ABSTRACT

Title of Dissertation:	BEYOND PROPERTY GRAPHS: DEVELOPMENT AND APPLICATIONS OF A HYBRID GRAPH ANALYTICS SYSTEM
	Alexandria Barghi Doctor of Philosophy, 2025
Dissertation Directed by:	Professor Manoj Franklin Department of Electrical and Computer Engineering

Over the past two decades, graph analytics has grown from a fairly niche discipline into a ubiquitous field touching nearly every aspect of modern computing. This growth began with the internet, which led to the standardization of OWL and RDF, and was accelerated by social media and large-scale e-commerce, which produced a slew of content that could easily be represented as a graph. Along the way, *graph databases* emerged as an alternative to relational databases, and *graph processing systems* were conceived to accelerate graph algorithms and process *graph queries*. Today, graphs are used to solve a huge variety of problems across domains, from recommendation systems that serve targeted advertisements, to bank and insurance claim fraud detection, to customer churn prediction, to molecular and genomic analysis.

We now stand at a new frontier of graph analytics, at the intersection of traditional graph analytics, the rapidly growing technology of *graph neural networks (GNNs)*, and large language models. These technologies are the key to unlocking the power of generative artificial intelli-

gence, delivering more accurate, complete, up-to-date, and hallucination-free models through the process of *retrieval augmented generation (RAG)*. However, the sheer size and complexity of enterprise-scale graphs makes deploying a graph-based RAG difficult. Much of the existing technology in these areas was not designed to work together, or to work at the scale needed by a large enterprise.

In this dissertation, I will discuss the state of graph analytics, how it can be applied to RAG, and the challenges currently preventing universal adoption of graph-based RAG. I will then introduce my work, the *BitGraph* framework, which provides the keys to unlocking GPU-accelerated graph-based RAG.

In the first component of my dissertation, I will start by describing the core construction of the BitGraph framework and its subcomponents, which include the *Gremlin++* query language and *Maelstrom*, a lightweight backend for accelerating vector operations on both the CPU and GPU. I will show how the BitGraph framework is constructed in layers, with Maelstrom as the bottom layer, Gremlin++ as the middle layer, and BitGraph as the top layer, and how this construction makes the framework extremely versatile and enables acceleration of up to 35x over an equivalent naive CPU implementation.

In the second component of my dissertation, I will analyze and discuss handling and processing graph queries at scale, focusing on the theory behind query acceleration and how it applies to the paradigms and data structures used in the BitGraph framework. I will then show how specific types of query optimizations, called *traversal strategies*, work together to perform end-to-end just-in-time optimization to deliver 100% speedup on a simple query, and 40% speedup on a more advanced query.

In the third and final component of my dissertation, I will show how to use BitGraph and

its query language, *Gremlin++*, to solve a large-scale RAG problem. I will also discuss how I further extended the BitGraph framework to include support for vertex embeddings, which were critical in developing what I call *prize-aware graph traversal*, a type of graph traversal that starts from a set of initial vertices and greedily selects additional vertices in the neighborhood through embedding comparison.

BEYOND PROPERTY GRAPHS:
DEVELOPMENT AND APPLICATIONS OF
A HYBRID GRAPH ANALYTICS SYSTEM

by

Alexandria Barghi

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2025

Advisory Committee:
Professor Manoj Franklin, Chair/Advisor
Professor Donald Yeung
Professor Shruvya Bhattacharyya
Professor Cunxi Yu
Professor Ramani Duraiswami, Dean's Representative

© Copyright by
Alexandria Barghi
2025

Dedication

To Emily: You kept the dream alive.

Acknowledgments

First, I would to start by thanking my advisor, Professor Manoj Franklin, who has guided me through my academic career since I was an undergraduate student. I have definitely not always been an easy student to advise, but he has consistently been patient and understanding throughout my time at UMD.

Next, I want to thank Emily, for everything she has done to support me over the past two years. I could not have done this without her support, and I am eager to finally have more time for us to spend together.

I would also like to thank my father, the first Dr. Barghi, for encouraging me to embark on this long journey, and supporting me throughout my life, as well as the rest of my family, for their unconditional support as well.

Finally, as a student who has also been a working professional in industry for several years now, I could not be here without the support and encouragement from my managers and coworkers. I would like to call out Dr. Bradley Rees and Dr. Lauren Kennell for being especially supportive of my PhD work.

Table of Contents

Dedication ii

Acknowledgements iii

Table of Contents iv

List of Tables vii

List of Figures viii

List of Abbreviations x

Chapter 1: Introduction 1
- 1.1 Overview 1
- 1.2 Traditional Graph Analytics 2
  - 1.2.1 Structural Graphs 2
  - 1.2.2 Property Graphs 3
  - 1.2.3 Graph Algorithms 5
  - 1.2.4 Graph Databases and Query Languages 5
- 1.3 Graph Processing Systems 8
- 1.4 Graph Neural Networks 8
- 1.5 Large Language Models 10
- 1.6 Main Contributions 11

Chapter 2: Background and Previous Work 13
- 2.1 Background and Motivation 13
- 2.2 Previous Work 16
  - 2.2.1 BitGraph-2019 16
  - 2.2.2 Other Work 17

Chapter 3: A Framework for GPU-Accelerated Graph Query Processing 18
- 3.1 Overview 18
- 3.2 Related Work 18
  - 3.2.1 Gunrock 18
  - 3.2.2 cuSTINGER and Hornet 20
  - 3.2.3 Blazing SQL and Dask-SQL 20
  - 3.2.4 Lotan 21

3.2.5 OLAP Gremlin/Spark-Gremlin . . . 21
3.2.6 Gremlin Server and Drivers . . . 22
3.3 Designing Maelstrom . . . 22
3.3.1 Overview and Motivation . . . 22
3.3.2 The Maelstrom API . . . 25
3.4 Designing Gremlin++ . . . 27
3.4.1 Overview . . . 27
3.4.2 Structure of a Query . . . 27
3.4.3 The Traverser Abstraction . . . 28
3.4.4 Life of a Query . . . 29
3.5 Designing BitGraph . . . 32
3.5.1 Overview . . . 32
3.5.2 Storage and Access of the Graph Structure . . . 32
3.5.3 Storage and Access of Vertex and Edge Properties . . . 34
3.6 Performance Analysis . . . 34
3.6.1 Overview . . . 34
3.6.2 Connected Components Benchmarks . . . 35
3.6.3 Temporal Shortest Paths . . . 37
3.7 Discussion . . . 40

Chapter 4: Graph Query Optimizations for Machine Learning and RAG Use Cases 42
4.1 Overview and Motivation . . . 42
4.2 Related Work . . . 43
4.2.1 Banyan and GOpt . . . 43
4.2.2 GraphScope . . . 43
4.2.3 BitGraph Framework . . . 45
4.2.4 Java-Gremlin . . . 45
4.3 How Graph Queries are Used . . . 46
4.4 Prerequisites to Optimization . . . 49
4.5 The Optimization Process . . . 49
4.5.1 Overview . . . 49
4.5.2 Fusion . . . 52
4.5.3 Reordering . . . 54
4.5.4 Loop Unrolling . . . 55
4.5.5 Pattern Detection . . . 55
4.5.6 Backend-Specific Optimizations . . . 57
4.5.7 Overall Impact . . . 59
4.6 RAG Case Study . . . 60
4.6.1 Overview . . . 60
4.6.2 Benchmark Description . . . 62
4.6.3 Benchmark Results . . . 62
4.7 Discussion . . . 66

Chapter 5: Development and Performance of a Query-Based Graph RAG 67
5.1 Overview . . . 67

5.2 Motivation 67
5.3 Background and Related Work 68
5.3.1 Retrieval Augmented Generation 68
5.3.2 G-Retriever 72
5.3.3 HotpotQA and 2WikiMultihopQA 74
5.3.4 FAISS 76
5.4 High-Level Design and API 77
5.5 Incorporating Vector Indexing 81
5.5.1 Memory Usage and Behavior of Vector Search 81
5.5.2 Incorporating FAISS into BitGraph 82
5.6 Assembling QRAG 83
5.6.1 The QRAG Process 83
5.6.2 Query Tuning 85
5.7 Performance Analysis 85
5.7.1 The Knowledge Graph 85
5.7.2 Model and Training Pipeline Structure 87
5.7.3 Benchmark Results 91
5.7.4 Opportunities for Improvement 92
5.8 Discussion 94

Chapter 6: Conclusion and Future Work 95
6.1 Conclusion 95
6.2 Future Work 97
6.2.1 Beyond PCI-e 97
6.2.2 Alternatives to FAISS 98
6.2.3 Multi-GPU Processing 98
6.2.4 Other Query Languages 99

List of Tables

3.1 Connected Components Benchmark Datasets 36
3.2 Logistics Pathfinding Problem 38
3.3 Temporal Shortest Paths Benchmark Datasets 39

5.1 QRAG Benchmark Models 88

List of Figures

1.1 "The Crew", a property graph created by the Apache TinkerPop team for illustration purposes . . . 4
1.2 Graph schema and corresponding query used to illustrate graph query language syntax, adapted from [23] . . . 7
1.3 Example of a SPARQL query to find films matching the criteria described in Figure 1.2. . . . 7
1.4 Example of a Cypher query to find films matching the criteria described in Figure 1.2. . . . 7
1.5 Example of a Gremlin query to find films matching the criteria described in Figure 1.2. . . . 8
1.6 The Graph Analytics Process . . . 9
1.7 Convolutional Neural Networks (CNNs) vs. Graph Neural Networks (GNNs) . . 10

3.1 Illustration created by the Apache TinkerPop team that shows the relationship between Gremlin drivers, Gremlin server, the query processing backend, and the upstream database . . . 23
3.2 A basic Gremlin query . . . 28
3.3 Comparison of Gremlin++ and Java-Gremlin traversals . . . 30
3.4 Life of a Gremlin query in the BitGraph Framework . . . 31
3.5 Gremlin query for the connected components algorithm . . . 35
3.6 Connected Components Benchmark Results . . . 37
3.7 Graph of a logistics network . . . 38
3.8 Gremlin query for the temporal shortest paths problem . . . 39
3.9 Temporal Shortest Paths Benchmark Results . . . 40

4.1 Scope application in Gremlin++ . . . 44
4.2 The dispatching process . . . 48
4.3 Translation of a Gremlin query to the immediate representation (traversal steps) . 51
4.4 Final step representation of the Gremlin query from Figure 4.3 . . . 52
4.5 Application of HasJoinStrategy to the query shown in Figure 4.3 . . . 53
4.6 Application of RepeatUnrollStrategy to the query shown in Figure 4.3 . . . 56
4.7 Application of BasicPatternExtractionStrategy to the query shown in Figure 4.3 . 58
4.8 Application of BitGraphStrategy and BitGraphSelectionStrategy to the query shown in Figure 4.3 . . . 59
4.9 Cumulative speedup of each traversal strategy (optimization) on the query from Figure 4.3 . . . 61
4.10 Workflow diagram showing a direct input graph-based RAG . . . 63

4.11 Basic parameterized query that retrieves additional documents from a set of initial documents (vertices) as part of a direct input RAG 64
4.12 Simple RAG Query Benchmark Results 65

5.1 Illustration of a direct-input graph-based RAG (DGR) using data from 2WikiMultihopQA [83] 71
5.2 Illustration of a combined graph-based RAG (CGR) using data from 2WikiMultihopQA [83] 73
5.3 The Encode Step in Gremlin++ 78
5.4 The Like Step in Gremlin++ 80
5.5 Speedup over the non-indexed seed selection step delivered by BitGraph's IVFPQ FAISS indexes 84
5.6 End-to-end QRAG process 86
5.7 Sample knowledge graph resembling the knowledge graph constructed from 2WikiMultihopQA [83] 87
5.8 Stages 1-3 of the QRAG training pipeline 89
5.9 The fourth and fifth stages of the QRAG training pipeline 90
5.10 Mean embedding difference of each model 93

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
ANNS	Approximate Nearest-Neighbor Search
API	Application Programming Interface
BSP	Bulk-synchronous Parallel
CGR	Combination Graph-based RAG
CNN	Convolutional Neural Network
COO	Coordinate List
CPU	Central Processing Unit
CSC	Compressed Sparse Column
CSR	Compressed Sparse Row
DBMS	Database Management System
DGR	Direct-input Graph-based RAG
ETL	Extract, Transform, Load
GAT	Graph Attention Network
GNN	Graph Neural Network
GPU	Graphics Processing Unit
IVF	Inverted File
IVFPQ	Inverted File with Product Quantization
LLM	Large Language Model
ML	Machine Learning
NER	Named Entity Recognition
OLAP	Online Analytical Processing
OLTP	Online Transactional Processing

PCST	Prize-Collecting Steiner Tree
PQ	Product Quantization
RAG	Retrieval Augmented Generation
RDF	Resource Description Framework
SQL	Structured Query Language
UVA	Unified Virtual Addressing
UVM	Unified Virtual Memory

Chapter 1: Introduction

1.1 Overview

Graph analytics is one of the most dynamic subfields in data analytics, becoming ubiquitious over the course of the past two decades [1]. Long-gone is the era where a typical graph can be drawn on paper - real-world graphs are rapidly approaching trillion-edge scale [1]. Analyzing, visualizing, and manipulating these networks is a complex problem not just in data storage, but also in software and systems engineering. Furthermore, as graph analytics is applied to new problems, such as those related to large language models (LLMs) [2, 3], graph analytics systems have to support an increasingly broad set of tools.

Tabular data analysis has long been dominated by query languages [4, 5, 6, 7, 8], which are easy to apply to a broad set of problems, and lend themselves well to integration with downstream machine learning models. Graph query languages [9, 10, 11, 12, 13] aim to solve this problem for graphs, but there are many problems unique to graphs that make them difficult to apply to large-scale graphs. In this dissertation, I will focus on these unique problems and discuss how my own software and systems engineering project, the BitGraph framework, helps solve these problems and make graph query languages much more useful and powerful.

1.2 Traditional Graph Analytics

1.2.1 Structural Graphs

The field of graph analytics began with Leonhard Euler in the 18th century. It began as a means of solving a specific problem, and grew from there into a complete field of mathematics. Euler's definition of a graph describes what we would now refer to as a *structural graph*. Formally, a structural graph is defined as:

$$G = (V, E) \tag{1.1}$$

V refers to a set of vertices, and *E* refers to a set of edges where each edge

$$e = (u, v); u, v \in V \tag{1.2}$$

Since the twentieth century, structural graphs have been applied to a variety of problems outside theoretical mathematics, including engineering, computer science, and social sciences. A typical data structures and algorithms course now covers basic graph theory, and expands it to discuss how graphs are stored and processed, either as *adjacency lists*, or *adjacency matrices*. The ideal format depends on the algorithms to be run on the graph, the size of the graph, and the sparsity of the graph [14].

