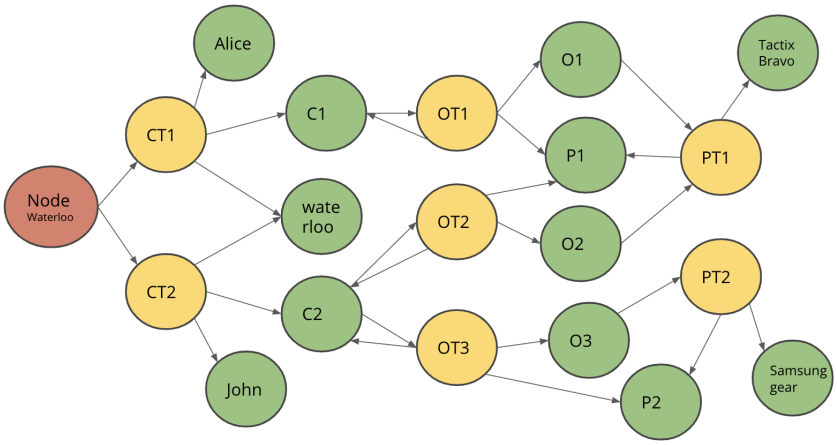
In the beginning, as soon as the user provides the database authentication information, GW collects the necessary information from the database and calculates the joinable columns among available tables. For instance, in our running example, the Customer table can be joined with the Order table using the respective CID columns. The Order table can further join with the Product table using the respective PID columns.

Next, when a node is created, the system detects which relation the node is drawn from and gets the joinable columns with other relations. First GW queries the database to get all the lineages of the node. These lineage tuples will be stored in the cache. This is the first degree neighbor of the node. For each tuple in the lineage, the system detects the joinable column values. For each join relations, the system executes a query to the database to get the joinable tuples in the join-table. Since the system already has the join values from the lineages of the node, it transforms the join query into a select query having an "IN" operator. For example, suppose we have a city node "Waterloo" from the Customer table. Waterloo node has 2 lineages (C1, Alice, Waterloo), (C2, John, Waterloo). To find out the joinable tuples in the Order table relating to the node "Waterloo", the system executes the following select query: "SELECT * From Orders WHERE CID In C1, C2".

When we have the second-degree neighborhood of the node, we can follow a similar process to extend it to the third-degree neighborhood. For example, suppose, the above query gets the following three tuples from the order table (O1, C1, P1),(O2, C2, P1),(O3, C2, P2). Now, to extend the third-degree neighborhood, the system executes the following query: "SELECT * From Product WHERE PID In P1, P2". Thus GW avoids JOIN queries, which makes the query execution much faster. GW stores up to the third-degree neighborhood of a node in the cache for edge prediction. Figure 2.5a shows the neighboring tuples of the 'Waterloo' node.



(a) Getting neighboring tuples from joinable tables.



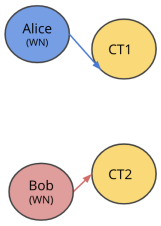
(b) Graph Structure to store the neighborhood in GW's in-memory cache.

Figure 2.5: Neighborhood of the 'Waterloo' node.

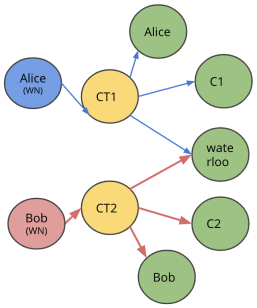
Figure 2.5b shows how the data is stored in GW's in-memory cache data storage. We choose the graph structure for the in-memory data storage to easily traverse the neighborhood of a node. The graph structure contains the following three categories of nodes:

i. *Wrangling node (WN):* WN refers to the nodes wrangled by the user.
ii. *Tuple node (TN):* For each tuple in a table, a TN category node is created.
iii. *Value node (VN)* The values of each tuple are represented by the VNs.

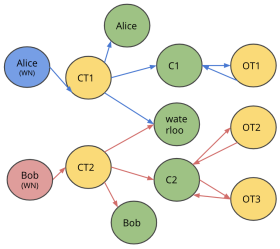
In Figure 2.5b, the red color denotes the WNs, yellow denotes the TNs, and the greens are the VNs. As we can see in the figure, a WN is connected to multiple TN category nodes. These TNs represent the lineages of the WN. Every value in the graph appears only once. We can see, the VN 'C1' has two arrows indicating a TN in the Customer table and a TN in the Order table. Thus the neighborhood of the WNs creates a connected graph.



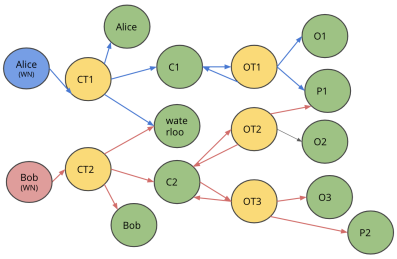
(a) Step 1



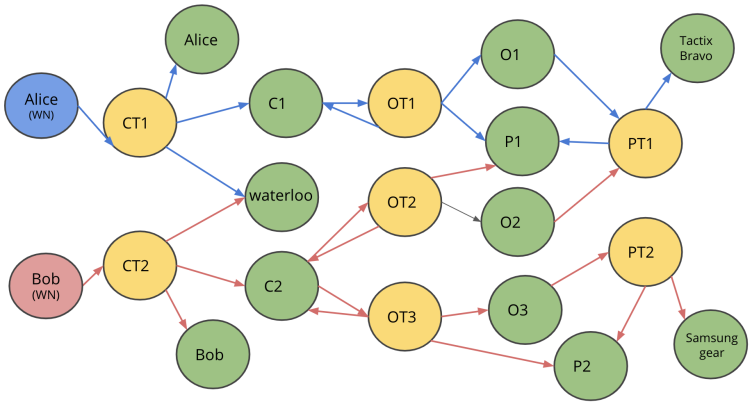
(b) Step 2



(c) Step 3



(d) Step 4



(e) Step 5

Figure 2.6: Bi-directional BFS steps

2.3.2 Searching for edge types using the In-Memory Cache

When a user draws an edge between two nodes, GW traverses the neighborhood of the nodes to find out all the possible connections available between them. The neighborhood is created as soon as the nodes were created. Therefore, no new queries need to be executed after the user draws an edge. To make the traversal faster to find all possible paths between two nodes, we adopt the bidirectional breadth-first search (BDBFS) algorithm.

We start with the source and destination nodes (two ends of the edge) and start expanding the neighborhood from both nodes. First, we expand the source and destination WNs to their first-degree neighborhoods and see if any of the neighboring nodes intersect. If any intersection is non-empty, GW adds the edge definition generated from this intersection to the possible edge suggestion list along with the tuple that intersected. GW continues this process and keeps adding edge definitions for intersecting nodes until it traverses the fifth-degree neighborhood. Figure 2.6 shows the steps of BDBFS when a user draws an edge between two Customer nodes: Alice and Bob. In the first step, Figure 2.6a, GW expands the WNs (showed in blue and red color) to their corresponding first degree neighborhood, which indicates their lineage tuples (TNs). In this figure, we see that both Alice and Bob have single lineages. Then we intersect the first degree neighborhood. No common node is found in this example. Next, the TNs from step 1 expand to their corresponding value nodes (VNs): Figure 2.6b. Here, we find a non-empty intersection, showing both WNs have same city = "Waterloo". We record this finding as a possible edge definition in the suggestion list. The VNs from step 2 then further expands to TNs, which indicates a non-empty join with the Order table. No intersection found in this step as shown in Figure 2.6c. Figure 2.6d shows the next stage of the search, where the graph expands to the VNs, showing the attributes of each tuple. Here, the paths from both WN intersect with same product IDs. This relation is also added to the suggestion list. Finally, in Figure 2.6e, the VNs from step 4 expand to TNs, showing another join between order and product tables. Here, we find another possible edge relation with product table. The graph further expands to the corresponding VNs. However, we stop the search here.

2.4 Implementations

GW runs as a web application and interacts with the user's MySQL database through a backend application. The backend application has an in-memory cache to hold the custom graph described in Section 2.3.1. Figure 2.7 shows the system architecture of GW.

The web application is built on the AngularJS framework. As shown in Figure 2.1, GW

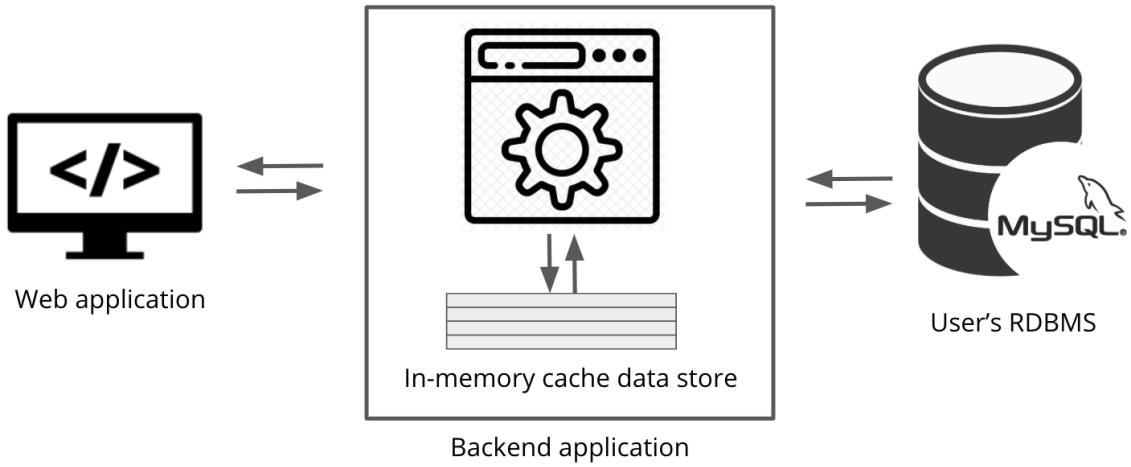


Figure 2.7: GraphWrangler System Architecture

user interface has a graph node-link panel and a table panel. The node-link panel shows the graph and supports all the graph-related interactions, such as, drawing an edge between two nodes to define an edge type, exploring the first and second-degree neighborhood of a node, adding properties on nodes and edges. For smooth graph visualization and user interactions, we integrated the D3.js [4] library. The D3.js library provides different functionalities to show data using SVG, CSS, and HTML. We integrated some of the built-in functionalities from the D3.js library into our code to customize the diverse user interactions that GW supports.

The table panel shows the available tables from the user's RDBMS. A search button on every table lets the users search specific tuples or values in the RDBMS. GW supports drag and drop actions from table panel to the node-link panel. We use interact.js [18] library to capture drag and drop actions.

The front end application captures the user-interaction and communicates with the system's backend application through REST APIs to perform the operations intended by the user-interaction. The backend application also communicates with the user's RDBMS using the user-provided authentication information on the web application. We adopt the Node.js framework to implement GW's backend application. The Node.js application uses a MySQL connector plugin [8] to directly interact with MySQL. The application memory

is used as the in-memory cache data store as explained in Section 2.3.1. Finally, we use a standard implementation of the BDBFS algorithm.

Chapter 3

R2GSync

In this chapter, we describe how we keep a graph in a GDBMS in sync with the updates in the RDBMS that it was extracted from. At first, we explore a design space for automatically synchronizing an RDBMS and a GDBMS in Section 3.1. Next, Section 3.2 discusses R2GSync, a system that we developed to synchronize graphs stored in GraphflowDB upon each update to a source RDBMS. R2GSync is based on *edge views*, which is a technique to allow applications to query a graph using virtual edges over some *base* edges. We then describe two optimizations for queries that contain edge views in Section 3.3. Finally, Section 3.4 illustrates the evaluation of edge views and our proposed optimization strategies on different datasets and queries.

3.1 Design Space for Automatic RDBMS-GDBMS Synchronization

Consider a graph extracted from a relational database D using the mapping we described in Chapter 2. Throughout this chapter, we will also use G and D to refer to the GDBMS and the RDBMS that store G and D, respectively. Let Q_N be a set of SQL queries such that $q_{N_i} \in Q_N$ extracts from D all the nodes of type NT_i in G. Similarly, let Q_E be a set of SQL queries such that $q_{E_i} \in Q_E$ extracts from D all the edges of type ET_i in G. Therefore, Q_N and Q_E are views over the RDBMS. Hence, we have two overall options to keep G synchronized with D:

1. As Q_E and Q_N are views over D, this synchronization problem can be defined as

an *incremental view maintenance (IVM)* problem. We can perform the IVM in two ways:

1.1 Inside the RDBMS: In this design, we create RDBMS views for each $q_{N_i} \in Q_N$ and $q_{E_i} \in Q_E$ and materialize them to capture each update in the views. For example, RDBMS triggers can be used to update the materialized views upon each update in the RDBMS.

1.2 Outside the RDBMS: In this design, we maintain the views outside of the RDBMS, possibly inside of a GDBMS or on an intermediate layer with IVM capabilities, such as the IVM software developed by *Materialize, Inc* [25].

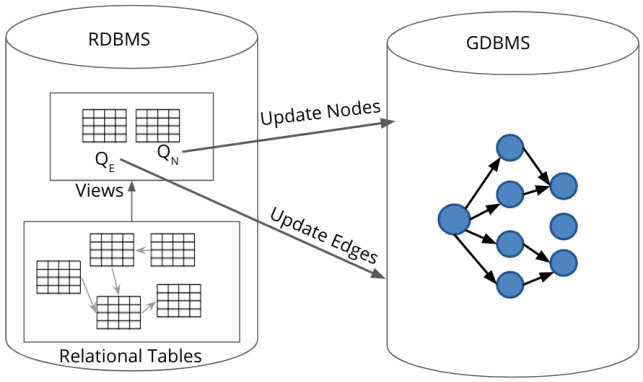
2. Node and Edge Views: In this design, we map the base tables that are used in Q_N and Q_E into *base* node and edges in the GDBMS and support *node and edge views* in the GDBMS to give access to the node and edge types in Q_N and Q_E without materializing the actual nodes and edges.

In the rest of this section, we discuss the pros and cons of each of these designs, which are summarized in Table 3.1 for reference.

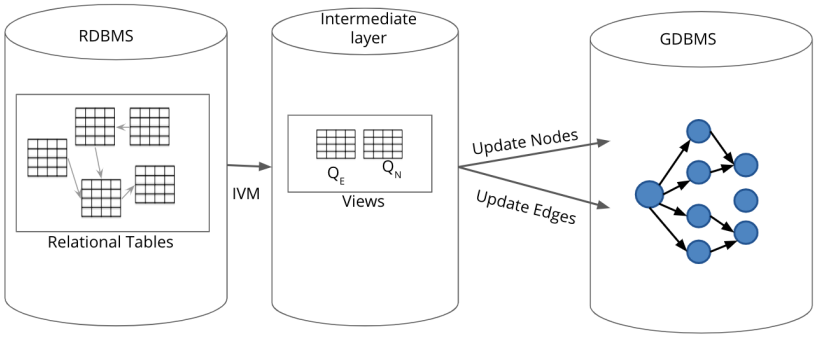
Design	Overhead to RDBMS	Changes to GDBMS	Synchronization Speed	Read Performance	Additional Storage
IVM inside RDBMS	New views	None	Slow	Fast	$2\|G\|$
Separate IVM System	None	None	Slow	Fast	$\|D\| + \|G\|$
Node and Edge Views	None	Edge/ Node View Support	Fast	Slow	$\|D\|$

Table 3.1: A design space for automatic synchronization

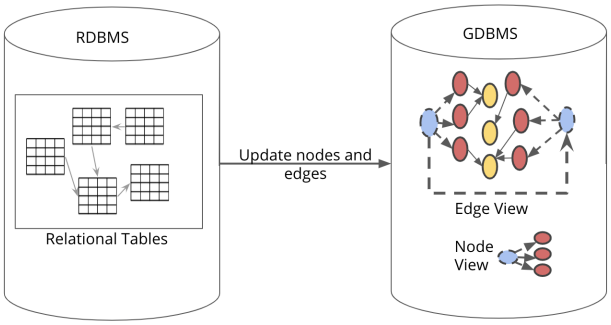
Design 1: IVM inside RDBMS. Figure 3.1a shows our first design, where the node and edge query outputs are incrementally maintained and materialized as views inside the RDBMS. The GDBMS has no overheads in this design. However, supporting IVM inside the RDBMS has several drawbacks. First, the additional storage requirement is $2|G|$ because the graph G has to be stored in both the RDBMS and the GDBMS. This might be a significant overhead because the size of the graph G can be much larger than the source



(a) IVM inside RDBMS



(b) Intermediate layer for IVM



(c) Node and Edge Views

Figure 3.1: Design spaces for automatic synchronization.

database D. For example, the co-purchase edges mentioned in Equation 2.2 in Section 2.1.2, one customer who bought a single product may generate multiple edges to customers who also bought the same product. In addition, the RDBMS has to incrementally maintain each query in Q_E and Q_N, which adds additional workload on the RDBMS and slows down its update performance. If the RDBMS is a critical transactional store in an enterprise, this design is possibly not a practical option.

Design 2: Separate layer for IVM. Figure 3.1b shows the design of having an intermediate layer between RDBMS and GDBMS for IVM. This intermediate layer consists of a data storage to store the materialized views and a system to incrementally maintain them upon each update in the RDBMS. An example of such a system is the IVM software developed by *Materialize, Inc* [25]. This design relieves both RDBMS and GDBMS from any change or overhead for the synchronization process. However, the additional storage requirement is $|D| + |G|$ because a copy of D has to be stored in the separate IVM system and G has to be stored in the GDBMS. As long as the extracted graphs are not prohibitively large, this design is ideal as it puts together three systems without requiring major modifications to any system. For example, we can use Materialized as the IVM layer, which incrementally maintains the RDBMS materialized views. However, this product is only commercially available. We are not aware of any publicly available IVM system, except of course to use a second RDBMS that supports IVM.

Design 3: Node and Edge Views. Finally, Figure 3.1c shows the node and edge view design, which stores the data of the participating relations in Q_E and Q_N queries as base nodes and edges directly in the GDBMS. Therefore, each update in the RDBMS directly maps into a node and/or edge update. Because this design does not perform IVM, synchronizing the GDBMS is faster than the previous two designs. To query the actual transformed graph, the GDBMS needs to support views on nodes and edges according to the node and edge queries in Q_N and Q_E, respectively. Similar to Design 2, this design also has no overheads to the RDBMS. Furthermore, the node and edge view design guarantees no significant data expansion as the size of the data moved to the GDBMS is equal to the size of D regardless of the size of G. However, the read performance for the GDBMS gets slower as we need to execute node and edge views on graph queries.

Although there are settings, when any of these designs can be preferred over others, in this thesis, we investigate the design option of node and edge views.

3.2 R2GSync and Edge Views

In this section, we describe R2GSync, a system that we developed to automatically synchronize a transformed graph in a GDBMS upon each update in the source RDBMS. R2GSync adopts the node and edge view design described in Section 3.1. Throughout this section, we will differentiate between a *transformed graph* G and a *base graph* G_B. G is the actual graph the user wrangled from D and consists of nodes and edges according to the node and edge type queries Q_N and Q_E, respectively. The base graph will contain the *base nodes* and *base edges* that represent the tables and the joins between the tables in Q_N and Q_E. G_B may differ in structure from G. In Design 3, the node and edge view capabilities inside G_B lets users access and query G hiding G_B. For simplicity, R2GSync only supports edge views. That is, we do not support any node definitions that require having a node view on the stored graph (explain in more detail momentarily).

Throughout this section, our running example includes an RDBMS with the following schemas: Product(PID, PName, MID), Customer(CID, CName, City), Order(OID, CID, PID, Amount) and Merchant(MID, MName). Here, PID, CID, OID, and MID are primary keys of relations Product, Customer, Order, and Merchant respectively. MID in the Product table is the foreign key of the Merchant table and PID and CID attributes in the Order relation are foreign keys of Product and Customer relations respectively. A user wrangles a graph using GraphWrangler which includes customer node typed having (CID, CName, City) from the Customer relation. In addition, a co-purchase edge is wrangled between customers who bought the same products. The edge definition between two customers v1 and v2 is defined as below (here, `Or` and `Pr` represent the `Order` and `Product` tables, respectively):

$$v_1.lin \underset{v_1.CID=Or.CID}{\bowtie} Or \underset{Or.PID=Pr.PID}{\bowtie} Pr \underset{Pr.PID=Or.PID}{\bowtie} Or \underset{Or.CID=v_2.CID}{\bowtie} v2.lin$$

3.2.1 Direct Mapping of Relational Tables into a Graph

In order to avoid implementing node view capability, R2GSync supports node definitions where each tuple of a relation transforms into a distinct node, i.e., each node has the lineage count, i.e., $|v.lin| = 1$. Consider a node type NT_k defined by $(R_i, cols(NT_k) \subseteq cols(R_i))$. Recall from Section 2.1.1 that each distinct value of the projection of R_i into $cols(NT_k)$ is transformed into a node of type NT_k in the graph. To ensure each node has $|v.lin| = 1$ the projected columns $cols(NT_k)$ must include a key (primary or a unique key) of R_i.