1.2.2 Property Graphs

In the early twenty-first century, the property graph emerged as a new model that extended the structural graph created by Euler. There are a variety of formulae used to describe property graphs, but the most convenient for discussion here adds a set of functions

$$F(x); x \in V \cup E \tag{1.3}$$

Each function f in the set of functions transforms a vertex or edge into a numerical, symbolic, or string value;

$$f(x) \in F \to y \in U \tag{1.4}$$

U represents the universal set. This produces the formal definition

$$G = (V, E, F) \tag{1.5}$$

for a property graph. From this definition, we can define structural graphs as degenerate cases of property graphs, where

$$G = (V, E, F); F = \emptyset \tag{1.6}$$

In simpler terms, a property graph is a structural graph where vertices and edges can have *properties*. These properties can be numerical, symbolic, or string values. This is how most graph databases describe their model, rather than using the mathematical description from the previous paragraph. This dissertation will also mostly describe property graphs in these terms. Figure 1.1 shows an example of a property graph.

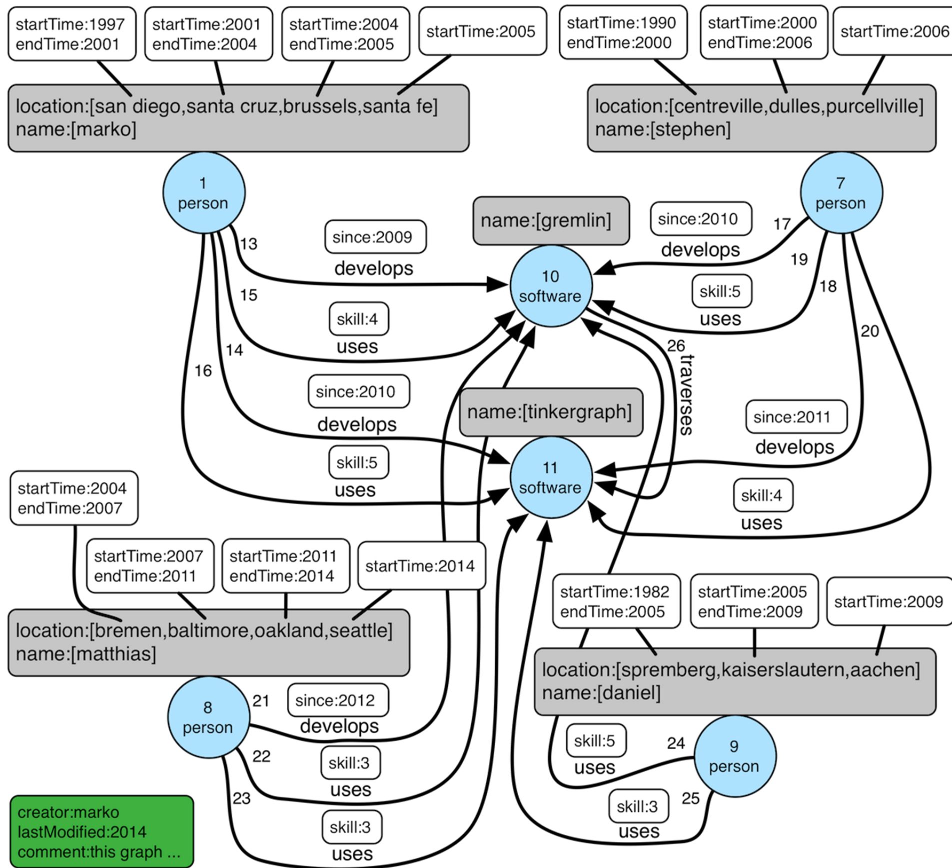


Figure 1.1: "The Crew", a property graph created by the Apache TinkerPop team for illustration purposes [15]. This graph contains six vertices and fourteen edges. Vertices and edges can contain zero or more properties of any name. Different vertices and edges can have properties with the same name; the vertex or edge is what differentiates them. For instance, consider the "location" property. It is present on vertices 1, 7, 8, and 9. In this graph, "location" represents the locations where a person lives. Vertices 10 and 11 do not have this property because they are not people; they are software. Whether a vertex is a “person” or a “software” is defined by the vertex label. Vertex labels segment vertices into different types, and are very useful when performing indexing. They correspond well to vertex types in GNNs. Edge labels, such as “develops” and “uses” serve the same function for edges. All vertices and edges in the graph must have at most one label (no label is also permissible). The label can be thought of as a special property. This graph also illustrates multiproperties and metaproperties, which define multiple values of a single property over a given time. Most property graph implementations, including most Gremlin backends, do not support multiproperties or metaproperties.

1.2.3 Graph Algorithms

Graph algorithms are defined as operations on a graph. Simple graph algorithms include breadth-first and depth-first search [14], connected components [14, 16], and PageRank [17]. There are three main categories of graph algorithms: *centrality algorithms*, *connectivity algorithms*, and *traversal algorithms*. Centrality algorithms include PageRank, eigenvector centrality, betweenness centrality [18], and Katz centrality [19]. Connectivity algorithms include connected components, bipartite graph matching [14], and minimum spanning tree [14]. Traversal algorithms include breath-first and depth-first search [14] and neighborhood sampling [20]. Various implementations and variations of these algorithms exist, specialized for certain versions of the problem, computation engines, or data structures. For decades, improving and accelerating graph algorithms has been a major research area within the field of computer science.

1.2.4 Graph Databases and Query Languages

1.2.4.1 Overview

Graph databases are databases where data is natively stored and accessed in a graph format. Nearly all graph databases represent property graphs, and store these properties along with the graph structure [21, 22, 23]. Examples of graph databases include Neo4j [24], TigerGraph [25], and Amazon Neptune [26]. Graph databases, like traditional SQL databases, have an associated query language that allows users to interact with and extract data from the graph [22, 23]. Graph query languages include Cypher [9], Gremlin [10], GQL [12], GSQL [11], and SPARQL [13]. Unlike SQL, there is no standard graph query language, though there have been some efforts to

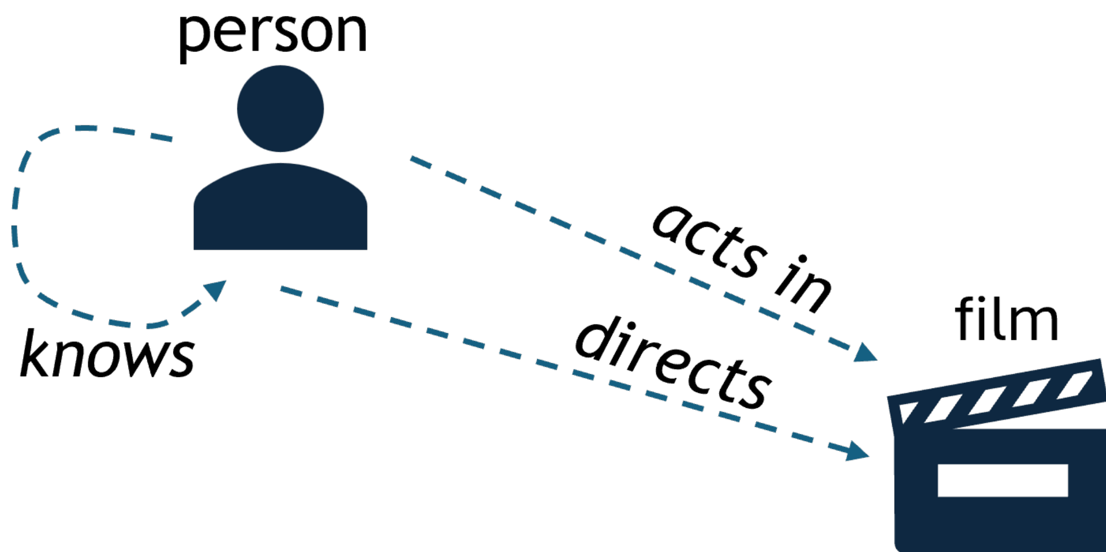
construct one [12, 13, 21].

1.2.4.2 Query Semantics

Graph query languages, like their SQL-based relatives, are predominantly declarative languages [23, 4]. Unlike SQL-based languages, which don't contain a complete set of control flow instructions [4], graph query languages often do incorporate instructions for control flow [9, 10] or even functional components [10, 23]. Gremlin, in particular, is quite complex, and is arguably its a complete functional programming language [27] on its own [10, 23]. Functional programming languages, like Gremlin, are excellent for decomposition into a series of mathematical operations [27], making them ideal for acceleration through *bulk-synchronous parallel* (BSP) execution [28]. This is described in more detail in Chapter 3.

There are many graph query languages, such as *Cypher* [9], *Gremlin* [10], *GSQL* [11], *SPARQL* [13], and *GQL* [12]. Queries written in one language can often be translated to another through compilation or transpilation [29, 30]. Figure 1.3 shows an example of a query written in SPARQL. Figure 1.4 shows the same query written in Cypher, and Figure 1.5 shows the same query written in Gremlin.

While the syntax of graph query languages can differ significantly, nearly all possess two fundamental features: *pattern matching* and *navigational querying* [23]. Pattern matching is the process of first understanding a user-specified pattern, then transforming the pattern into a series of steps to be executed by the query engine, and finally, returning entities in the graph matching the pattern [23]. Cypher primarily focuses on pattern matching, with most of its syntax serving to enable different pattern expressions [9, 23]. Gremlin, on the other hand, focuses more on



Query: *Return a list of films where Clint Eastwood was either an actor or a director.*

Figure 1.2: Graph schema and corresponding query used to illustrate graph query language syntax, adapted from [23]. There are two types of entities, *person* and *film*, and three types of edges, *acts in*, between a person and a film, *directs*, also between a person and a film, and *knows*, between a person and another person.

```
SELECT ?x
WHERE {
  { :Clint_Eastwood :acts_in ?x . }
  UNION { :Clint_Eastwood :directs ?x . }
}
```

Figure 1.3: Example of a SPARQL query to find films matching the criteria described in Figure 1.2.

```
MATCH (:Person {name:"Clint Eastwood"}) -[:acts_in]-> (x3:Movie)
RETURN x3.title
UNION ALL MATCH (:Person {name:"Clint Eastwood"})
  -[:directs]-> (x3:Movie)
RETURN x3.title
```

Figure 1.4: Example of a Cypher query to find films matching the criteria described in Figure 1.2.

```
g.V()
   .hasLabel("Person")
   .has("name", "Clint Eastwood")
   .out("directs", "acts_in")
   .values("title")
```

Figure 1.5: Example of a Gremlin query to find films matching the criteria described in Figure 1.2.

navigational querying; Gremlin queries start from a set of input vertices or edges, and each step specifies where to visit next [10, 23]. SPARQL is somewhere between the two [13, 23].

1.3 Graph Processing Systems

Graph processing systems accelerate the execution of graph algorithms and query languages, but do not provide support for transactions, persistence, or other traditional database features [21]. They usually sit on top of or adjacent to graph databases, serving as tools to process data as part of a data analytics pipeline [21, 31]. Examples include Spark-Gremlin [15], cuGraph [32, 33], and Gunrock [34]. Figure 1.6 illustrates the relationship between graph databases, graph processing systems, graph query languages, and downstream models.

1.4 Graph Neural Networks

Graph Neural Networks (GNNs) are an extension of deep learning models to the graph data format. A simple way to explain the difference between deep learning models (mostly CNNs) and GNNs is to consider the structure of each (Figure 1.7). For instance, in a typical image recognition task, there is a concept of adjacency. Each pixel in an input image is adjacent to its neighboring pixels, and that adjacency matrix is always fixed and uniform across all pixels in the

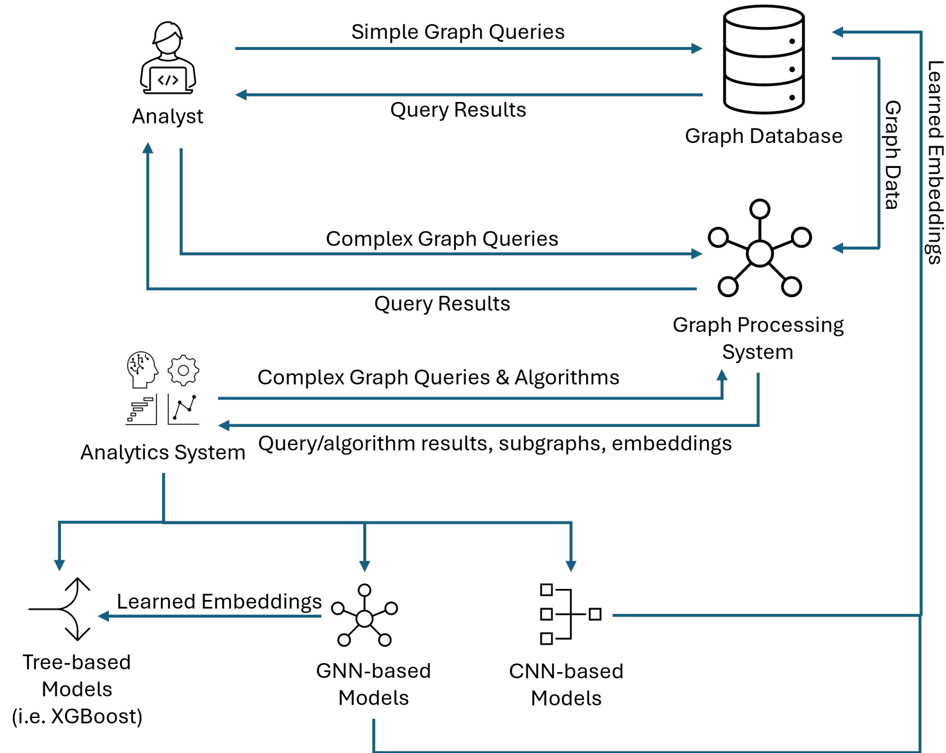


Figure 1.6: The Graph Analytics Process

image (excluding edge pixels, of course, which are a special case where the adjacency matrix is smaller and not square). Most deep learning models use some form of this paradigm, regardless of the data source and data type.

In a GNN, by contrast, rather than individual pixels, we have vertices. The localized adjacency matrix of a vertex has no constraints. It need not be square, fixed, or uniform, because this is not the case in graphs. In a graph, edges determine which vertices are adjacent, and information is typically exchanged through message passing along edges, rather than convolution [20]. There can even be multiple edges between two vertices, and these edges may have either the same type, or different types. Each vertex or edge type may be treated differently by a GNN model as well; again, graphs are not uniform.

GNNs allow for more expressivity and complexity, at the expense of performance. There

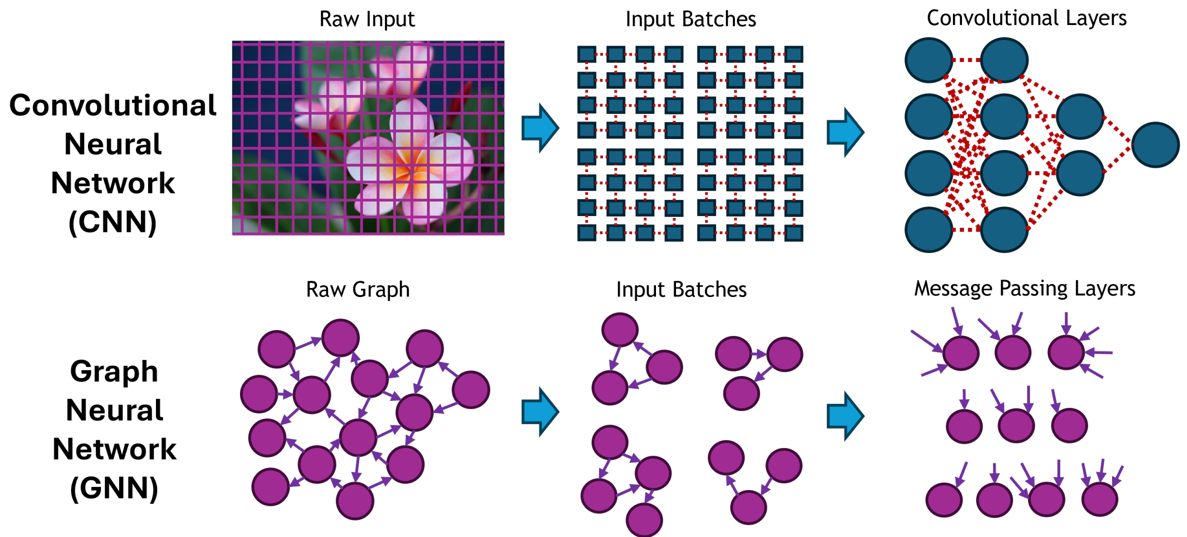


Figure 1.7: Convolutional Neural Networks (CNNs) vs. Graph Neural Networks (GNNs)

are more processes involved in training a GNN, such as *sampling* [20], although one can argue that these processes are not entirely different than those needed to clean and prepare data to produces batches for training a convolutional neural network.

GNNs are relevant to this dissertation for two reasons: first, because they are an excellent example of a downstream model that can use data from graph queries and graph algorithms (Figure 1.6), and second, because they are useful for RAG [35], a critical component of *Large Language Models* (LLMs).

1.5 Large Language Models

Large Language Models (LLMs) are transformer-based models trained to perform language-based tasks, typically framed as responses to user prompts [36]. When used in practice, they have generally been pre-trained on a very large corpus of documents, a process which can take thousands of GPU-hours [37]. Foundation models, like *Llama* [37], can be later fine-tuned [35, 36,

38] for specific tasks, such as question answering and chat.

There are two dominant families of LLM models: *GPT*, driven by OpenAI [36, 39] and *Llama*, driven by Meta [36, 37]. Most Llama models are open-source, making them more common in academic research [36, 37, 40].

Foundation models are naturally stale, having been trained on data that is likely at least several months out of date [36, 41]. They can also be prone to *hallucinations*, which are essentially forms of incorrect output. There are two types of hallucinations: *intrinsic hallucinations*, which occur when the model produces output that directly conflicts with the information it was trained on, and *extrinsic hallucinations*, which occur when the model speculates about information it did not have access to during the training process [36]. *Retrieval augmented generation* (RAG), discussed more in Chapter 4 and Chapter 5, is a way to prevent hallucinations by providing additional context to prompts. This context contains information that either reinforces what the LLM should already know, or contains updated information such as recent news [41]. There are many types of RAG, including traditional RAG based purely on vector search, as well as graph-based RAG that incorporates a knowledge graph [2, 3, 35, 41, 42].

1.6 Main Contributions

My main contributions comprise three key areas, each corresponding to a chapter of this dissertation.

1. First, the concept and development of the BitGraph framework. This was a major undertaking; the new BitGraph framework is exactly that - a *framework* consisting of three separate libraries that work together to provide large-scale, GPU-accelerated graph analyt-

ics. After developing and integrating the components of the BitGraph framework, I then benchmarked its performance across multiple datasets of varying sizes and several different graph queries. These benchmarks demonstrated a runtime speedup of up to 35x over an equivalent naive CPU implementation [28].

2. Second, the process of just-in-time optimization used by Gremlin++, and implementing backends like BitGraph, to deliver superior performance without leaking implementation details to the user. After developing and refining the just-in-time optimization process, I tested it against two different queries. I used the first query to show the step-by-step speedup delivered by each optimization, culminating in a total speedup of 102%. Then, I used the second query to test query optimization against the real-world problem of retrieval augmented generation (RAG), and achieved a total speedup of 40% on large queries, and up to 25% on smaller queries [31].

3. Third and finally, the extension of the BitGraph framework to support RAG, which added unique features including vector search integration and fuzzy matching. This is, as far as I know, the first graph processing system to incorporate embeddings and vector search as first-class API features. My extensions to the framework allow BitGraph to support prize-aware graph traversal, which is the core feature of QRAG, a RAG implementation built using the BitGraph framework. I showed how these extensions delivered up to 840x speedup over naive brute-force vector search, making the QRAG process feasible, and how the direct-input version of QRAG delivered an increase of 14% in question answering accuracy over the non-RAG baseline, well ahead of the other RAG types I tested [43].

Chapter 2: Background and Previous Work

2.1 Background and Motivation

In 2019, industry was starting to catch on to the idea of using GPUs for tasks beyond machine learning. Gunrock [34], released in 2017, was the first serious attempt at a GPU graph analytics library, and cuGraph [32, 33], released in 2018, followed shortly after.[1] Gunrock, cuGraph, and their contemporaries [44, 45] focused on structural graphs, leaving property graphs to be handled by graph databases. Graph databases and graph query languages were very popular, and continued to grow, but they had been mostly left out of the accelerated computing revolution. The first serious attempt at a GPU-accelerated graph query engine was promising, but it was promptly bought, and then killed by Amazon [46, 47]. In response, I created the first version of BitGraph for my master's thesis [48]. This version of BitGraph, which I'll refer to as *BitGraph-2019*, was very different than the successor framework described in detail in this dissertation and related papers published in 2023 and 2024. BitGraph-2019 painstakingly tried to replicate both the semantics and inner workings of the Gremlin [10] query language and TinkerPop [15] standard with limited results. At that point, nobody had attempted to develop something like this in C++, so that by itself was a major struggle, let alone incorporating the custom OpenCL kernels needed to execute filtering and reduction operations [48].

[1] I became a maintainer of cuGraph in 2022.

BitGraph-2019 worked, but it was a flawed and limited product. Every single GPU operation required a custom kernel, which made development difficult. And more fundamentally, BitGraph-2019's API made it nearly impossible[2] to persist any data on the GPU, requiring any GPU operation to first copy data from the CPU, execute on the GPU, and then copy results back to the CPU [48]. Series of GPU operations could not be chained. This was due to baggage inherited from the original Java implementation of Gremlin, which was not really appropriate for any accelerated computing paradigm other than Apache Spark [15, 49].

After defending my master's thesis, I immediately started working on a new version of BitGraph, which became the foundation for my research proposal. This version is a major paradigm shift from the original, relying less on custom GPU kernels and more on an abstract API based on Gremlin's traversal step representation. As I started working on the new BitGraph, the world of graph analytics continued to evolve. Graphs were now becoming input to other models, such as Convolutional Neural Networks (CNNs), Graph Neural Networks (GNNs), and even simpler models like XGBoost and Random Forest. Graph sizes were also continuing to grow, with a typical enterprise graph now comprising at least several million nodes, and hundreds of millions of edges. More and more large enterprises were starting to analyze billion-plus edge graphs. Traditional graph analytics, which includes algorithms like connected components [16], Katz Centrality [19], and PageRank [17], remained relevant, but were increasingly used to extract features needed to train other models rather than as a final product.

Mining massive scale graphs to produce subgraphs and features for downstream models was a natural fit for GPU acceleration. Most of the downstream models were written in PyTorch

[2]Persistence is possible when there is a chip-to-chip interconnect and shared memory, such as in Grace Hopper or Grace Blackwell systems, but these were not available when I was working on BitGraph-2019 or the successor version I started working on shortly afterwards. Furthermore, BitGraph-2019 persisted data in GPU-unfriendly data structures, so there would still be a conversion process (just no transfer over PCI-e).

[50] or similar frameworks, which were already GPU-accelerated. But graph databases, which were increasingly the dominant player in enterprise-scale graph analytics, still didn't support GPU-accelerated graph processing. They remained well behind the curve, but did not feel much pressure to do anything about it.

This changed dramatically with the rise of large language models (LLMs). LLMs require a massive trove of data for training, but this alone is not enough to produce an effective model. Because of the size and complexity of LLMs, constantly retraining an LLM with updated data is time consuming and cost-prohibitive. The solution to this problem is a process known as retrieval augmented generation (RAG). RAG provides updated and complete context to an LLM during inference to provide a full, up-to-date answer to a user's prompt. RAG is also one of the most effective guards against hallucination (see Section 1.5), as this additional context prevents the model from using less-reliable parametric memory to recall facts. Developing a good RAG is difficult and complicated, and requires a fusion of several technologies including vector search/indexing, information retrieval and management, and cross-document correlation. As it turned out, this was an excellent application for large-scale graph analytics [2, 41]. Graphs were very effective in representing information linkage, and graph query languages already offered a way to search and traverse the graph data structure. Still, doing this effectively at scale was hard, so even the most innovative approaches opted for a fairly naive subgraph extraction process, leaving further refinement to a downstream model or algorithm [35].

BitGraph started as a project to accelerate traditional graph analytics, but as the world of graphs evolved, it also evolved into a whole framework, whose development was documented in three research papers. This dissertation describes this work, and how it solves three problems:

1. The need for a GPU-accelerated graph processing backend that can effectively decompose graph queries and traversals into vector and matrix operations well-suited for the GPU.

2. The need for better graph query optimizations that ensure users' queries will run quickly without them needing to know implementation details.

3. The need for a system that bridges the gap between accelerated graph query processing and accelerated vector search in order to effectively serve the RAG use case.

2.2 Previous Work

2.2.1 BitGraph-2019

BitGraph-2019 was the first attempt at implementing a processing frontend and backend for the Gremlin query language in a low-level language. It included Gremlin++, the first low-level language implementation of a Gremlin interpreter [48]. A lot of the work on BitGraph-2019 was dedicated to replicating the structure of Java-Gremlin using C++ constructs such as pointers. This is in contrast to the new version of the BitGraph framework, which I describe in this dissertation, which drops many of the paradigms of Java-Gremlin and uses a vector-based approach [28]. BitGraph-2019 also included an indexing backend, which was very helpful for graph ETL, and GPU implementations of certain Gremlin steps, which were written using separate OpenCL kernels[3]. Overall, BitGraph-2019 had superior ETL performance to other Gremlin-supporting graph processing systems, achieving 2x speedup over the second-fastest backend (TinkerGraph), but struggled in algorithm performance, losing to TinkerGraph when processing larger graphs

[3] I switched to CUDA when writing Maelstrom, the data structures and algorithms library used by the new version of BitGraph.

[48]. As discussed earlier, constantly copying data between the CPU and GPU limited performance significantly, and even if a system like Grace Hopper or Grace Blackwell would have been available at the time, BitGraph-2019 would still probably underperform TinkerGraph due to its localized data format limiting GPU utilization.

2.2.2 Other Work

Prior to BitGraph-2019, I was hired by the Johns Hopkins University Applied Physics Lab (APL) to write a complete graph analytics library using the Gremlin query language. This library included common algorithms, such as connected components, betweenness centrality, and various propagation algorithms, as well as basic graph ETL, all written in Gremlin. The primary objective of this library was to support customers across a broad variety of technology stacks, including those that used Spark through Spark-Gremlin, as well as those that used graph databases like Neo4j and JanusGraph. While working on this library, I encountered a variety of performance issues with especially large graphs. Despite having access to very powerful GPUs, which by then had been proven to be capable of handling very large graphs, there was no way of taking advantage of them to accelerate the Gremlin queries I needed to run. This, as well as another research project where I rewrote various machine learning algorithms using the RAPIDS stack, motivated BitGraph-2019.

Chapter 3: A Framework for GPU-Accelerated Graph Query Processing

3.1 Overview

This component of my dissertation discusses the design of the BitGraph framework, from the bottom up. It is based on a large body of work, much of which was included in [28]. As opposed to the RAG-focused work referenced later in this dissertation, this section focuses on applications to traditional graph analytics and temporal graph analytics, and includes appropriate benchmarks. Similar to Gunrock [34], the BitGraph framework, specifically its Gremlin++ query language, provides a way for users to write their own algorithms. This is done by researchers and practitioners across fields like cybersecurity, social network analysis, and molecular biology. However, Gremlin++ supports GPU acceleration, allowing these custom algorithms to run much faster without any knowledge of GPU programming.

3.2 Related Work

3.2.1 Gunrock

Released in 2017, *Gunrock* [34] was an early foray into the world of GPU-accelerated graph processing. Rather than provide a query language for interaction with the graph, Gunrock exposes *graph primitives* through its API. Gunrock was one of the earliest graph processing

systems to use *frontier parallelism*, which the BitGraph framework also uses. In frontier parallelism, each operation operates upon a frontier, a data structure of uniform elements that can be *advanced* through individual steps. Algorithms are expressed as a series of these steps. The three main primitives of Gunrock are *advance*, *filter*, and *segmented intersection* [34]. These are very similar to the operations included in Maelstrom's vector algorithm library. Gunrock further divides advance primitives into four subtypes: *Vertex-to-Vertex*, *Vertex-to-Edge*, *Edge-to-Vertex*, and *Edge-to-Edge* [34]. These can be mapped to the Gremlin++ steps that perform the same operations, such as the *VertexStep*, which performs three of these four operations. Again looking at the higher-level API in Gremlin++, the *FilterStep* of Gremlin++ also has near-identical function to its Gunrock equivalent. While there is no single step that maps to Gunrock's segmented intersection step, it can still be mapped to a combination of Gremlin++ steps. Gunrock also included a set of optimizations, including kernel fusion, which are similar to Gremlin++ *traversal strategies*. Unlike kernel fusion, though, traversal strategies are part of a higher-level API. Chapter 3 of this dissertation discusses traversal strategies extensively.