R2GSync stores the attributes in $cols(NT_k)$ as node properties in G_B to map each tuple in the table to a distinct node in the graph. Therefore, direct mapping directly follows the node definition defined by the users.

Example 3.2.1: For customer type nodes, each tuple in the Customer relation maps into a distinct customer node having three properties: CID, CName, and City. The CID property of the customer directly maps the node to a distinct tuple in the customer relation.

An edge in GraphWrangler is the result of joins of different lengths connecting the node lineages. As we restrict node definition to contain a key of the relation, R2GSync supports joins in our edge definitions, where the equality predicates match the key of one relation with the key of another. Specifically, given an edge type:

$$v_{N_1} \underset{jc_{N_1}=jc_{1_L}}{\bowtie} R_{i_1} \cdots \underset{jc_{(m-1)_R}=jc_{m_L}}{\bowtie} R_{i_m} \underset{jc_{m_R}=jc_{(m+1)_L}}{\bowtie} \cdots R_{i_r} \underset{jc_{r_R}=jc_{N_2}}{\bowtie} v_{N_2} \tag{3.1}$$

jc_{N_1} is a key from the relation R_{N_1} and also a property of node type NT_1. jc_{N_2} is key from the relation R_{N_2} and also a property of node type NT_2. R_{i_k} are the intermediate relations participating in the join and jc_{k_L} and jc_{k_R} are two attributes of the relation R_{i_k}, where k = 1, 2,...,r. We map the relations in the edge definition to the following base nodes and edges:

1. Each unique R_{i_k} maps to a base node type $NT_{i_k} = (R_{i_k}, cols(NT_{i_k}))$ in the GDBMS where $cols(NT_{i_k})$ includes the attribute jc_{k_L} and jc_{k_R}. If neither of the attributes represent a primary (or unique key) of R_{i_k}, $cols(NT_{i_k})$ includes the primary key jc_{k_p} of R_{i_k}.

2. For each (R_{i_j}, $R_{i_{j+1}}$), there is a base edge type $ET_{j,j+1}$ between nodes extracted from relation R_{i_j} and $R_{i_{j+1}}$. Here, the base edge type $ET_{j,j+1}$ between two nodes v_{i_j} and $v_{i_{j+1}}$ of node type respectively, NT_{i_j} and $NT_{i_{j+1}}$, is denoted as follows:

$$v_{i_j} \underset{jc_{j_R}=jc_{(j+1)_L}}{\bowtie} v_{i_{j+1}}$$

   If $jc_{(j+1)_L}$ in $R_{i_{j+1}}$ is a foreign key of attribute jc_{j_R} in R_{i_j}, the edge $ET_{j,j+1}$ is directed from node type NT_{i_j} to node type $NT_{i_{j+1}}$. If the opposite is true, $ET_{j,j+1}$ is directed from node type $NT_{i_{j+1}}$ to node type NT_{i_j}.

Example 3.2.2: For the co-purchase edge in our running example, the direct relation mapping creates the following base nodes and base edges:

- *Base node types* `PRODUCT` *and ORDER maps the* `Product` *and* `Order` *relations respectively.*
- *For* `Customer, Order` *relations, there are a base edge types* `CUST-OR` *between node type* `CUSTOMER` *and base node type* `ORDER`*. Similarly, for* `Order, Product` *relation pair, there is a base edge types* `PR-OR` *between base nodes* `ORDER` *and* `PRODUCT`*.*

3.2.2 Mapping Each Update in the RDBMS to the Corresponding Update in the GDBMS

We next explain how R2GSync issues updates to GDBMS upon each update in the RDBMS. For each insertion, deletion, and update event in the RDBMS, R2GSync executes the following changes:

- *Insertion*: Consider an insertion of a tuple t into a relation R_i with the primary key jc_{i_P}. For each Q_{E_j} that R_i is part of, R2GSync performs the following: consider the left and right join attributes of R_i in Q_{E_j} be jc_{i_L} and jc_{i_R}, respectively.
  1. Insert a base node n_i of type NT_i with jc_{i_P}, jc_{i_L}, and jc_{i_R} as properties of the node.
  2. Insert a base edge of type $ET_{i-1,i}$ between n_i and nodes of type NT_{i-1} having the same property value as jc_{i_L} (If R_i is not the leftmost relation in Q_{E_j}).
  3. Insert another base edge of type $ET_{i,i+1}$ between n_i and nodes of type NT_{i+1} having the same property value as jc_{i_R} (If R_i is not the rightmost relation in Q_{E_j}).
- *Deletion*: Suppose t in now deleted from R_i:
  1. Delete all the base edges linked with the base node n_i.
  2. Delete the base node n_i.
- *Update*: An update in tuple t into $\acute{t}$ is handled by simply deleting t and inserting $\acute{t}$.

Example 3.2.3: Suppose a tuple with values ('O1', 'C1', 'P1', 10.5) is inserted into the Order table. R2GSync issues the following queries :

```
CREATE (n:ORDER {OID: 'O1'}) return n
CREATE (PRODUCT{PID: 'P1'})<-[:PR-OR]-(n)<-[:CUST-OR]-(CUSTOMER {CID: 'C1' })
```

3.2.3 Edge Views

Edge views allow users to access and query G hiding from users G_B, which is actually stored in the GDBMS. For instance, using edge views, in our running example a user can directly query (C1:CUSTOMER)-[:COPURCHASE] $\rightarrow$ (C2:CUSTOMER) even though there is no edge of type `COPURCHASE` in G_B. Consider an edge type ET_k that was defined when G was wrangled. Recall from Section 2.1.2 that there is an edge of type ET_k between two wrangled nodes u and v if the join, according to ET_k, between the lineages of u v is non-empty. When edges of type ET_k are not materialized in G_B and access to them is through edge views, this is equivalent to the conditiona that two nodes u and v have an edge with the given edge view ET_k, if there is at least one path (using base edges) between u and v that satisfy the ET_k's join. Therefore, computing the edges (forward or backward) of a node u according to an edge view ET_k is the result of the join of ET_k over G_B with a final DISTINCT computations.

Example 3.2.4: In our running example, the COPURCHASE edges of a customer node c_i is the result of the following query:

MATCH (C1:CUSTOMER) $\rightarrow$ (O1:ORDER) $\rightarrow$ (P:PRODUCT) $\leftarrow$ (O2:ORDER) $\leftarrow$ (C2:CUSTOMER)
WHERE C1.ID = c_i
RETURN DISTINCT c2

Implementation

We implemented edge views on top of a prototype in-memory graph database called GraphflowDB. GraphflowDB has a traditional DBMS architecture. The system uses the *Cypher* query language [12]. In Cypher, a query consists of 3 parts: i) a `MATCH` clause that describes a subgraph query pattern $Q(V_Q, E_Q)$, where V_Q and E_Q are the query nodes and edges, respectively, that the system matches on an input graph; ii) a `WHERE` clause that contains a predicate ρ over properties of the edges and nodes that the matched subgraph must satisfy; and iii) a `RETURN` statement that returns a projection of the variables in the match query or performs a group-by and aggregate information. The query plans of GraphflowDB are linear and consist of standard relational operators: Scan, Extend/Intersect, which is the join operator, Filter, GroupBy, and OrderBy. The following contain a brief description of the major operators used for matching a subgraph pattern and evaluating predicates in the query.

- `SCAN`: Scans a set of nodes and edges from an input graph.

- `EXTEND/INTERSECT (E/I)`: At a high level, the E/I operator matches the subgraph query pattern Q, one query vertex at a time. The input to the E/I operator is a partial match t that has already matched k of the query edges in Q. The operator extends a partial match t with, say, k already matched query vertices with one more query vertex u_q. In order to do this, the operator finds all query vertex and edge pairs $(v_{q1}, e_{q1}), ..., (v_{qz}, , e_{q1})$, where v_{qi} have already been matched in t and the other end of e_{qi} is u_q. Note that each of the v_{qi} are already bound to some actual vertex v_i in the input graph and e_{qi} indicates either the forward or backward adjacency list of v_i. The E/I operator takes these adjacency lists and intersects them and extends t with each result of this intersection. If there is only one adjacency list, then the operator simply extends t with each of the nodes in that adjacency list.

- `FILTER`: This is the selection operator in the relational algebra which removes a partial match t from query results if t does not satisfy a given predicate ρ.

- `GROUP-BY-AND-AGGREGATE`: This is an implementation of the standard group by and aggregate relational algebra operator.

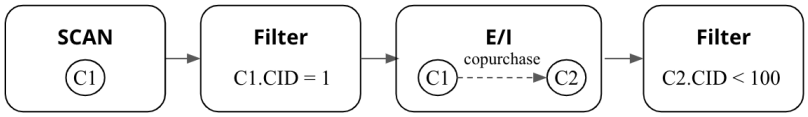
In GraphflowDB, a user can define a path query as an edge view ET_k between two nodes. The following query template supports edge view query definition:

CREATE EDGE VIEW $\langle EV_T \rangle$ AS
MATCH $\langle Q_P(V_Q, E_Q) \rangle$
RETURN DISTINCT v_{N_1}, v_{N_2}

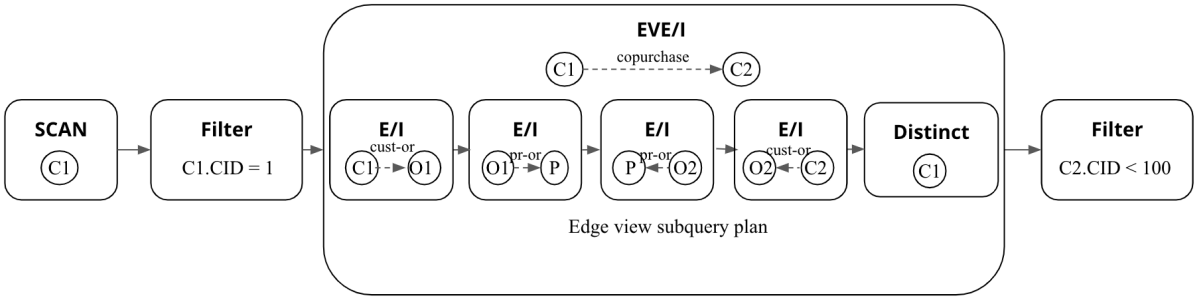
Above v_{N_1} and v_{N_2} denotes variables designating the source and destination vertices of an edge view of type ET_k. $Q_P(V_Q, E_Q)$ is a path query with end variables v_{N_1} and v_{N_2}. For our edge view definition in 3.1, the path query $Q_P(V_Q, E_Q)$ can be expressed as follows:

$$\begin{aligned} Q_P(V_Q, E_Q) : (v_{N_1} : NT_1) - [: ET_{1,i_1}] \rightarrow (: NT_{i_1}) \cdots \rightarrow (v_{i_m} : NT_{i_m}) \leftarrow \ldots \\ \ldots (: NT_{i_k}) \leftarrow [: ET_{2,i_k}] - (v_{N_2} : NT_2) \end{aligned} \tag{3.2}$$

Here, the edge directions are drawn arbitrarily. In our implementation in GraphflowDB, we manage a separate edge view store S_{EV}. For each edge view ET_k, the system stores two plans, *a forward extension plan* P_F and *a backward extension plan* P_B, to compute the actual edges of vertices with type ET_k during query processing. The version of GraphflowDB does not contain a full-fledged optimizer, and instead has a plan enumerator from which a user can manually pick a plan. We use this enumerator to pick plans P_F and P_B. Specifically, for P_F, we issue the following query:



(a) A query execution plan with an Extend operator using the COPURCHASE edge view



(b) The same plan when E/I is replaced with EVE/I

Figure 3.2: The final query execution plans that replace the Extend operator for edge views with EdgeViewExtend operator having sub-plans for query Q_T

MATCH $\langle Q_P(V_Q, E_Q)\rangle$
WHERE $v_{N_1}.ID = 1$
RETURN DISTINCT v_{N_2}

Then from the returned list of plans we pick the plan that starts by scanning and filtering v_{N_1} as the first operators. Similarly, for P_B, we issue a query where v_{N_2} is fixed to a single vertex and the RETURN clause returns v_{N_1}.

For queries using edge views, we generate the query execution plan in the following two steps utilizing GraphflowDB's default plan enumerator.

1. We generate plans using the system's enumerator which treats edge views as regular edges. For example, consider the following query that uses a `COPURCHASE` edge view.

Query 3.2.3

MATCH (C1: CUSTOMER) − [e: COPURCHASE] → (C2 : CUSTOMER)
WHERE C1.CID = 1 AND C2.CID = 5
RETURN C2

When generating plans, GraphflowDB treats the `COPURCHASE` edge view as a regular edge and uses an Extend (COPURCHASE) operator for each COPURCHASE edge view as shown in Figure 3.2, which exhibits two different query plans generated by GraphflowDB's default enumerator for this query.

2. We replace each Extend operator having an edge view in a plan with a special *EdgeViewExtend/Intersect* (EVE/I) operator. EVE/I operator performs exactly the same computation as the E/I operator, except for each forward (backward) adjacency list of a vertex that it needs to access, EVE/I first runs the P_F (P_B) to first generate the adjacency list. For example, Figure 3.2b presents the final query plan for Q_T, which replaces the Extend operators in Figure 3.2a with EVE/I operator.

3.3 Optimizations for EdgeViewExtend/Intersect

In this section, we explore optimization opportunities for executing queries using edge views. We start by describing an optimization that pushes the final distinct operator that EVE/I operator uses to earlier intermediate joins. Next, we propose an optimization on queries that perform the intersection of two or more edge views in EVE/I operator. We also explain the implementation details for both of these optimization strategies inside GraphflowDB.

3.3.1 Early Distinct

Recall from Section 3.2.3 that we compute an edge view between two nodes using an EVE/I operator by executing a subplan that starts from a specific starting node, v_{N_1} of say type NT_1, and returns the distinct target nodes v_{N2}, of say type NT_2, satisfying the edge view path query. Recall that the edge definition in Equation 3.1, the EVE/I operator executing a forward plan can be expressed as follows (we draw the edge directions arbitrarily):

Query 3.3.1

ET_k : MATCH $(v_{N_1}{:}NT_1)\text{–}[{:}ET_{1,i_1}]\rightarrow (:NT_{i_1}) \ldots \rightarrow (v_{i_m}{:}NT_{i_m}) \leftarrow \ldots (:NT_{i_k}) \leftarrow [{:}ET_{i_k,2}]\text{–} (v_{N_2}{:}NT_2)$
WHERE v_{N_1}.ID $= v_1$
RETURN DISTINCT v_{N_2}

Here, the EVE/I operator's edge view subplan starts from each incoming node v_{N_1} of type NT_1 and returns the distinct values of node v_{N_2} of type NT_2. Recall further that we can express ET_k in the relational algebra as follows:

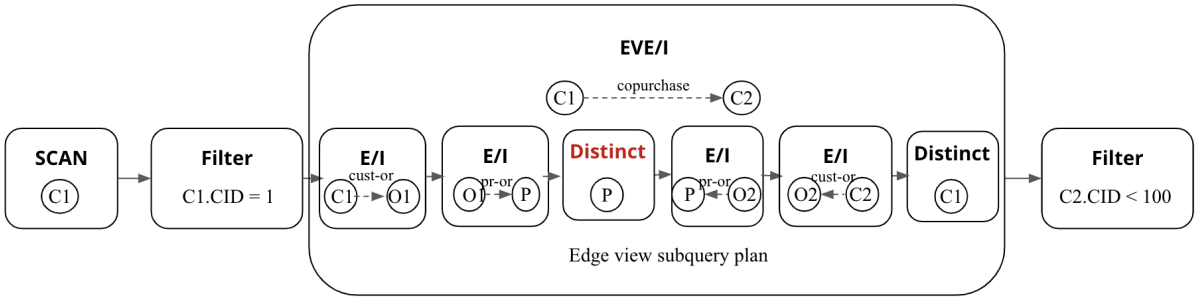


Figure 3.3: A Query plan for query Q_T with an EVE/I-ED operator that computes an early distinct on the product node

$$ET_k = \pi_{v_{N_2}}(\sigma_{ID=v_1}(NT_1) \bowtie NT_{i_1} \ \ldots \ \bowtie NT_{i_m} \ \ldots \ \bowtie NT_{i_k} \bowtie\bowtie NT_2 \ )$$

NT_p above represents a node table of type NT_p. Note that because we are computing the distinct node values of NT_2 through a projection, we can also compute distinct after any intermediate join in the above query. For example, the following expressions are equivalent to ET_k in relational algebra, each computing a distinct operation on different parts of the query.

$$\begin{aligned}
ET_k &= \pi_{v_{N_2}}(\sigma_{ID=v_1}(NT_1) \bowtie NT_{i_1} \ldots \bowtie NT_{i_m} \ldots \bowtie NT_{i_k} \bowtie NT_2) && (3.3)\\
&= \pi_{v_{N_2}}(\pi_{v_{i_1}}(\sigma_{ID=v_1}(NT_1) \bowtie NT_{i_1}) \ldots \bowtie NT_{i_m} \ldots \bowtie NT_{i_k} \bowtie NT_2) && (3.4)\\
&= \pi_{v_{N_2}}(\pi_{v_{i_m}}(\pi_{v_{i_1}}(\sigma_{ID=v_1}(NT_1) \bowtie NT_{i_1}) \ldots \bowtie NT_{i_m}) \ldots \bowtie NT_{i_k} \bowtie NT_2) && (3.5)\\
&= \pi_{v_{N_2}}(\pi_{v_{i_m}}(\sigma_{ID=v_1}(NT_1) \bowtie NT_{i_1} \ldots \bowtie NT_{i_m}) \ldots \bowtie NT_{i_k} \bowtie NT_2) && (3.6)
\end{aligned}$$

Therefore, we can compute distinct early after any intermediate join in the query. If one considers the path query of an edge type as being evaluated through breadth-first-search traversal, this optimization in graph terms is equivalent to taking a distinct computation at some intermediate level i in the search and expanding the frontier of level i from each node in level i. However, computing distinct adds computational overhead and is only beneficial when the nodes in a level are likely to contain multiple paths arriving to them, e.g., if the previous extension was an n-to-1 or an n-to-n join. In relational algebraic terms, computing distinct early on relation NT_{i_m} should be done if the join $\sigma_{ID=v_1}(NT_1) \bowtie NT_{i_1} \ldots \bowtie NT_{i_m}$

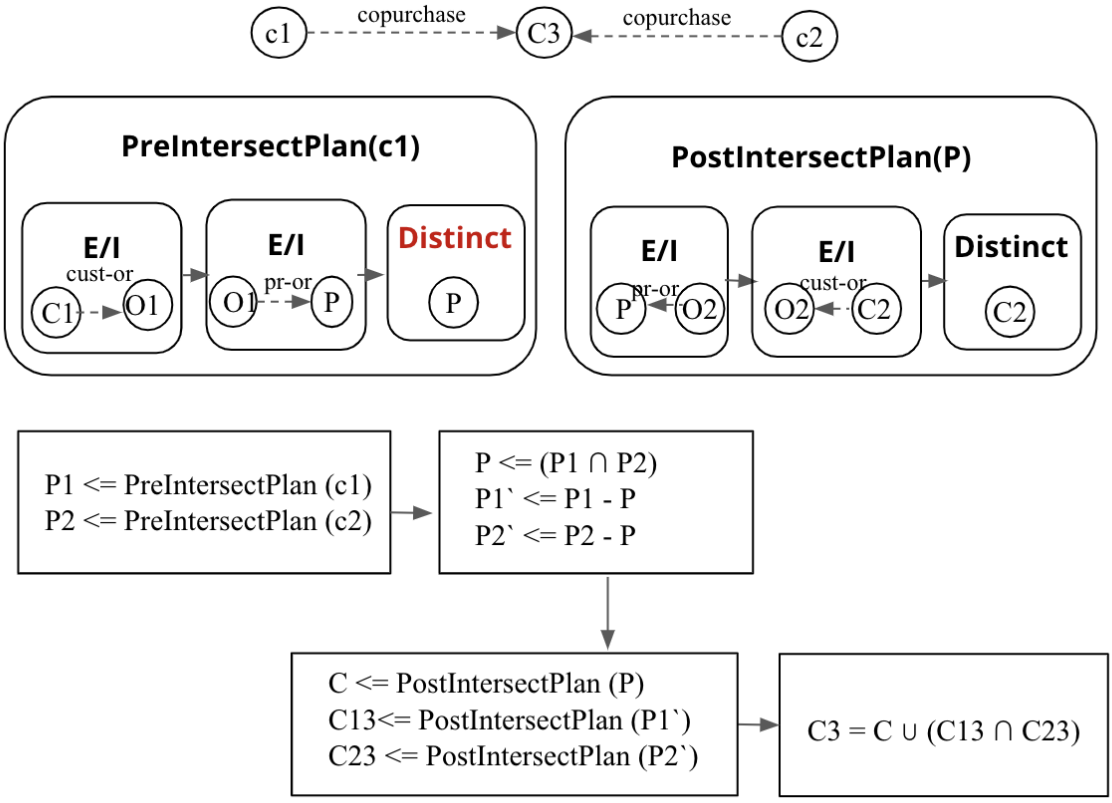


Figure 3.4: Implementation of Early Intersect Optimization in EVE/I operator

can return multiple tuples with the same NT_{i_m} values. This can happen for example if the previous join $NT_{i_{m-1}} \bowtie NT_{i_m}$ is an n-to-1 or n-to-n join (or some earlier join). For example, when computing a path from a CUSTOMER node to PRODUCT nodes in the COPURCHASE edge view, a customer Alice may have bought a single product 10 times, i.e., 10 different orders from Alice (o1, o1, o3, ...) has product p. Therefore, when computing co-purchasers of Alice, instead of expanding p 10 times, p can be expanded only once if we compute an early distinct on the product node. On the other hand, computing distinct on the order node does not make sense as all the customers have distinct orders.