While there are many similarities between the two frameworks, there are some key differences. The most obvious is the BitGraph framework's support for a fully-featured query language (Gremlin++). Gremlin++ offers near-identical semantics to OLAP Gremlin, which is Turing-complete [10]. This includes features far beyond simple graph operations; for instance, Gremlin++ offers filtering, reduction, and projection operations. Gremlin++ is also more targeted towards the property graph use case, incorporating steps that can mutate vertex and edge properties, and perform control flow based on property values. It is much easier to write algorithms in a fully-featured query language that is designed to be expressive than with a set of primitives, and most data scientists, already being used to SQL, would have an easier time adopting Gremlin++,

especially because it is based on an existing language with many examples and supporting graph databases.

3.2.2 cuSTINGER and Hornet

cuSTINGER [45] and *Hornet* [44] are high-performance data structures for dynamic graph analytics, similar to those used by BitGraph. Hornet was created as a successor to cuSTINGER, and added key feature such as dynamic sparse matrices and memory reclamation. For many workloads, cuSTINGER and Hornet can outperform CSR [44]. BitGraph also supports dynamic graphs, and allows mutation of the graph through the Gremlin++ query language, but uses its own *canonical representation*, which is a combination of the CSR, COO, and CSC matrix formats. This is a better fit for the frontier-based Maelstrom algorithms used to manipulate the graph, and is also well-suited for multi-GPU processing. While BitGraph currently does not support multi-GPU processing, adopting the canonical representation helps make a future multi-GPU extension easier.

3.2.3 Blazing SQL and Dask-SQL

These two projects brought GPU acceleration to SQL queries, much like BitGraph does for graph queries. Both were built on the NVIDIA RAPIDS ecosystem [51], but *dask-SQL* abstracts most of RAPIDS through the Dask API [52], and is easier to scale to multi-node, multi-GPU applications. *Blazing SQL* achieved up to 20x speedup over Apache Spark for a series of ETL queries [51], which motivated me to achieve a similar speedup with BitGraph.

3.2.4 Lotan

Lotan [53] is a framework that connects GNNs to graph databases, similar to what I describe in Chapter 5 of this dissertation, but not targeted to RAG. Instead, Lotan focuses on graph sampling and subgraph extraction, which can be done through graph queries. Lotan has its own just-in-time query optimization system analogous to that of Gremlin++, but ultimately, it does not go beyond modifying the input query. It is still up to the upstream graph database and its query processing engine to execute the query, and that process is still not GPU-accelerated. Therefore, Lotan does not completely remove the bottleneck that the BitGraph framework is addressing. Lotan's query acceleration is also limited to queries used for GNN training, as opposed to BitGraph's, which accelerates all queries.

3.2.5 OLAP Gremlin/Spark-Gremlin

Spark-Gremlin [15] is a graph processing system for OLAP Gremlin traversals. It uses a bulk-synchronous parallel (BSP) processing model, which scatters individual traversers across processes to be executed [15, 49]. This is conceptually similar to what the BitGraph framework does, but with some key differences. First, in the BSP model, each computation unit is independent. Computation units in Gremlin and Gremlin++ correspond to *traversers*. In Java-Gremlin and Spark-Gremlin, traversers contain all data needed to execute a computation, including *side effects*, *paths*, etc. However, in Gremlin++, traversers are only an abstraction, and data is stored across several data structures in a frontier model. This is explained in more detail later in this chapter, along with the reasons for this choice.

3.2.6 Gremlin Server and Drivers

Gremlin Server [15] is a system for running a Gremlin query on a remote backend. *Gremlin drivers* [15] are implementations of Gremlin that can run natively in other host languages, such as Python, JavaScript, and .Net. Gremlin servers and drivers work together to allow separation of graph processing from the rest of the workflow. For instance, you could have an always-running graph hosted on a large cluster with Gremlin server that users could access from their local machines to run queries against. If a user wanted to run a Python script that made a Gremlin query, they could use Gremlin-Python, the driver for Python, to compile send the query to the server. This relationship is illustrated in Figure 3.1

While the BitGraph framework does not have an equivalent of Gremlin server, it does have Python bindings for Gremlin++ which are equivalent to the Gremlin-Python driver for traditional Gremlin. These Python bindings were written after publication of [28] and used to integrate Gremlin++ queries with PyTorch as part of the research done in [31] and [43].

3.3 Designing Maelstrom

3.3.1 Overview and Motivation

When I first started work on the new BitGraph, I originally wanted to keep as much of BitGraph-2019 as possible, but change the underlying data structures, and add GPU versions of more Gremlin++ traversal steps. This was a very involved process, since each new GPU step needed a separate kernel [48]. I also had to maintain a CPU-compatible library for comparison. Eventually, I reached a point where I could benchmark connected components and a few other

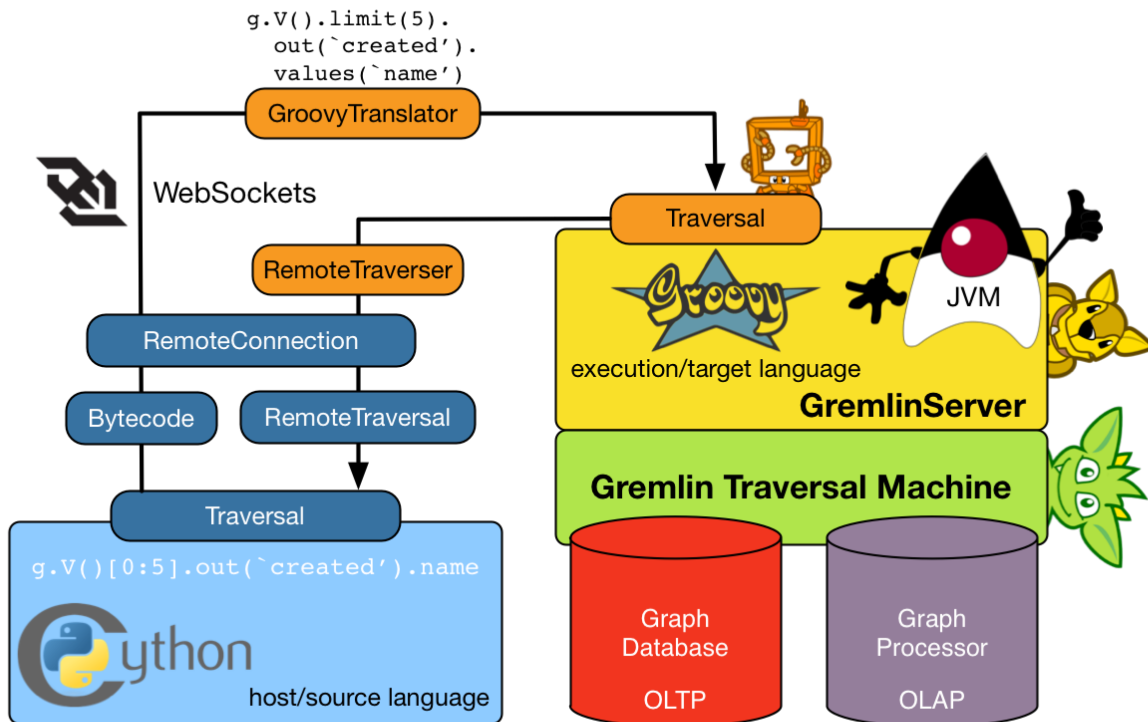


Figure 3.1: Illustration created by the Apache TinkerPop team that shows the relationship between Gremlin drivers, Gremlin server, the query processing backend, and the upstream database [15]. The Python driver compiles a Gremlin query, and uses WebSockets to send the query to the Gremlin server connected to the database. The query is handled by the Gremlin Traversal Machine, the graph processing backend of traditional Gremlin, which I usually refer to as "Java-Gremlin". Java-Gremlin connects either directly to the database, or to an OLAP processing backend like Spark-Gremlin, where the data is stored.

algorithms. Unfortunately, not only had this taken far too long to accomplish, but the performance was sub-par. The speedup was minimal at best - again, because data kept being cycled back and forth between the CPU and GPU, which incurred not only the overhead of PCI-e transfer, but also the overhead of converting traverser objects on the CPU to and from vectors that the GPU could process.

At this point, I realized something needed to change. One of the issues, even on the CPU, with maintaining individual traversers, was that this resulted in poor spatial locality, and therefore many page faults. Queries that touched a large portion of the graph performed especially poorly; for instance, the connected components query, which accesses every vertex in each iteration. So not only was this paradigm incurring unnecessary work for GPU processing, it was also negatively impacting operations that had to be done on the CPU. This reminded me of my original goal of what I called a *hybrid* graph processing system. I'd originally envisioned a hybrid graph processing system as managing both a CSR-based structural graph and table-based property graph, and accelerating both. But why keep them separate at all? As long as I could replicate most of the Gremlin API, why continue to hold on to the table-based structure of Gremlin-Java?

Furthermore, why even keep the distinction between CPU and GPU processing? Why couldn't I merge them into a single API that automatically called the right code based on where the data was stored? *Thrust* [54] had solved a similar problem by providing a vector and algorithm interface that could run on both the CPU and GPU. So, I decided to do the same. The BitGraph framework would be rebuilt on top of an API that was both storage-agnostic (host memory, device memory, managed memory, pinned memory) and compute-agnostic (CPU/GPU), and would use Thrust to write most of the algorithms. It would also handle the thorny issue of type erasure, which was an ongoing issue due to Gremlin++'s support for properties with any value (string,

integer, float, double, etc.). Developing type-erased algorithms was much simpler in Thrust, since I could take advantage of C++ templates to compile multiple versions of the algorithms, as long as this new API could determine which version to call at runtime.

I named this new API *Maelstrom*, and created the Maelstrom library and GitHub project [55]. Maelstrom provides type-erased, compute agnostic versions of critical matrix and vector operations needed by Gremlin++ and BitGraph. With Maelstrom, I would rewrite Gremlin++ and BitGraph as a series of calls to the Maelstrom API. This would make development significantly easier, so long as I updated the data structures and compute paradigm of Gremlin++ to match.

3.3.2 The Maelstrom API

The Maelstrom API was designed to be as simple as possible, and was modeled after the Thrust [54] and PyTorch [50] APIs. Like PyTorch, Maelstrom uses the same API for all operations regardless of where the data is stored or computed. Where to execute a Maelstrom algorithm is a runtime decision, made based on the data stored in the vectors or matrices being operated upon. Maelstrom is capable of handling not just device and host memory, but also *managed memory* (sometimes referred to as UVM in other literature) and *pinned memory* (also referred to as UVA in other literature). Managed memory separates data into pages which can be cached on device memory, and spilled to host memory when necessary. It does require extra memory management steps, and swapping of entire pages, which incurs a performance penalty, but for many applications, this penalty is minimal [56]. Managed memory is especially good for handling vertex and edge properties, since those are large and infrequently accessed. Pinned memory, sometimes referred to as host-pinned memory, is page-locked host memory that can be

implicitly transferred to the device as needed [56]. When there are frequent random accesses to small amounts of data, pinned memory can outperform managed memory. Subgraph extraction and sampling algorithms often follow this access pattern, making it a good option for storing certain vertex and edge properties when they cannot fit into device memory. Maelstrom allows a user to declare where data is stored, just as a PyTorch user can specify whether to store a tensor on host, device, or pinned memory. Also like PyTorch tensors, Maelstrom vectors can be moved across memory types by calling the *to()* function.

As mentioned before, Maelstrom also supports type erasure. Maelstrom algorithms work on any of the primitive types supported by Maelstrom (*(u)int64*, *(u)int32*, *(u)int8*, *float64*, *float32*), and the correct template instantiation is determined at runtime. This eliminates the need to perform any type checking in the higher-level Gremlin++ and BitGraph APIs.

Maelstrom currently supports three key data structures: *vectors*, similar to vectors in the C++ standard library or 1D PyTorch tensors, *hash tables*, and *sparse matrices*. Maelstrom's algorithms library operates on these core data structures. The hash table structure makes extensive use of the cuCollections dynamic map API [57], and the sparse matrix structure can be thought of as a type and compute-erased version of *cuSPARSE* [58], which inspired it, along with some special algorithms designed for frontier parallelism-based operations.

Finally, the Maelstrom API was written to support multi-GPU and even multi-node processing with zero code change. There are currently a handful of Maelstrom algorithms that support multi-GPU processing [55].

3.4 Designing Gremlin++

3.4.1 Overview

Gremlin++ is the core of the BitGraph framework, sitting on top of Gremlin++ and below BitGraph. It provides the Gremlin++ query language, which is the primary means of interaction with the graph. The current version of Gremlin++ is a complete redesign from what was part of BitGraph-2019. It is now completely independent from BitGraph; any backend that implements the Gremlin++ API can use its query processing engine.

3.4.2 Structure of a Query

Gremlin is often referred to as a *traversal language*, a special class of query language that traverses another data structure [10]. It is also a *functional language*, defining a series of mathematical operations. There are two version of traditional Gremlin, OLAP Gremlin and OLTP Gremlin. OLAP (online analytical processing) is related to BSP (bulk-synchronous parallel) processing. In the OLAP model, each stage operates on *all* the data, and then advances [59]. This is the paradigm used by systems like Apache Spark [49], as well as some graph processing systems like Spark-Gremlin [15], Gunrock [34], cuGraph [32, 33], and BitGraph [28, 31]. In the OLTP (online transactional processing) model, data is split into individual components that are handled separately and then combined into a single transaction [59]. OLTP is generally used for small, basic queries rather than large-scale analytics [10, 15, 59].

Gremlin++ is a separate query language based on Gremlin OLAP. It has nearly identical semantics to Gremlin OLAP, with few key differences in handling of side effects, which are

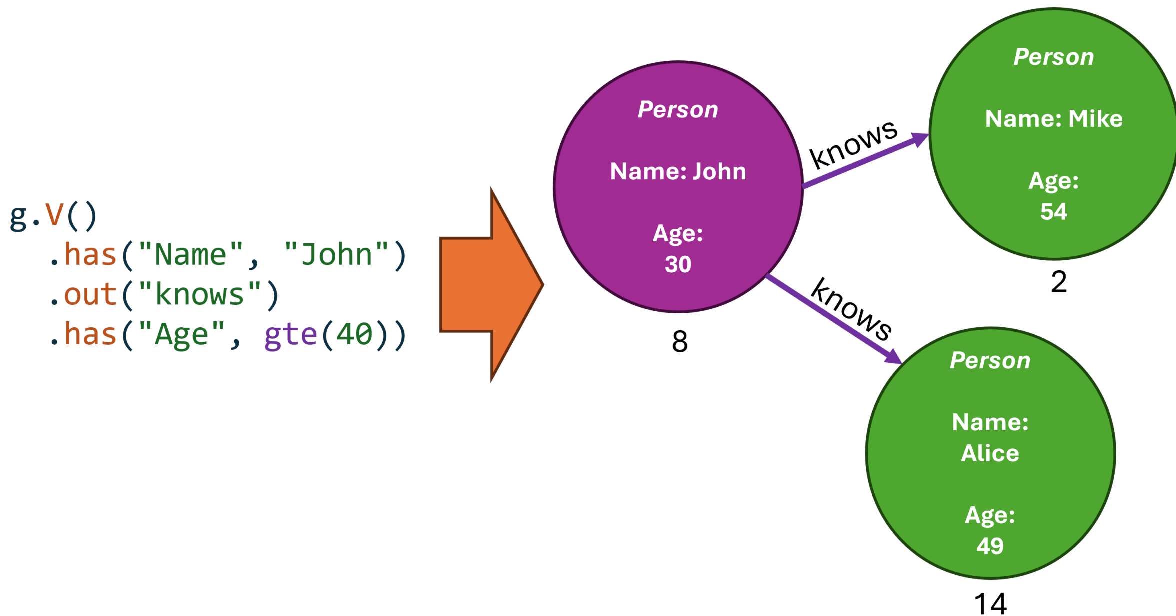


Figure 3.2: A basic Gremlin query. This query starts from the vertex with a value of "John" for the "Name" property, traversers edges with the "known" edge label, and filters the resulting vertices to those whose "Age" property is greater than or equal to 40 (vertex 2 and vertex 14). Chapter 4 illustrates Gremlin++ queries in more detail.

like temporary variables that get passed through from step to step. This makes it easy for users already familiar with traditional Gremlin to adopt Gremlin++. Figure 3.2 shows an example of a Gremlin++ query.

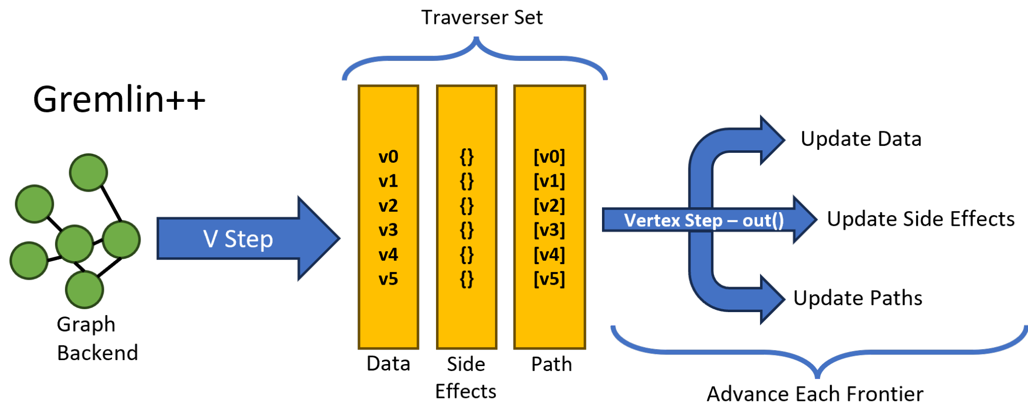
3.4.3 The Traverser Abstraction

In Java-Gremlin, traversers are individual objects that represent data currently being traversed. For example, in Figure 3.2, we start with one traverser, which points to vertex 8, and end with two traversers pointing to vertex 2 and vertex 14. Steps in Gremlin OLAP, such as the *HasStep - has()* and *VertexStep - out()*, take a set of traversers as input and output a new set of traversers. These traversers are aware of the path they took, and any intermediate properties, called side effects, that were stored from previous steps.

Gremlin++ keeps this traverser abstraction, but does not actually have a traverser data structure. Instead, it uses a structure called a *traverser set*, which manages three internal structures: *traversed objects*, *side effects*, and *path information* [28]. These three structures globalize the data that Java-Gremlin localizes to individual traversers, as shown in Figure 3.3.

3.4.4 Life of a Query

Gremlin++ is comprised of two APIs, the traversal API, which defines each step in the query language, and the structure API, which defines the basic components of a property graph (vertices, edges, and properties) that executing backends like BitGraph need to provide [28]. A Gremlin++ query is executed using a combination of steps defined in Gremlin++ and structure API calls. Unlike the traversal API, the structure API in Gremlin++ consists entirely of abstract classes, leaving it to backends to determine how and where the graph will be stored, and how to access and modify its structure and properties. This closely mirrors Java-Gremlin, but, like everything else in Gremlin++, the structure API is designed for vector rather than scalar parameters. For instance, the structure API function in Gremlin++ that retrieves adjacent vertices from the graph takes a vector of vertex ids as opposed to the Java-Gremlin equivalent which takes a single vertex parameter. Keeping the API vector-based makes it easy to implement it in BitGraph using Maelstrom. Figure 3.4 shows the complete interaction between Gremlin++, BitGraph, and Maelstrom as a Gremlin++ query is executed.



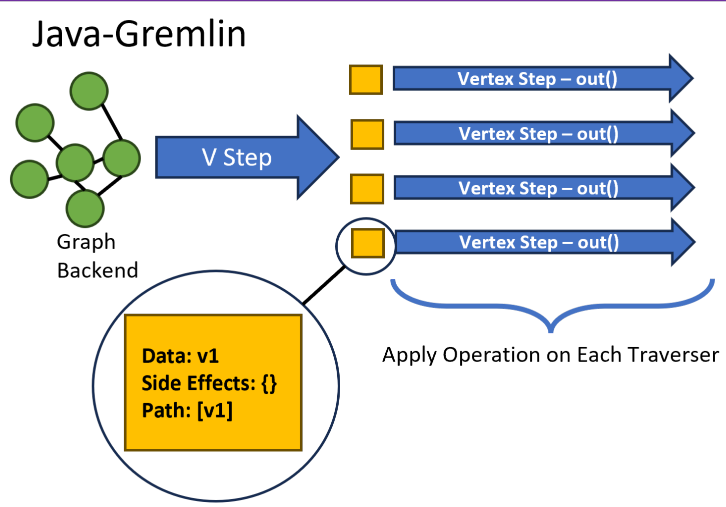


Figure 3.3: Comparison of Gremlin++ and Java-Gremlin traversals. Gremlin++ globalizes the local traverser objects into the *traverser set* structure. Each step in Gremlin++ operates on the globalized data as part of an *advance* operation. This is how Gremlin++ implements frontier parallelism, which makes it easy to take advantage of the GPU. As discussed earlier, advance operations are implemented as a series of Maelstrom API calls that mutate the data in the existing traverser set and output a new traverser set.

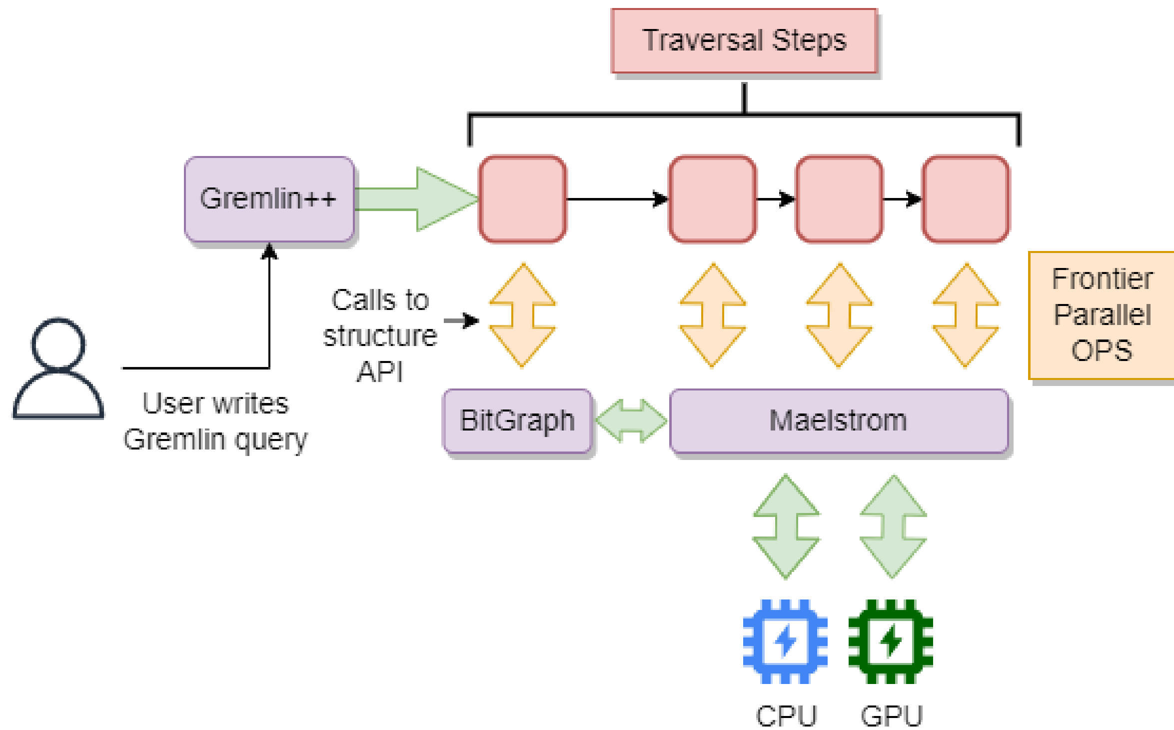


Figure 3.4: Life of a Gremlin query in the BitGraph Framework. Individual queries are broken up into their component steps (this is discussed further in Chapter 4), which are defined as a series of vector and matrix operations in Maelstrom. Steps that need access to graph data make calls to the Gremlin++ structure API, which is implemented by BitGraph, also using Maelstrom. Maelstrom's compute-agnostic API allows these steps to be executed on the CPU or GPU depending on where traversed data is stored.

3.5 Designing BitGraph

3.5.1 Overview

BitGraph is the highest-level component of the BitGraph framework, using both the Maelstrom and Gremlin++ APIs to provide the first true implementation of a hybrid graph analytics system. BitGraph maintains the graph structure and properties in the hash table and sparse matrix data structures provided by Maelstrom, and implements the Gremlin++ structure API to allow Gremlin++ queries to access data stored in BitGraph. Again, by using the power of the Maelstrom API, BitGraph is a very lightweight library, and it is easy to add new features or performance tweaks.

3.5.2 Storage and Access of the Graph Structure

BitGraph stores the graph in a variety of structures, switching between them as needed depending on the algorithm or operation being executed against the graph. These structures are collectively referred to as the *canonical representation* of the graph, and are comprised of canonical COO (coordinate list) format, canonical CSR (compressed sparse row) format, and canonical CSC (compressed sparse column) format.

3.5.2.1 Canonical COO Format

When modifying the graph, BitGraph transforms the structural representation to the canonical COO format if it is not already in that format already. Canonical COO format is a standard COO matrix sorted by insertion order. Insertion or deletion of vertices or edges in this format

takes a best case time of $O(1)$ and worst case time of $O(|E|)$. Canonical COO format also saves memory by eliminating the need for an additional $O(|E|)$ vector to store edge IDs, resulting in a total memory usage of $O(|E|)$ [28]

3.5.2.2 Canonical CSR/CSC Format

Canonical CSR and canonical CSC formats are used when traversing the graph, such as when a *VertexStep* is called. They include an $O(|E|)$ edge permutation vector along with the CSR or CSC sparse matrix representation of the graph, so that edge IDs are preserved, and the canonical COO matrix can be recreated when needed. These formats deliver the best performance for *adjacency queries*, which are operations on the graph that take a set of input vertices and output a set of neighboring vertices. Canonical CSR format is used for adjacency queries of outgoing edges, and canonical CSC format is used for adjacency queries of incoming edges. These operations have a time complexity of $O(|T|)$, where T is the number of vertices whose neighbors are being queried (the input vertices to the adjacency query). This is far superior to the time complexity of $O(|T||E||V|)$ that this operation would take with the canonical COO format.

Canonical CSR/CSC format has a total memory complexity of $O(|E|+|V|)$ for the CSR/CSR sparse matrix and $O(|E|)$ for the edge permutation vector. This simplifies to $O(|E|+|V|)$. Further considering that generally $|E| \gg |V|$, the approximate memory usage of canonical CSR/CSC format is $O(|E|)$, making it asymptotically no more expensive than canonical COO format. This means that the benefit of faster adjacency queries can be achieved without a major impact on memory usage [28]

3.5.3 Storage and Access of Vertex and Edge Properties

Unlike the graph structure, vertex and edge properties can have many different types, and may or may not be present on any given vertex or edge. For this reason, they cannot be stored as a vector or series of vectors. Instead, BitGraph uses Maelstrom's hash table API to store vertex and edge properties. BitGraph exposes Maelstrom's compute-agnostic capabilities to the user, allowing them to choose whether their data should be stored in device memory, host memory, managed memory, or pinned memory. Device memory and managed memory are the most practical choices, with pinned memory serving as a good alternative to managed memory when there are many random accesses of individual vertex property values, as opposed to queries that access large parts of the graph.

3.6 Performance Analysis

3.6.1 Overview

By taking advantage of Maelstrom's accelerated vector and matrix operations, BitGraph was able to achieve significant speedup over Java-Gremlin in processing a variety of queries. I determined this speedup through benchmarks run against Java-Gremlin with TinkerGraph, the standard Gremlin in-memory graph backend [15].

```
g->V().property("cc", id()).iterate();
g->V().property("old_cc", values("cc")).iterate();
size_t diff = 1;
while(diff > 0) {
    diff = g->V()
        .property("old_cc", values("cc"))
        .property("cc",
                  union({
                        both().values("old_cc"),
                        values("old_cc")
                  }).min()
               )
        .elementMap({"cc", "old_cc"}).where("cc", neq("old_cc"))
        .count().next();
}
```

Figure 3.5: Gremlin query for the connected components algorithm. Written using the Gremlin++ C++ API. Based on the implementation described in [16].