We perform the early distinct optimization in GraphflowDB by adding a new DISTINCT aggregator operator after some extend in the P_F and P_B plans that EVE/I uses to perform extensions using edge views. Currently we manually pick where in P_F and P_B

the distinct computation should be performed. Integrating this choice into the optimizer of the system is left for future work.

To compute an early distinct on v_{i_m}, we manually add a distinct operator after the E/I operator for node v_{i_m} in the edge view subquery plan (P_F or P_B). The rest of the plan stays the same. For example, Figure 3.2b shows the final plan with the default EVE/I operator for the COPURCHASE edge view stated in Query 3.2.3. We can apply the early distinct optimization by taking distinct on the product nodes. The optimized query plan is shown in Figure 3.3, where a distinct operator is placed after getting products from extending PR-OR edges in the EVE/I operator, which computes an early distinct on the product nodes.

3.3.2 Early Intersect

Next, we investigate the intersection operation in EVE/I operator. The EVE/I operator computes an intersection if it is configured with two or more edge views. Cyclic queries are examples where the query execution plan needs to compute an intersection of two or more edges. Suppose an EVE/I operator that takes two adjacency lists of node $v1$ and $v2$ and computes an intersection of edge views from $v1$ and $v2$ nodes, respectively. Suppose for simplicity that $v1$ and $v2$ are both of type NT_1 and their edge views are the same and of type ET_k, which is expressed in Query 3.3.1. The following describes the relational algebraic operation that the EVE/I operator would performing when intersecting $v1$ and $v2$'s edges of type ET_k:

$$\begin{aligned} Q_{Intersect} = \pi_{v_{N_2}}(\sigma_{v1}(NT_1) \bowtie NT_{i_1 1} \ldots \bowtie NT_{i_m} \ldots \bowtie NT_{i_k} \bowtie NT_2) \\ \cap \pi_{v_{N_2}}(\sigma_{v_2}(NT_1) \bowtie NT_{i_1 2} \ldots \bowtie NT_{i_m} \ldots \bowtie NT_{i_k} \bowtie NT_2) \end{aligned} \tag{3.7}$$

Consier the following three sub-expressions from 3.7:

$$\begin{aligned} SE_1 : \sigma_{v_1}(NT_1) \bowtie NT_{i_1 1} \ldots \bowtie NT_{i_m} \\ SE_2 : \sigma_{v_2}(NT_1) \bowtie NT_{i_1 2} \ldots \bowtie NT_{i_m} \\ SE_m : NT_{i_m} \ldots \bowtie NT_{i_k} \bowtie NT_2 \end{aligned}$$

Rewriting the expression in 3.7, we get:

$$Q_{Intersect} = \pi_{v_{N_2}}(\pi_{v_m}(SE_1) \bowtie_{v_m} SE_m) \cap \pi_{v_{N_2}}(\pi_{v_m}(SE_2) \bowtie_{v_m} SE_m) \tag{3.8}$$

Observe that SE_m is a common expression both in v_1's edge view query as well as v_2's. Now consider computing the intersection of $\pi_{v_m}(SE_1) \cap \pi_{v_m}(SE_2)$. This "early" intersect will identify common nodes of type NT_{i_m} that the edge view from v_1 and v_2 have both reached, which implies that the join of this early intersect's result with SE_m are guaranteed to be in the final intersection. We can turn this observation into the following optimization: Now let us define the following terms:

$$\pi_{v_m} SE_1 = M1$$
$$\pi_{v_m} SE_2 = M2$$
$$M = M_1 \cap M_2$$

Finally, let us rewrite the expression in Equation 3.8 as follows:

$$Q_{Intersect} = \pi_{v_{N_2}}((M \cup (M1/M)) \bowtie_{v_m} SE_m) \cap \pi_{v_{N_2}}((M \cup (M1/M)) \bowtie_{v_m} SE_m) \quad (3.9)$$
$$= (\pi_{v_{N_2}}(M \bowtie_{v_m} SE_m) \cup \pi_{v_{N_2}}((M1/M) \bowtie_{v_m} SE_m)) \quad (3.10)$$
$$\cap (\pi_{v_{N_2}}(M \bowtie_{v_m} SE_m) \cup \pi_{v_{N_2}}((M2/M) \bowtie_{v_m} SE_m)) \quad (3.11)$$
$$= (Q_{S_m} \cup Q_{S_1}) \cap (Q_{S_m} \cup Q_{S_2}) \quad (3.12)$$
$$= Q_{S_m} \cup (Q_{S_1} \cap Q_{S_2}) \quad (3.13)$$

Therefore, the query $Q_{Intersect}$ can be computed using three subqueries Q_{S_1}, Q_{S_2}, and Q_{S_m}, where $\boldsymbol{Q_{S_1} = \pi_{v_{N_2}}(((M1/M) \bowtie_{v_m} SE_m))}$, $\boldsymbol{Q_{S_2} = \pi_{v_{N_2}}((M1/M) \bowtie_{v_m} SE_m))}$, and $\boldsymbol{Q_{S_m} = \pi_{v_{N_2}}(M \bowtie_{v_m} SE_m)}$. This can be a faster way to perform $Q_{Intersect}$ because often the sizes of the intermediate results in path queries increase as we perform more joins (when joins are 1-to-n or n-to-n). Therefore the intersection produced when computing $\pi_{v_m}(SE_1) \cap \pi_{v_m}(SE_2)$ and the following join with S_m to compute Q_{S_m}) can be smaller than the original intersection in Equation 3.7. Furthermore, when one of the $(M1/M)$ or $(M2/M)$ is zero, $(Q_{S_1} \cap Q_{S_2})$ will return an empty set. In this case, we can only execute subquery Q_{S_m} to get all the final outputs. For example, consider computing a set of customers (c3) who are co-purchasers of both Alice and Bob, where both of them are also co-purchasers of each other. Here, an EVE/I operator executes an intersection of the COPURCHASE edge views from CUSTOMER nodes for Alice and Bob. If Alice and Bob has a large number of common products, computing an early intersection on the PRODUCT node in the COPURCHASE path query will expand the common products only once, which may reduce execution time. Furthermore, suppose, Alice bought 100 products and Bob bought only 2 smart watches, which was also bought by Alice. In this

case, as Bob has no products other than the common products with Alice, we can only expand the PRODUCT nodes for these two smart watches to get all the co-purchasers of both Alice and Bob. In our evaluations, we will show that the early intersect optimization can yield significant improvements in the runtimes of some queries.

In order to implement the early intersect operator, we first manually pick the NT_{i_m} on which the early intersect should happen and then modify the EVE/I operator to perform the rest of the computation in Equation 3.13. For example, Figure 3.4 shows the implementation of the EVE/I operator for this query.

3.4 Evaluation of Query Processing With Edge Views

In this section, we demonstrate the performance of the early distinct and early intersect optimizations that we discussed in Section 3.3 over the default implementation of the EdgeViewE/I operator for different datasets and queries.

3.4.1 Setup

Hardware: For all our experiments, we use a single Macbook Pro machine that has a 2.3 GHz Intel Core i5 processor and 8 GB of RAM. The machine has two physical cores and four logical cores. We use a single thread for all experiments. We set the maximum size of the JVM heap to 6 GB and keep JVM's default minimum size.

Datasets: We use the *IMDB* [16] and *TPC-H* [38] (size factor 10) datasets for our experiments; both are converted from a tabular format to a data graph. Table 3.2 provides the graph characteristics for each datasets.

Queries: In our experiments, we evaluate our query evaluation techniques with edge views using three queries for *IMDB* and one query for *TPC-H* for a total of four queries. We generate our queries from a very simple query template that can demonstrate the benefits of our optimizations. Our query template Q_T is a two hop query, which takes four parameters: 1) a node type NT_k; 2) an edge view ET_k; and 3) two vertex ID values ID_l and ID_r picked such that a→b ∈ ET_k where a.ID = ID_l and b.ID = ID_r. Q_T is defined below:

$Q_T(NT_k, ET_k, \text{ID}_l, \text{ID}_r)$:
MATCH (a: NT_k)−[e1: ET_k]→(c: NT_k)←[e1: ET_k]−(b: NT_k)
WHERE a.ID = ID_l, b.ID = ID_r
RETURN DISTINCT c

Graph	Nodes		Edges	
IMDB	**Type**	**Count**	**Type**	**Count**
	ACTOR	817,718	ACTOR-MOVIE	3,431,966
	MOVIE	363,575	MOVIE-GENRE	395,119
	GENRE	21	MOVIE-DIRECTOR	371,180
	DIRECTOR	85,794		
	Total	**1,267,108**	**Total**	**4,198,265**
TPC-H	**Type**	**Count**	**Type**	**Count**
	CUSTOMER	999,982	CUST-ORDER	15,000,000
	ORDER	15,000,000	ORDER-LINEITEM	59,986,052
	LINEITEM	2,000,000		
	Total	**17,999,982**	**Total**	**74,986,052**

Table 3.2: Dataset Description

Our four queries are Q_T with different parameters passed. The first three are defined on the *IMDB* dataset and the fourth is defined on the *TPC-H* dataset:

- Given two actors a and b who acted in at least a single movie together, $Q_{co-actor}$ returns the set of actors who acted with each.
  $\boldsymbol{Q_{co-actor}}$**(ID$_l$, ID$_r$)** $=$ $\boldsymbol{Q_T}$**(ACTOR, COSTAR, ID$_l$, ID$_r$)** where the **CO-ACTOR** edge view between two ACTOR nodes is defined using the path query in 3.2 as follows:

  (a: ACTOR)−[:ACTOR−MOVIE]→(m:MOVIE)←[:ACTOR−MOVIE]−(b: ACTOR)

- Given two actors a and b who share at least a single genre in movies they acted in, $Q_{same-genre}$ returns the set of actors who share at least a single genre with each.
  $\boldsymbol{Q_{same-genre}}$**(ID$_l$, ID$_r$)** $=$ $\boldsymbol{Q_T}$**(ACTOR, SAME-GENRE, ID$_l$, ID$_r$)** where the **SAME-GENRE** edge view between two ACTOR nodes is defined by the following path:

  (a: ACTOR)−[:ACTOR−MOVIE]→(m1:MOVIE)−[:MOVIE−GENRE]→(g: GENRE)
  ←[:MOVIE−GENRE]−(m2:Movie)←[:ACTOR−MOVIE]−(b: ACTOR)

- Given two actors a and b who worked with at least one common director, $Q_{same-director}$ returns the set of actors who worked with at least one common director with each.
  $\boldsymbol{Q_{same-director}}$**(ID$_l$, ID$_r$)** $=$ $\boldsymbol{Q_T}$**(ACTOR, SAME-DIRECTOR, ID$_l$, ID$_r$)**

where **SAME-DIRECTOR** edge view between two ACTOR nodes is defined by the following path:

(a: ACTOR)−[:ACTOR−MOVIE]→(m1:MOVIE)−[:MOVIE−DIRECTOR]→(d: DIRECTOR) ←[:MOVIE−DIRECTOR] −(m2:MOVIE)←[:ACTOR−MOVIE]−(b: ACTOR)

- Given two customers a and b who bought at least a single common item, $Q_{co-purchaser}$ returns the set of customers who bought at least a single common item with each. $\boldsymbol{Q_{co-purchaser}(ID_l, ID_r) = Q_T(\text{CUSTOMER, CO-PURCHASER, } ID_l, ID_r)}$ where the **CO-PURCHASER** edge view between two CUSTOMER nodes is defined by the following path:

(a: CUSTOMER)−[:CUST−ORDER]→(o1:ORDER)−[:ORDER−LINEITEM]→(l: LINEITEM) ←[:ORDER−LINEITEM]−(o2:ORDER)←[:CUST−ORDER]−(b: CUSTOMER)

For $Q_{co-actor}$, we take an early distinct and an early intersect on the MOVIE (m) node. Similarly, the early distinct and early intersect is taken on the GENRE (g), DIRECTOR (d), and LINEITEM (l) nodes for $Q_{same-genre}$, $Q_{same-director}$, and $Q_{co-purchaser}$, respectively.

Query Generation: Each query Q_T in our experiment needs IDs of two nodes a and b where a and b also have an edge view ET_k that connects these two nodes. For our experiment, we first generate the list of (ID_l, ID_r) pair which are the IDs of nodes a and b, respectively. We randomly pick a node `a` of type NT_k from the data graph and then compute all the nodes that are connected with `a` through the edge view ET_k. We then randomly pick at most five of these nodes as `b` and store these (a,b) pairs. For example, for $Q_{co-actor}$, our randomly chosen ACTOR node from the IMDB data graph has ID = 1. Suppose, the node has CO-ACTOR edge to 3 other ACTOR nodes with IDs 2, 4, and 5. Therefore, we store (1, 2), (1, 4), (1, 5) pairs as (ID_l, ID_r) pair for our $Q_{co-actor}$ queries. We do this process for 100 randomly chosen nodes, who has at least one edge view with another node. Table 3.3 shows the number of instances generated for each query Q_T. Some of the randomly chosen nodes have edge view ET_k with more than five nodes. Some of them has less than five.

3.4.2 Comparative Performance Analysis

The query list generated for each query type is executed using three different implementations, default, early distinct, and early intersect, of our EVE/I operator in GraphflowDB as explained in Sections 3.2.3 and 3.3. Recall that in the implementation of early intersect,

Query	# instances	Execution Time (ms)		
		Default	Early Distinct	Early Intersect
$Q_{co-actor}$	375	0.056	0.053	0.019
$Q_{same-genre}$	375	583.04	348.21	93.62
$Q_{same-director}$	439	0.63	0.46	0.17
$Q_{co-purchaser}$	497	2.54	2.31	2.54

Table 3.3: Geometric mean of the execution time (in ms) for different edge view queries

we also take an early distinct on the node where we apply early intersect as shown in Figure 3.4. Therefore, in early intersect, we apply both of these optimizations together. Table 3.3 shows the geometric mean of the runtime (in ms) for different implementations of each of these four different queries. We are reporting the geometric mean as reporting only the mean value can miss the performance of individual queries. Besides, the outliers with long execution times can dominate the run time leading to erroneous conclusions. From Table 3.3, we can see that the geometric mean runtime for the early intersect implementation is the lowest for all three edge views in the IMDB data graph. Early distinct implementation demonstrates on average 1.37x speed up than the default implementation for the IMDB data graph. However, none of the implementations show any significant difference in runtime for the TPC-H data graph.

To further investigate, we list the execution details of some selected queries for default, early distinct, and early intersect implementations in Table 3.4. Note that, P denotes the node on which we apply the early distinct and early intersection optimizations, i.e., P represents the MOVIE (m), Genre (g), DIRECTOR (g), and LINEITEM (l) nodes for $Q_{co-actor}$, $Q_{same-genre}$, $Q_{same-director}$, and $Q_{co-purchaser}$ queries, respectively. In this table, we list the number of P nodes starting from a, denoted as P1, and the number of P nodes starting from b, denoted as P2. Note that, the values of P1 and P2 are distinct for early distinct. Table 3.4 also reports the number of nodes in P=P1 $\cap$ P2, P1'=(P1-P), and P2'=(P2-P). Next, we report the total number of c nodes got from all the nodes in P1 (denoted as c(P1)), P2(denoted as c(P2)), P(denoted as c(P)), P1'(denoted as c(P1')), and P2'(denoted as c(P2')). Lastly, the table include the runtimes for each query.

In the rest of this section, we study the impact and gain of early distinct optimization and early intersect optimization separately by using factor analysis. Starting from the default implementation, we first add the early distinct optimization and analyse the gains. Next, we add the early intersect optimization on top of early distinct and do a similar analysis.

#	Default					Early Distinct					Early Intersect						
	P1	P2	c(P1)	c(P2)	r	P1	P2	c(P1)	c(P2)	r	P	P1'	P2'	c(P)	c(P1')	c(P2')	r
Q1	50	1	700	18	0.19	50	1	700	18	0.15	1	49	0	18	0	0	0.03
Q2	20	26	468	629	0.18	20	26	468	679	0.19	2	18	24	68	401	603	0.24
Q3	6	46	96	523	0.09	6	46	96	523	0.01	1	5	45	12	84	511	0.10
Q4	1	2	81	112	0.034	1	2	81	112	0.035	1	0	1	81	0	0	0.10
Q5	2	17	57	472	0.11	2	17	57	472	0.11	2	0	15	57	0	0	0.016
Q6	4	2	3141k	1303k	481.75	3	2	2053k	1303k	344.04	1	2	0	213k	0	0	21.18
Q7	1	2	59k	170k	27.77	1	2	59k	170k	27.86	1	0	1	59k	0	0	5.47
Q8	2	3	1161k	2064k	465.08	2	3	1161k	2064k	478.78	1	1	2	1088k	73532	975k	284.03
Q9	21	46	14442k	19269k	5255.7	7	12	2820k	3304k	986.858	5	2	7	2488k	332k	816k	429.85
Q10	2	3	1288k	2053k	509.14	2	3	1288k	2053k	541.00	1	1	2	1088k	200k	965k	269.27
Q11	2	25	493	13172	1.05	2	23	493	12092	0.82	1	1	22	477	16	11615	1.38
Q12	1	83	526	21934	12.36	1	63	526	15680	1.35	1	0	62	526	0	0	0.10
Q13	1	16	526	3707	0.59	1	15	526	3533	0.37	1	0	14	526	0	0	0.07
Q14	15	1	6220	235	0.57	15	1	6220	235	0.67	1	14	0	235	0	0	0.10
Q15	34	39	33405	17935	2.59	14	14	6510	5397	0.92	4	10	10	3299	3211	2098	1.00
Q16	71	28	2121	892	2.55	71	28	2121	892	1.86	1	70	27	34	2087	858	1.89
Q17	37	84	1100	2681	1.834	37	84	1100	2681	2.58	1	36	83	30	1070	2651	2.254
Q18	78	58	2389	1723	3.94	78	58	2389	1723	3.65	1	77	57	35	2354	1688	3.65
Q19	37	64	1113	1969	3.67	37	64	1113	1969	2.33	1	36	63	21	1092	1948	2.34
Q20	43	62	1325	1972	1.48	43	62	1325	1972	1.84	1	42	61	34	1291	1938	1.69

Table 3.4: Execution details of the default, early distinct, and early intersect optimizations for selected query instances. Here P1'=(P1-P), P2'=(P2-P), and the columns with header 'r' denotes runtime. Q1-Q5 are type $Q_{co-actor}$, Q6-Q10 are type $Q_{same-genre}$, Q11-Q15 are type $Q_{same-director}$, and Q16-Q20 are type $Q_{co-purchaser}$,

Early Distinct Optimization Analysis

Figure 3.5 shows runtime improvement of the early distinct implementation over the default implementation. We can see that for majority of the query instances, the improvement is within 1-1.5x for all four type of queries. For $Q_{same-genre}$ and $Q_{same-director}$, the runtime has up to 5x improvement. However, we can see a number of queries also has less than 1x improvement. When there is no repetition in P1 and P2, early distinct does not show any benefit over the default. For example, in Table 3.4, for queries Q7, Q8, Q10, and Q14, the number of P1 and the number of P2 are same for both early distinct and default. Therefore, they both executes the same number of tuples and hence have similar execution time. However, computing distinct may add a little overhead than the default implementation when computing distinct does not reduce the number of nodes. For example, early distinct does not reduce the number of nodes in Q2, Q7, Q8 and Q10. For these queries, early distinct shows slight increase in runtime than the default. On the other hand, we can see a lot of reduction in the tuple execution as well as runtime if either P1 or P2 has repetitive