3.6.2 Connected Components Benchmarks

3.6.2.1 Overview

Connected components is a ubiquitous graph algorithm that can be written in nearly any programming or graph query language, and has significant utility across problems like community detection, graph classification, and temporal analysis. While it is certainly possible to write a CUDA kernel that would easily beat the performance of Gremlin++, most users will have no idea how to do that. This makes it an excellent benchmark - the connected components algorithm is a lot like other algorithms users may want to write in Gremlin or Gremlin++, so it gives us a good idea of what a typical user can achieve with Gremlin++ versus traditional Gremlin. The actual traversal used for this benchmark is shown in Figure 3.5. It is roughly based on the implementation described in [16].

To illustrate how data size affected speedup, I ran this benchmark using four different

Dataset	*Vertices*	*Edges*	*System*
Facebook-Combined [60]	4K	90K	Intel i7-12700K + RTX A6000
Twitter-Combined [60]	80K	2.4M	Intel i7-12700K + RTX A6000
GPlus-Combined [60]	100K	30M	Intel i7-12700K + RTX A6000
Livejournal [61, 62]	5M	70M	AMD EPYC 7452 + RTX A6000

Table 3.1: Connected Components Benchmark Datasets

datasets: *facebook-combined* [60], *twitter-combined* [60], *gplus-combined* [60], and *livejournal* [61, 62]. All datasets for this benchmark were obtained from SNAP [63]. Table 3.1 shows the size of each dataset. This benchmark was run using an overclocked Intel i7-12700K CPU and NVIDIA RTX A6000 GPU, except for the livejournal benchmark, which was run on a different system with more host memory due to the high RAM usage of Java-Gremlin on that dataset.

3.6.2.2 Results

The results of this benchmark are shown in Figure 3.6. BitGraph achieved a significant speedup on this benchmark, generally doing better with larger graphs. BitGraph achieved a maximum speedup of 35x on the livejournal dataset. BitGraph's canonical CSR format was primarily responsible for delivering this result, allowing very fast adjacency queries, along with the highly-performant Maelstrom hash tables, which allowed fast lookup and updating of the "cc" and "old_cc" properties. BitGraph's frontier parallelism even allowed these properties to be queried and updated in parallel, which Java-Gremlin does not support. GPU-based minimum aggregation also played a role, allowing the minimum component id for each vertex to be determined as part of a global operation spanning the GPU rather than a series of independent operations.

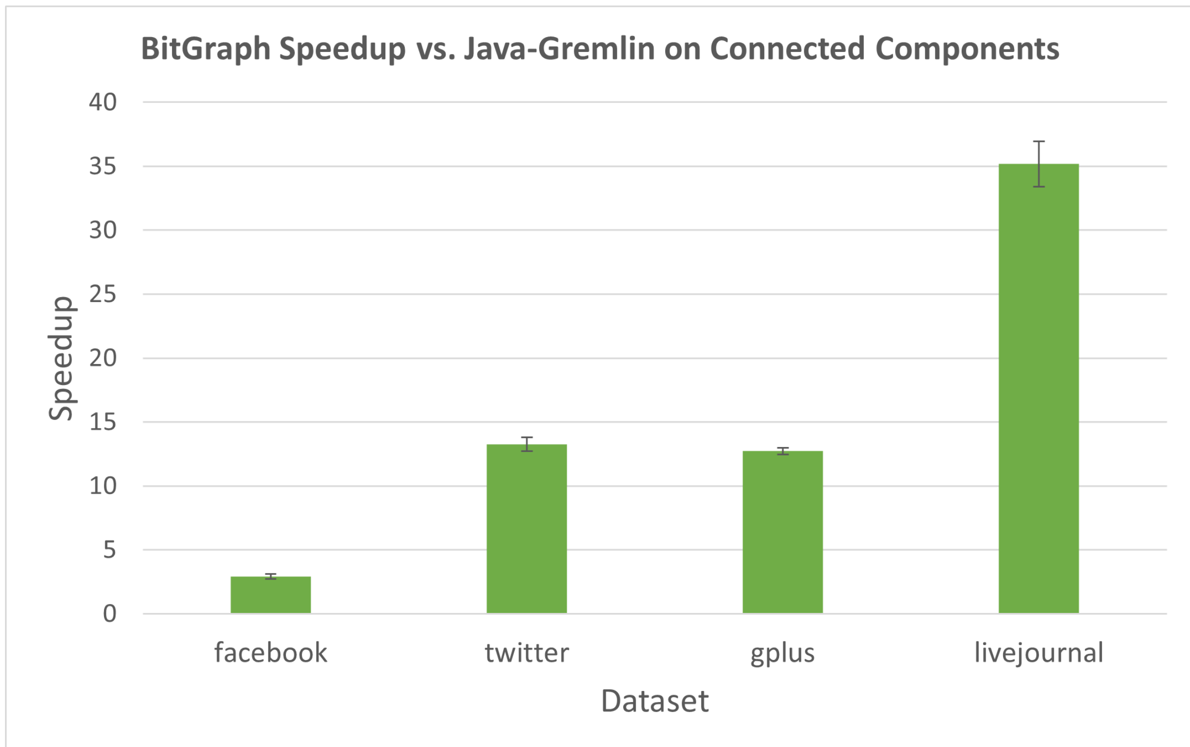


Figure 3.6: Connected Components Benchmark Results

3.6.3 Temporal Shortest Paths

3.6.3.1 Overview

Shortest path algorithms are just as ubiquitous as the connected components algorithm. There are many versions of these algorithms, such as *Bellman-Ford* and *Dijkstra's Algorithm* for the single-source shortest paths problem, and the *Floyd-Warshall Algorithm* for the all-pairs shortest path algorithm [14]. This benchmark tests a Gremlin++ query 3.8 designed to solve a similar problem, given the logistics graph shown in Figure 3.7, which we describe in Table 3.2.

In this benchmark, I used the tgbl-flights dataset [64], from the Temporal Graph Benchmark (TGB) [65] project, along with a subset of that data comprised of the first 1 million edges. These datasets are described in Table 3.3. Again, for the smaller dataset I used an overclocked Intel

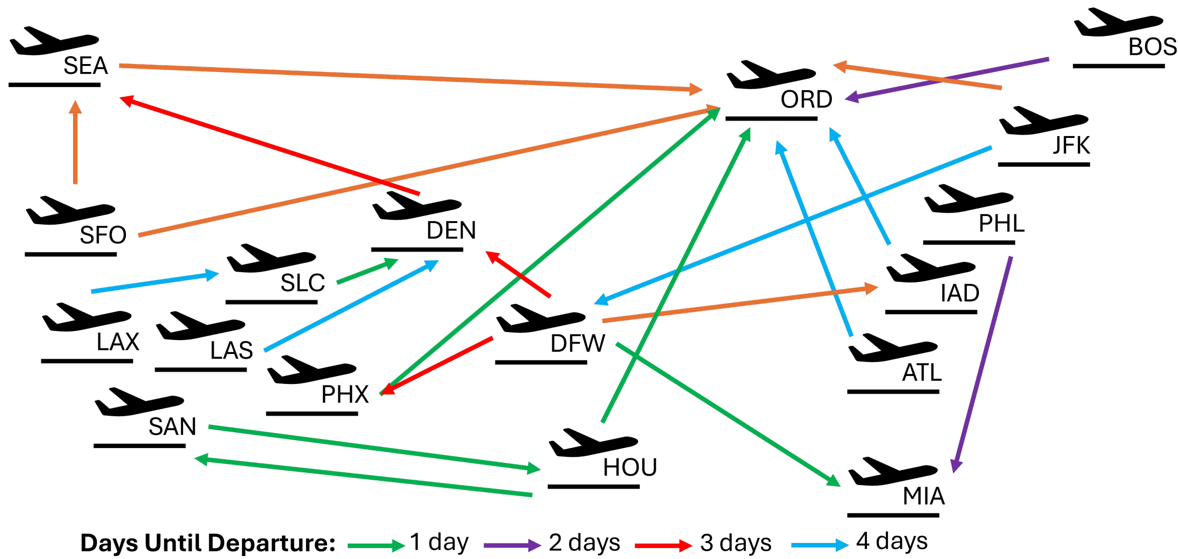


Figure 3.7: Graph of a logistics network that forms the basis for the problem described in Table 3.2. This graph shows a network of airports (vertices) and flights (edges), where edges are color-coded by date. This network was based on the tgbl-flights dataset [64, 65]. The actual benchmark used the complete tgbl-flights graph and actual dates, from 2019 to 2022.

Consider the following problem: *Suppose I have a package that I need to transfer through my logistics network. Whenever a package arrives at a destination facility, it needs to first be sorted overnight, before it can be sent on a plane the next day. How many nodes in the logistics network can this package reach within four days, provided I know the current location of the package?*

Table 3.2: Logistics Pathfinding Problem

```
g->V(v_start)
  .repeat(
      property("visited", 1)
      .elementMap({"last_time"})
      .outE().has("time", gremlinxx::P::lte(time_end))
      .elementMap({"time"})
      .where("time", gremlinxx::P::gt("last_time"))
      .as("last_e")
      .values("time").as("last_time")
      .select("last_e").inV().hasNot("visited")
      .elementMap({"name"})
      .as("v")
      .select("last_time").min(ScopeContext(Scope::local, "name"))
      .select("v")
      .property("last_time", select("last_time"))
  ).iterate();
auto v_total = g->V().has("visited").count().next();
```

Figure 3.8: Gremlin query that finds a solution to the temporal shortest paths problem, as described in Table 3.2. Written using the Gremlin++ C++ API.

Dataset	*Vertices*	*Edges*	*System*
TGBL-Flights-1M [64]	60K	1M	Intel i7-12700K + RTX A6000
TGBL-Flights [64]	60K	70M	AMD Epyc 7452 + RTX A6000

Table 3.3: Temporal Shortest Paths Benchmark Datasets

i7-12700K CPU and NVIDIA RTX A6000 GPU, and for the larger dataset, I used a machine with more RAM that was needed to run this benchmark on Java-Gremlin.

3.6.3.2 Results

Figure 3.9 shows the results of the temporal shortest paths benchmark. BitGraph performed well on this benchmark, but not nearly as well as it did on the connected components benchmark, achieving only a maximum 1.8x speedup on the full dataset. Most of this speedup can be attributed to the CSR structure, which allowed for very fast adjacency lookups that were parallelized by the GPU, as well as running the minimum aggregation on the GPU. However, the speedup was limited by poor performance of the Repeat Step, which incurred too many device-

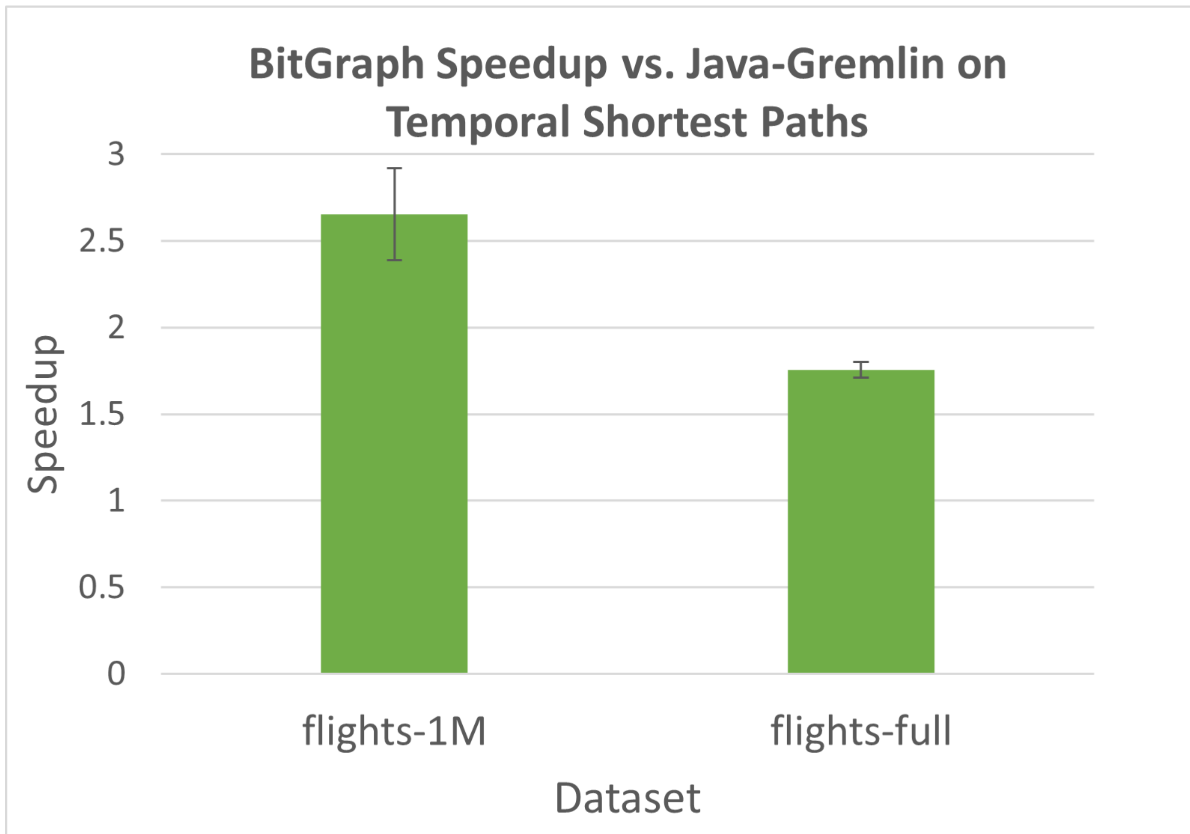


Figure 3.9: Temporal Shortest Paths Benchmark Results

to-host copies. This result helped motivate the work discussed in Chapter 4, which among other things aimed to eliminate these issues through *loop unrolling*.

3.7 Discussion

In this component of my dissertation, I provided an overview of the BitGraph framework, how it came to be, and how it was constructed from its three component libraries: Maelstrom, Gremlin++, and BitGraph. I described the architecture of each library and its API, explaining how they build on each other to create a fully-accelerated hybrid graph processing system. Finally, I analyzed the performance of this framework on two different use cases, showing one

case where the speedup was excellent, and another where it was more modest. This modest use case, the temporal shortest paths query, helped me determine where there was room for improvement. While I had done an excellent job creating a high-performance graph analytics framework, there were still gaps processing some types of queries. These queries would benefit from an improved set of just-in-time optimizations that removed redundant operations, unrolled loops, and performed other advanced techniques to speed up query execution. Chapter 4 discusses the query optimization process in Gremlin++ and illustrates how it can be used to solve this problem.

Chapter 4: Graph Query Optimizations for Machine Learning and RAG Use Cases

4.1 Overview and Motivation

This component of my dissertation describes the query optimization process in Gremlin++, which uses just-in-time optimizations referred to as *traversal strategies* to improve query runtime. There were three key motivations for this work: first, the performance issue discussed in Chapter 3 that could have been solved with loop unrolling; second, the need to prove that Gremlin++ was capable of handling the types of complex queries needed for downstream ML and RAG tasks; and finally, to produce a foundation for graph query optimization and acceleration that others could build from, since only a few scholars have researched graph query language optimizations at this level of detail.

Compared to the third and fifth chapters, this chapter is much more theory-focused, describing the basics of graph query optimization and organizing the types of optimizations into specific categories. However, to demonstrate these optimizations in practice, I use Gremlin++ to illustrate the complete end-to-end optimization process, concluding with a case study that examines the effects of this process on a RAG query and includes performance benchmarks.

This section is primarily based on work I published in [31].

4.2 Related Work

4.2.1 Banyan and GOpt

Created by Alibaba, *Banyan* [66] and *GOpt* [67] are open source projects for graph query acceleration, also focusing on Gremlin queries. Unlike Gremlin++, their focus is on OLTP rather than OLAP queries. Baynan and GOpt focus on accelerating two types of queries: *pattern-matching relational queries* [67] and *scopeable queries* [66]. Alibaba uses some of the same methods described in this chapter, since many can apply to both OLAP and OLTP queries. They also support some advanced optimizations that consider the shape of the graph at runtime to perform load balancing across processes. This can be seen as an extension of the backend-specific optimizations discussed here, although, again, these load balancing optimizations do not make as much sense in an OLAP system, which performs load balancing automatically. For instance, Gremlin++ uses the concept of *scopes*[1] to allow subqueries, such as those used to update vertex and edge properties, to be treated as top-level queries. Therefore all computation units on the GPU are available to them. This is illustrated in Figure 4.1.

4.2.2 GraphScope

GraphScope [68], also from Alibaba, is an open source project that allows several other Alibaba frameworks to work together. These frameworks include their GNN libraries, Gremlin query processing engine, and graph management system. GraphScope supports very limited GPU acceleration, only allowing what I refer to as *dispatching* (Figure 4.2) for a subset of al-

[1]Banyan also uses this concept, although it executes subqueries independently rather than combining them into a new top-level query.

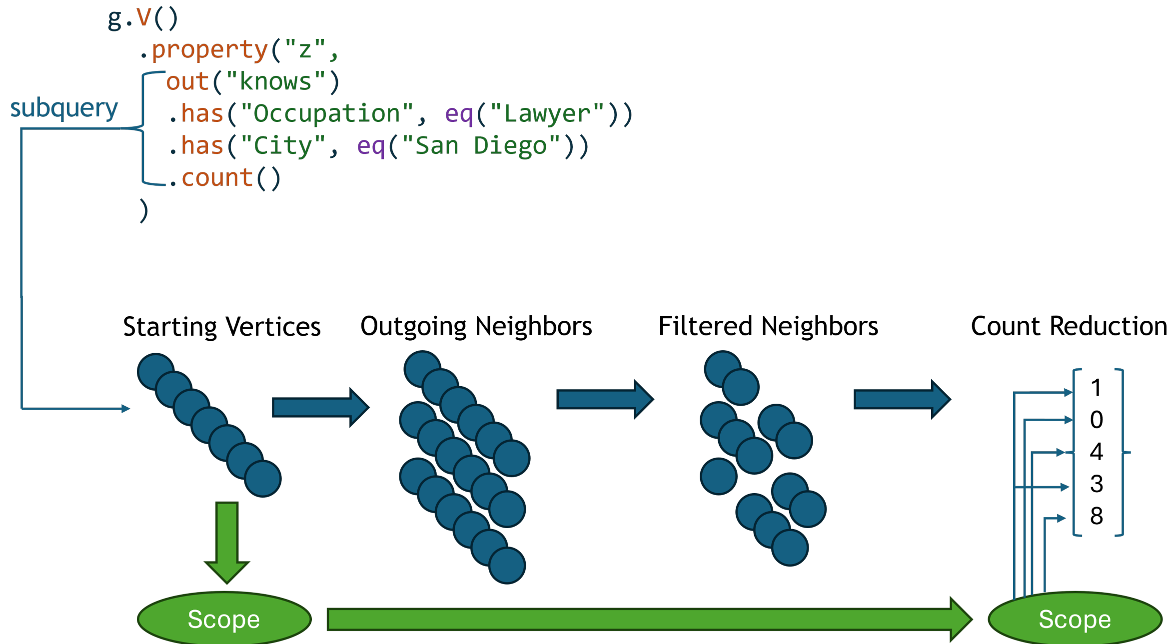


Figure 4.1: Scope application in Gremlin++. Gremlin++ stores the original vertices as a *scope* side effect, and starts the subquery as a new traversal from the original vertices. As the traversal progresses, each traverser inherits the scope side effect. When the reduction step is reached, the scope side effect of each traverser is accessed in order to properly compute the local count for each vertex. This allows a local operation to execute as a global operation, taking full advantage of the GPU and balancing the load across GPU work units. The query shown here is a simple Gremlin query that, for each vertex (presumably person), counts the number of lawyers they know who happen to live in San Diego and stores that count in the "z" vertex property.

gorithms. GraphScope, unlike Gremlin++, does not support end-to-end GPU acceleration for Gremlin queries.

4.2.3 BitGraph Framework

The BitGraph framework [28], discussed in detail in the previous chapter, is an ecosystem comprising three component libraries: *Maelstrom*, a library for GPU-accelerated, compute-agnostic, and type-erased data structures and algorithms, *Gremlin++*, an accelerated Gremlin query processing engine, and *BitGraph*, an accelerated graph backend that supports user interaction through the Gremlin++ query language. Gremlin++ decomposes each step in the query (traversal) into a series of low-level vector and matrix operations handled by Maelstrom. Gremlin++ uses an intermediate query representation based on the one provided by Java-Gremlin, but optimized for accelerating OLAP operations. Gremlin++ does especially well on queries that touch a large section of the graph, as opposed to GraphScope and Banyan, which are specialized for queries with many small subqueries that touch small, prunable subgraphs [66, 68].

4.2.4 Java-Gremlin

Java-Gremlin pioneered the original concept of the traversal [10], an intermediate query representation consisting of individual steps that roughly correspond to those inputted by the user. Java-Gremlin supports both OLAP and OLTP queries, and offers the *traversal strategy* framework for query optimization, which was also adapted by Gremlin++, with some key differences. These differences are discussed in this chapter.

4.3 How Graph Queries are Used

Today, property graphs and knowledge graphs have become somewhat (and erroneously) synonymous with graph databases [23]. This is in no small part due to the marketing of the database companies. Graph databases can store a very large amount of data, and deliver ACID compliance. For very basic analyst queries, such as querying the neighborhood of single vertex, they can be very effective. However, scaling to the kinds of queries needed to feed downstream models is much harder. GNNs, for instance, take many sampled subgraphs as input, and each subgraph is generated with many starting vertices, often thousands of starting vertices. Subgraphs can also include many neighbors across multiple hops. Biased random walks are another good example - running a query that performs a single random walk from one vertex is fast, but this begins to add up when performed for hundreds or thousands of vertices. Another example of real-world graph query usage is LLM inference. Deployed LLMs can have many user requests to process at once, and need to be able to retrieve relevant documents quickly, so they can append the appropriate context, avoiding hallucination and ensuring the user gets a trustworthy response.

Factoring in that enterprise-scale graphs can have billions of nodes and edges, having a performant, scalable graph query processing system is critical. Even before the rise of graph-based RAG and GNNs, practitioners of traditional ML faced the same problem. Querying a SQL database thousands of times during training or inference is too inefficient for a basic query engine. This led to the rise of *Spark* and *Dask*, frameworks that can quicky transform raw input data into ML-ready features using the same queries a traditional SQL engine can process [8, 52]. Spark and Dask work as a layer on top of an existing database, providing an accelerated solution that can produce quick results. They do not replace a traditional database - there is still a need

for an ACID-compliant, long-term storage solution, but Spark and Dask can quickly ingest and transform data from long-term storage to feed downstream models [49, 69].

Just as there is a solution to this problem for traditional machine learning, graph processing systems are the solution for large-scale graph analytics (Figure 1.6). This was previously discussed in Chapter 3. Graph processing systems started out by accelerating traditional graph analytics, such as centrality measurement, but have now grown to handle query processing and GNN input generation [68, 70]. Today, graph processing systems feed a diverse array of downstream models, including tree-based models, GNNs, probabilistic anomaly detection algorithms, and visualization engines.

Many graph databases have adopted a strategy called *dispatching* (Figure 4.2), which allows users to continue executing queries against the database, while dispatching another backend to handle more complicated queries. Gremlin-Java natively supports this; some Gremlin steps can implicitly execute *vertex programs*, which are executed using Spark-Gremlin (a different processing backend). Many databases' proprietary graph query languages also include dispatching support [25, 68, 71]. Backends used for dispatching do not need to support a query language, so long as they can accelerate a particular operation needed by the host system. For instance, a frontend might dispatch a connected components query to cuGraph [33] or Gunrock [34], sending only the necessary information about the graph structure needed to run the appropriate algorithm, and then transform the results into a format appropriate for the query once the dispatching backend completes its task.

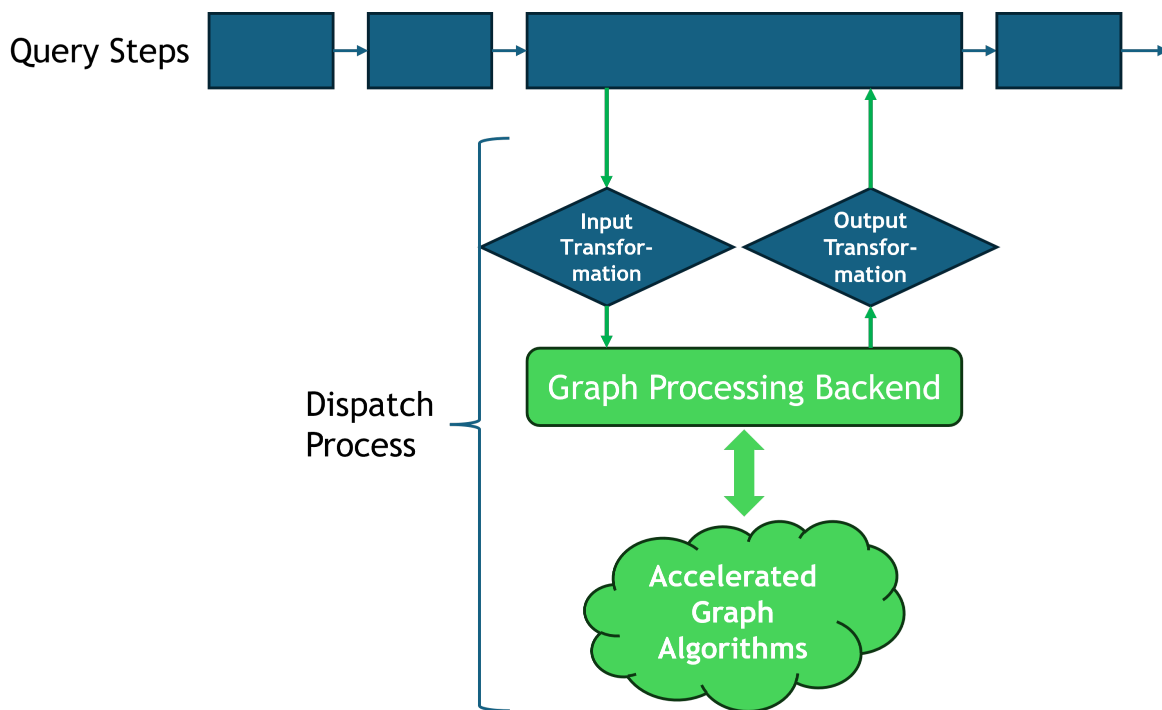


Figure 4.2: The dispatching process. The query processing backend processes each step in order, until reaching a step that it can dispatch. It then transforms the current data within the query, along with any other additional information about the graph needed by the backend, and sends that transformed data to the graph processing backend. The graph processing backend takes the input data, runs some accelerated algorithms that produce the information needed by the query step, and then returns the output to the query processing system. Finally, the query processing system transforms the output into a format appropriate for the query and proceeds to the next step.

4.4 Prerequisites to Optimization

When processing a query, the first step is translating the raw query into a machine-readable intermediate representation. This is analogous to the same process in Java, C/C++, and other compiled languages. Many query languages refer to this process as *query planning*, and there is a diverse array of literature on solving this problem for SQL-based queries [67, 72].

The intermediate representation used by Gremlin++ is a series of *traversal steps*, bulk-synchronous parallel (BSP) operations that transform input from a previous step. Each traversal step operates by taking a traverser set data structure as input [28], mutating it, and returning the new traverser set as output. Traverser sets are an abstraction based on the *traverser* concept in Gremlin. Gremlin traversers contain a traversed object, path information (what came before the current object in the traversal), and side effects (data stored with the traverser in previous steps). These are globalized in Gremlin++, as discussed in Chapter 3.

Once the intermediate representation has been produced, the optimization process can begin.

4.5 The Optimization Process

4.5.1 Overview

The optimization process transforms the intermediate representation of a query into an optimized version. I'll illustrate this using Gremlin++. Each step in Gremlin++ has a semantic definition, which specifies what the step expects as input and what it will return as output. Suppose we have a patent-citation graph such as the U.S. Patents 1975-1999 dataset [73, 74]. In this

graph, nodes represent patents, and edges represent citations (i.e. the source patent/vertex cited the destination patent/vertex). Now suppose we want to find patents applied for after 1970 with a self-citation rate of at least 60%, and order them by least-cited (lowest in-degree) to most cited (highest in-degree). To accomplish this, we would use a query like the one in Figure 4.3.