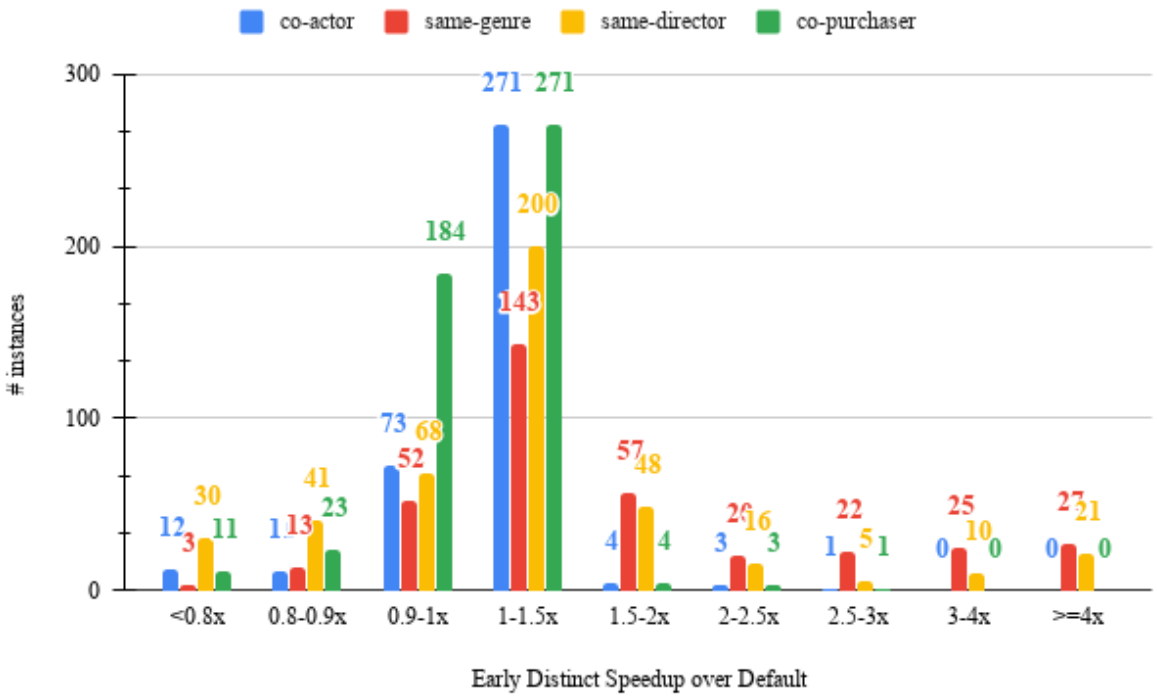


Figure 3.5: Query run time comparison between Default and Early Distinct. This is a histogram of number of queries with different amount of speed-ups or slow downs. Each bar is associated with an improvement range (l,r) and the height of the bar is the number of queries on which early distinct improves the default implementation within a factor k, such that l≤k<r.

values. Even with 1 repeated genre in Q6, we see a 1.4x speed-up for early distinct over the default. Early distinct exhibits geometric mean speed-up of 1.7x for $Q_{same-genre}$, geometric mean speed-up of 1.4x for $Q_{same-director}$.

For $Q_{co-purchaser}$ queries, early distinct performs relatively worse than the other three type of queries. 184 query instances of $Q_{co-purchaser}$ has 0.9-1.0x improvement for early distinct implementation over the default. This is because the TPC-H dataset has an uniform distribution of the lineitems and does not have any repetitive characteristic where a customer buys same product at least twice. For such datasets, early distinct does not have any significant effect. Table 3.4 shows similar performance for $Q_{co-purchaser}$ queries. For all $Q_{co-purchaser}$ queries, Q16-Q20, in Table 3.4, default and early distinct has same

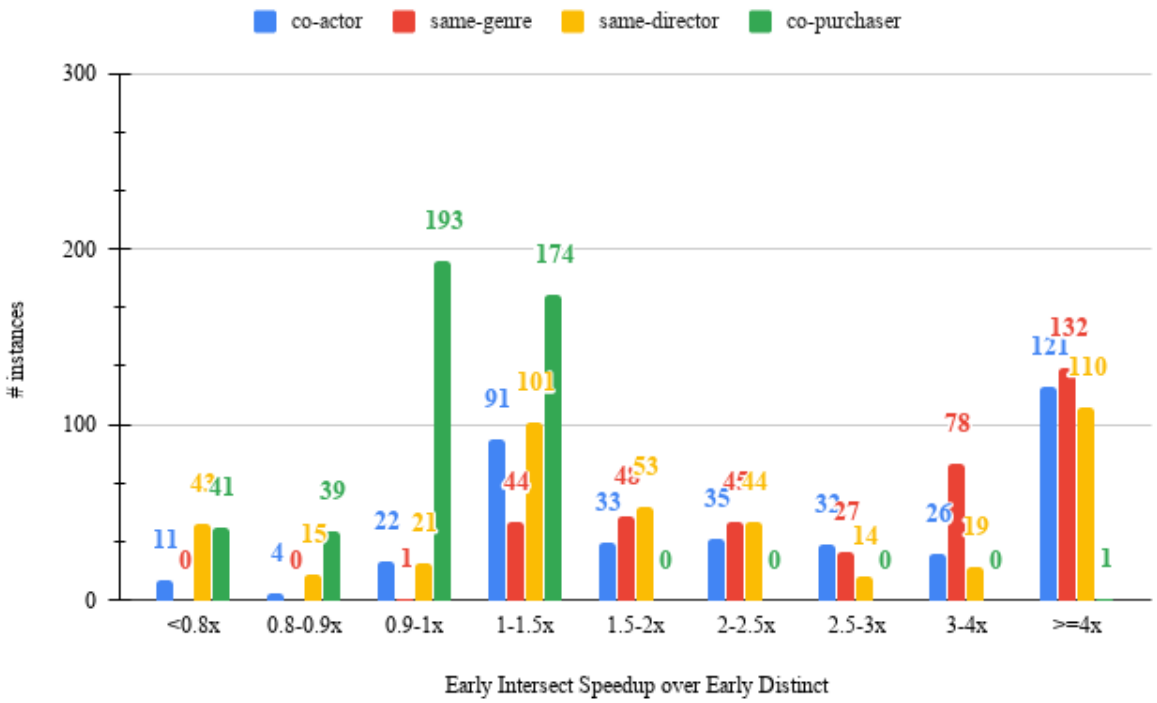


Figure 3.6: Query run time comparison between Early Distinct and Early Intersect. This is a histogram of number of queries with different amount of speed-ups or slow downs. Each bar is associated with an improvement range (l,r) and the height of the bar is the number of queries on which early intersect improves the early distinct implementation within a factor k, such that l≤k<r.

number of tuples executed and the execution time does not show significant difference. $Q_{co-actor}$ queries also shows a similar characteristics. However, $Q_{co-actor}$ has fewer number of queries that has less than 1x performance for early distinct than for $Q_{co-purchaser}$. Early distinct shows geometric mean speed-up of 1.06x for $Q_{co-actor}$ and geometric mean speed-up of 1.09x for $Q_{co-purchaser}$.

Early Intersect Optimization Analysis

The performance improvement of early intersect over early distinct is showed in Figure 3.6. We can see that, except for the $Q_{co-purchaser}$ queries, the majority number of queries has

greater than 1.5x improvement. From the execution details listed in Table 3.4, we see that for queries when either (P1-P) or (P2-P) has an empty set, the performance improves more than 3x for early intersect as this strategy discards the expansion of (P1-P) and (P2-P) completely. For example, Q1, Q4, Q4, Q6, Q12 and Q13 demonstrate such optimizations. When none of these sets are empty, early intersect still shows significant performance improvement for queries $Q_{same-genre}$, $Q_{same-director}$, and $Q_{co-actor}$. $Q_{co-purchaser}$ does not shows any significant difference in performance for early intersect over early distinct. This is because all the $Q_{co-purchaser}$ queries in Table 3.4 has P=1 and larger number of nodes in P1‘ and P2‘. Hence, early intersect does not have any benefits over early distinct. Q16-Q20 have either equal performance of early distinct and early intersect or a slight decrease in performance for early intersect over early distinct. Overall, early intersect shows geometric mean speed-up of 2.7x for $Q_{co-actor}$, geometric mean speed-up of 3.07x for $Q_{same-genre}$, geometric mean speed-up of 2.7x for $Q_{same-director}$, and geometric mean speed-up of 0.9x for $Q_{co-purchaser}$.

Chapter 4

Related Work

In this chapter, we review the related work for extracting graphs from relational tables and explore the literature related to the RDBMS-GDBMS synchronization process.

4.1 Extracting Graphs from RDBMS

Extracting and querying graph data that primarily resides in relational databases has recently gained popularity among enterprise users to perform various sophisticated tasks, notably among others, fraud and threat detection in e-commerce and financial applications (Alibaba [32]), social network search (LinkedIn's economic graph [6], Facebook's graph search [5]), and recommendations and promotions (such as in eBay [13] and Walmart [13]). Supporting the existing interest in graph analytics, researchers proposed and built several techniques [10, 36] and systems [40, 19, 11, 39] to access graphs in relational databases. Many systems (e.g., GraphGen [40], Vertexica [19], GRAIL [11]) perform graph analytics using relational databases by defining a layer above the database to convert the graph queries into SQL. In Vertexica and GRAIL, the graph queries are mapped into relational operators and the graph is stored in the relational table, whereas GraphGen stores a graph representation in-memory and lets users extract the hidden graph from relational tables using a Datalog-like DSL. However, the goal of GraphWrangler is fundamentally different from these three systems as GraphWrangler aims to generate graphs from relational tables so that the graphs can be inserted into any graph system later for performing graph analytics rather than converting graph queries into relational operators.

Existing GDBMSs [27, 31] provide mechanisms to import graph data from RDBMSs. For example, Neo4J developed an ETL Tool that allows users to directly import graphs

from RDBMSs using a JDBC connection and an interface to interactively define a relational to graph transformation [28]. The interface lets users define nodes and edges by selecting the relations from which the nodes and edges to be extracted. The ETL tool assumes that edges of a specific type are extracted from a single relation in the RDBMS. Another system, GraphBuilder [39], supports the graph extractions from unstructured data using a MapReduce framework. However, the users still need to write Java codes to specify the graphs to be extracted in GraphBuilder. Neither Neo4j nor GraphBuilder provide any information about the underlying data interconnection in the RDBMS and visualize the snippet of the transformed graph before actually extracting the graph from the entire RDBMS. Furthermore, GW's data transformation language supports edge types that are extracted by joining multiple relations, unlike the graph import services provided by existing GDBMSs.

Several works also focus on streamlining the ETL pipeline that requires users to write scripts to extract a specific graph [36, 10, 22]. Sequeda et. al. provides a formal solution in [36] to direct map RDBMS into RDF and OWL using the relational schema and integrity constraints. The RDF database can be directly mapped to property graph databases [1]. The authors in [10] propose an automatic approach to migrate RDBMS into general graph data, which also supports query translation. While these approaches relieve the users from writing scripts, these methods also restrict users from exploring different graph structures from a single RDBMS. For example, from a set of product, customer, and order relations, one may extract a graph having only customer type nodes and an edge between two customers shows both customers living in the same city. Another graph may contain product and customer nodes showing the purchase history. Direct mapping only supports a single type of graph depending on the mapping language. Table2Graph system provides a solution to this problem by allowing graph extraction from an XML mapping, which is configurable for reuse [22]. However, unlike GW, none of these systems support any interactive capability to give users the freedom to explore and extract a particular graph out of relational tables. Ploceus provides an interactive framework for users to define graphs from tabular data (e.g., RDBMS, CSV files), giving a visual analysis approach of tabular data exploiting the underlying network connection [23]. However, Ploceus does not support edge types that require join among three or more relations and any node or edge properties that require aggregation. The main motivation of Ploceus is visual exploration, not data transformation. Besides supporting a wide range of edge types and aggregation properties, GW helps users to define graphs using a predictive interaction framework and generate an automated script to extract and transform the entire graph from the source RDBMS into a GDBMS.

4.2 RDBMS-GDBMS Synchronization

Several systems have been proposed to support synchronization of extracted graphs with the updates in the source database [17, 29]. A recent system, SQL2RDF, incrementally update an RDF store on RDBMS changes using the R2Rml [17]. Neo4j Streaming Data allows users to integrate updates in RDBMS and other databases using a Kafka broker [29]. The users publish all the changes of their RDBMS to a Kafka, which is subscribed to the Neo4j streaming plugin. However, the users need to manage the Kafka system, and this system relies on getting node and edge data coming from specific relations. Therefore, edge definitions that use join among multiple relations are not supported using this method. To the best of our knowledge, there is no automated system to support the continuous synchronization of graphs that involves node and edge extractions using joins among multiple relations. R2GSync is different than these existing approaches as it automates the continuous synchronization process of diverse extracted graphs from an RDBMS that involves node and edge extractions using joins among multiple relations. Furthermore, R2GSync relieves the users from managing any intermediate systems like Kafka.

Keeping a graph in a GDBMS in sync with the updates in an RDBMS from where the graph is extracted is a process analogous to incremental maintenance of materialized views [7], which integrates updates of database changes incrementally in the materialized views [3]. Several studies focus on IVM for materialized views in RDBMS [34, 35, 24] and in GDBMS [37, 41]. The views in our system are node and edge definitions supported by GW that are specific projection queries in set semantics where we compute distinct end nodes of a path query. We keep views from an RDBMS materialized in a GDBMS. A recent study developed a solution to optimize query execution for graph analytics using graph views [9]. Graph views are similar to our node and edge views as a graph view stores the nodes and the multi-hop path relation connecting two nodes as edges. In this thesis, we propose a node and edge view-based approach to support the incremental changes in node and edge definition queries to a property graph database.

Chapter 5

Conclusions and Future Work

GDBMSs gained popularity due to their performance benefits and ease of use in analyzing highly connected application datasets which, in many enterprises, are primarily stored as relational tables in an RDBMS. In this thesis, we studied the challenges of extracting graphs out of relational tables and keeping a transformed graph in a GDBMS in sync with a source RDBMS. First, we described the GW system that we developed to interactively extract graphs from an RDBMS. In addition to streamlining the ETL pipeline, GW allows users to get a graph view on their relational tables before transforming the entire database. We also explained GW's predictive interaction framework and a set of techniques to maintain interactive speed.

Next, we explored three possible designs for RDBMS to GDBMS synchronization and developed a system, R2GSync, adopting the node and edge view design. R2GSync keeps graphs in a GDBMS synchronized with an RDBMS, from which the graphs are initially extracted. We described the concept of edge views, our edge view implementation inside GraphflowDB, and proposed two optimizations over the default implemention of EdgeViewE/I operator. We showed that for datasets where an entity is connected to another through multiple paths, both early distinct and early intersect optimizations perform significantly better than the default implementation. For example, for the IMDB dataset, early distinct has on average 1.37x speed up over the default and early intersect has on average 3.1x speed up over the early distinct implementation.

Noticeably missing from R2GSync is the support of the node and edge aggregation properties and node definitions that require node views for continuous synchronization. Currently, R2GSync's support for node properties is limited to the attribute values of a node relation. In the future, we want to support a wide range of node and edge types and

properties on node and edge views. Another future work is to extend the functionalities of GW and R2GSync systems for non-property graph stores such as RDF triple stores. In addition to an RDBMS database, we also plan to support extracting graphs from CSV files and non-relational data stores such as NoSQL databases.

References

[1] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. Directly mapping rdf databases to property graph databases. *arXiv*, pages arXiv–1912, 2019.

[2] Nafisa Anzum, Semih Salihoglu, and Daniel Vogel. Graphwrangler: An interactive graph view on relational data. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1865–1868, 2019.

[3] Jose A Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. *ACM SIGMOD Record*, 15(2):61–71, 1986.

[4] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-driven Documents. *TVCG*, 17(12), 2011.

[5] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. {TAO}: Facebook's distributed data store for the social graph. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 49–60, 2013.

[6] Xi Chen, Yiqun Liu, Liang Zhang, and Krishnaram Kenthapadi. How linkedin economic graph bonds information and product: applications in linkedin salary. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 120–129, 2018.

[7] Rada Chirkova and Jun Yang. Materialized views. *Databases*, 4(4):295–405, 2011.

[8] MySQL Connector/Node.js. https://dev.mysql.com/downloads/connector/nodejs/8.0.html.

[9] Joana MF da Trindade, Konstantinos Karanasos, Carlo Curino, Samuel Madden, and Julian Shun. Kaskade: Graph views for efficient graph analytics. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 193–204. IEEE, 2020.

[10] Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. Converting relational to graph databases. In *First International Workshop on Graph Data Management Experiences and Systems*, pages 1–6, 2013.

[11] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[12] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445, 2018.

[13] José Guia, Valéria Gonçalves Soares, and Jorge Bernardino. Graph databases: Neo4j analysis. In *ICEIS (1)*, pages 351–356, 2017.

[14] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 7(13):1379–1380, 2014.

[15] Jeffrey Heer, Joseph M. Hellerstein, and Sean Kandel. Predictive Interaction for Data Transformation. In *CIDR*, 2015.

[16] Imdb datasets. https://www.imdb.com/interfaces/.

[17] Sql2rdf: pump sql dml immediately to rdf triplestore. http://inova8.com/bg_inova8.com/sql2rdf-pump-sql-dml-immediately-to-rdf-triplestore/, 2018.

[18] interact.js. http://interactjs.io/.

[19] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. Vertexica: your relational friend for graph analytics! *Proceedings of the VLDB Endowment*, 7(13):1669–1672, 2014.

[20] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *CHI*, 2011.

[21] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An Active Graph Database. In *SIGMOD*, 2017.

[22] Sangkeun Lee, Byung H Park, Seung-Hwan Lim, and Mallikarjun Shankar. Table2graph: A scalable graph construction from relational tables using map-reduce. In *2015 IEEE First International Conference on Big Data Computing Service and Applications*, pages 294–301. IEEE, 2015.

[23] Zhicheng Liu, Shamkant B Navathe, and John T Stasko. Network-based visual analysis of tabular data. In *2011 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 41–50. IEEE, 2011.

[24] Gang Luo, Jeffrey F Naughton, Curt J Ellmann, and Michael W Watzke. A comparison of three methods for join view maintenance in parallel rdbms. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*, pages 177–188. IEEE, 2003.

[25] Materialized. https://materialize.io/.

[26] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324, page 36, 2013.

[27] Neo4j graph platform. https://neo4j.com/.

[28] Neo4j etl tool 1.3.1 release: White winter. https://medium.com/neo4j/neo4j-etl-tool-1-3-1-release-white-winter-2fc3c794d6a5, 2019.

[29] Streaming graphs: Combining kafka and neo4j. https://neo4j.com/blog/streaming-graphs-combining-kafka-neo4j/, 2019.

[30] Mark EJ Newman. Detecting community structure in networks. *The European physical journal B*, 38(2):321–330, 2004.

[31] Orientdb. https://www.orientdb.org/, 2019.

[32] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 11(12):1876–1888, 2018.

[33] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey. https://cs.uwaterloo.ca/~ssalihog/papers/graph-survey-extended.pdf, 2019.

[34] Kenneth Salem, Kevin Beyer, Bruce Lindsay, and Roberta Cochrane. How to roll a join: Asynchronous incremental view maintenance. *ACM SIGMOD Record*, 29(2):129–140, 2000.

[35] Abderrazak Sebaa and Abdelkamel Tari. Materialized view maintenance: issues, classification, and open challenges. *International Journal of Cooperative Information Systems*, 28(01):1930001, 2019.

[36] Juan F Sequeda, Marcelo Arenas, and Daniel P Miranker. On directly mapping relational databases to rdf and owl. In *Proceedings of the 21st international conference on World Wide Web*, pages 649–658, 2012.

[37] Gábor Szárnyas. Incremental view maintenance for property graph queries. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1843–1845, 2018.

[38] The tpc benchmark h. http://www.tpc.org/tpch/.

[39] Theodore L Willke, Nilesh Jain, and Haijie Gu. Graphbuilder–a scalable graph construction library for apache™ hadoop™. *Big Learning WS at NIPS*, 2012.

[40] Konstantinos Xirogiannopoulos, Udayan Khurana, and Amol Deshpande. Graphgen: Exploring interesting graphs in relational data. *Proceedings of the VLDB Endowment*, 8(12):2032–2035, 2015.

[41] Yue Zhuge and Hector Garcia-Molina. Graph structured views and their incremental maintenance. In *Proceedings 14th International Conference on Data Engineering*, pages 116–125. IEEE, 1998.