The step representation illustrated in Figure 4.3 is the starting point of the optimization process. Gremlin++ will apply various optimizations and transformations, known as *traversal strategies*, to produce the final optimized version [31]. Traversal strategies are applied just-in-time at the moment of query execution; this enables them to be aware of the state of the graph at runtime. Each traversal strategy looks at the entire array of steps, and is allowed to perform any number of mutations it deems appropriate, such as modifying steps, reordering steps, or removing steps, so long as it preserves the semantic meaning of the query. This behavior mirrors the equivalent optimization process for other query languages, as well as compiled programming languages [72].

Another use of traversal strategies is hiding implementation details, allowing the Gremlin++ query language to be backend-agnostic. Gremlin++ does not impose any restriction on how a graph backend stores its data, or where it executes operations on graph data. It only requires that graph backends implement the structure API and provide a small set of backend-specific steps and traversal strategies to replace default traversal steps with backend-specific ones [28, 31]. In addition to the required backend-specific steps, Gremlin++ also allows backends to add more traversal strategies and steps that can improve performance through awareness of the underlying graph structure [31]. Some of these will be discussed here.

Once all the traversal strategies, including backend-specific (in this case BitGraph-specific) strategies have been applied, the final traversal (Figure 4.4) is ready to be executed. For the rest

```
g.V()
  .has("APPYEAR", gte(1970))
  .has("SECDLWBD", gte(0.6))
  .order()
    .by(in().count())
  .limit(10000)
  .repeat(out()).times(2)
```

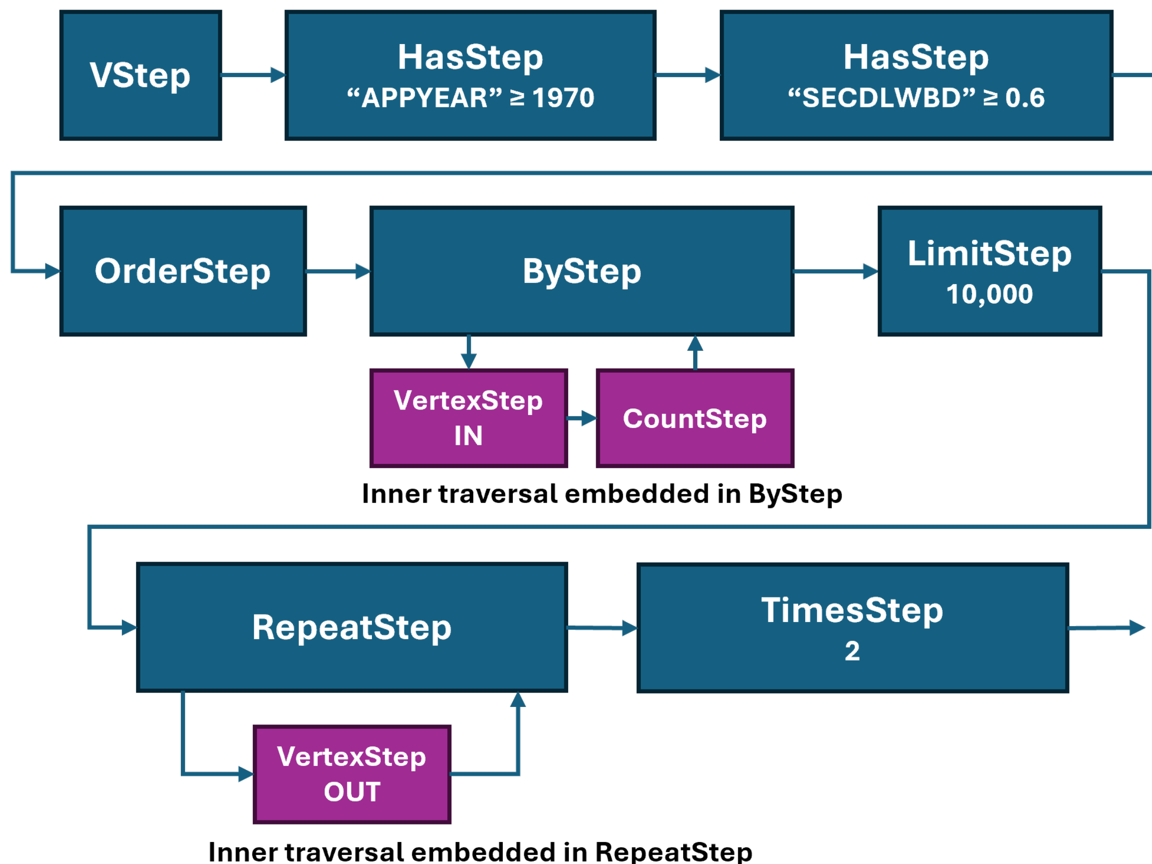


Figure 4.3: Translation of a Gremlin query to the immediate representation (traversal steps). This query searches for patents applied for after 1970 with a self-citation rate of at least 60%, and orders them from least-cited to most-cited. The initial intermediate representation consists of eleven Gremlin++ steps. Some of these steps are contained within other steps (the ByStep contains a VertexStep, followed by a CountStep and the RepeatStep contains a single VertexStep). This initial step representation needs to be optimized before it can be executed.

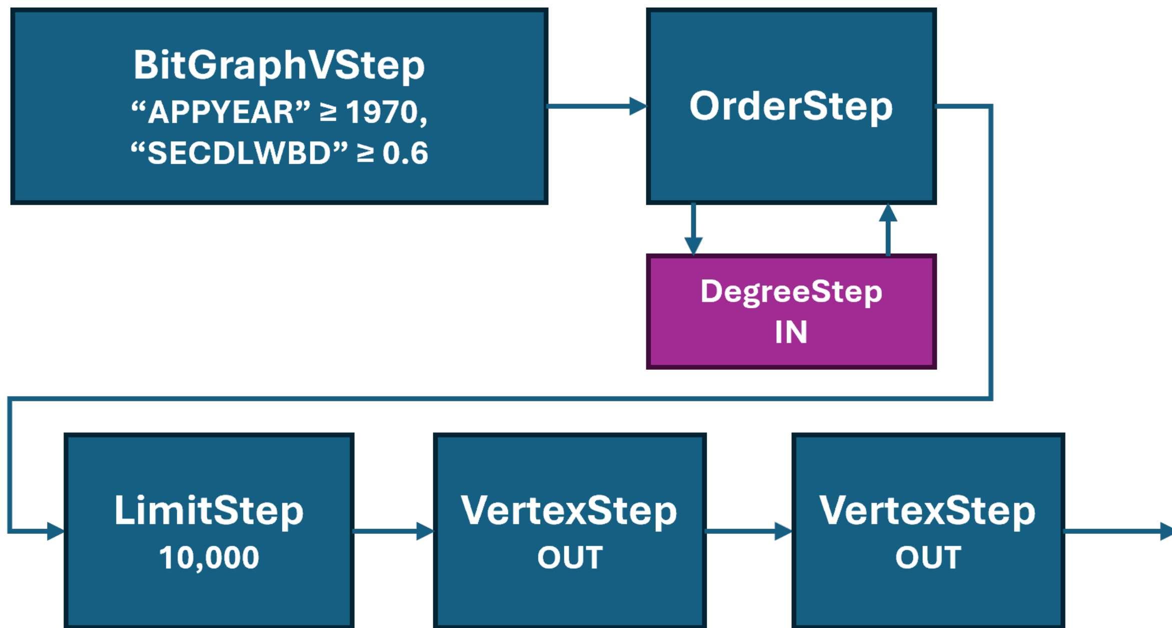


Figure 4.4: Final step representation of the Gremlin query from Figure 4.3

of this section, I’ll discuss how Gremlin++ reaches this final traversal from the initial version in Figure 4.3, highlighting the most important optimizations and their impact on query runtime.

4.5.2 Fusion

As the name suggests, *fusion* is the process of combining two or more query steps into a smaller number of steps. This is done in SQL with join combination and nested query flattening [72]. In graph query languages, the equivalents are filter combination [15], prior limit optimizations (applying a limit prior to returning the output to the next step) [72], and combining multiple paths within a projection [15, 31, 66].

The initial step array in Figure 4.3 has two consecutive *Has Steps*. Has Steps are filters that remove vertices or edges based on vertex or edge property values. Gremlin++ combines these two steps using the *HasJoinStrategy*, shown in Figure 4.5.

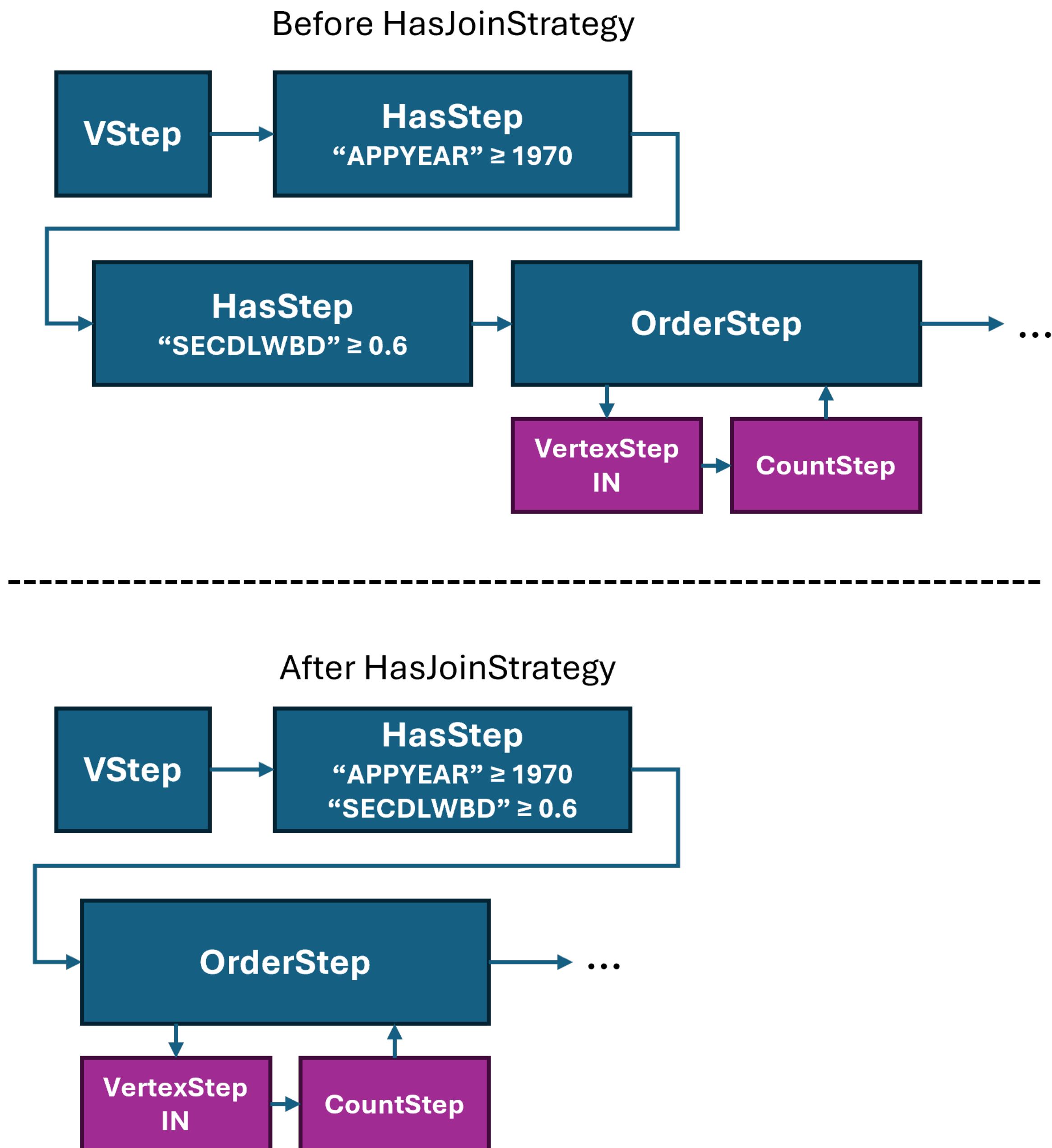


Figure 4.5: Application of HasJoinStrategy to the query shown in Figure 4.3. Only the affected portion of the original query is shown. The number of Has Steps is reduced from two to one, and the filters are applied together, skipping a call to *advance*, which would unnecessarily update the traverser set. This results in only a small performance improvement, but enables more impactful downstream optimizations.

On its own, *HasJoinStrategy* does not do much to accelerate the traversal, yielding a tiny 1% speedup (Figure 4.9). However, it is critical in enabling downstream optimizations, such as the reordering optimizations I'll discuss next [31].

4.5.3 Reordering

Like fusion, *reordering* can easily be mapped to its equivalents in SQL [66, 67]. Because one filter may cull more rows than another, the order in which filters are applied can significantly alter performance. Consider this example from [31]:

"Consider a database of products, where each product has a cost and warehouse location. Suppose we wanted to display a list of products less than ten dollars available at Warehouse K. If there are only 14 warehouses, and thousands of products less than ten dollars, then the best filter order would be to filter by warehouse, then by product (assuming a more or less uniform random distribution of products to warehouses). This is because filtering by warehouse would reduce the number of possible products the most."

The original query (Figure 4.3) filters by *APPYEAR* first; however, this is probably not a good first filter since most of the patents were filed after 1970 and it will eliminate very few traversers. This is impossible to know at compile-time, but since traversal strategies are applied just-in-time at runtime, and it is very cheap to get a count of the number of vertices with given property, it is a good idea to reconsider the filter order right before executing the step [31]. This optimization extends the initial 1% speedup from the *HasJoinStrategy* to about 10% (Figure 4.9).

4.5.4 Loop Unrolling

Loop unrolling is a well-known optimization in the field of imperative programming languages [75], but is not common within traditional SQL-based query languages. As discussed earlier, Gremlin++ incorporates many features from functional programming languages in addition to traditional query language syntax [10].

Like fusion and reordering, loop unrolling can either yield performance improvement on its own, or enable more downstream optimizations. Direct performance improvement comes through elimination of the context required to jump from the end of a loop to the beginning of the next loop. If there are few enough iterations such that overloading the stack isn't a concern, removing the additional context and bookkeeping, eliminating end condition evaluation, and writing the loop as a sequence of operations may be significantly faster. Downstream performance improvement comes from reevaluating the complete sequence of steps once it has been unrolled, and checking whether fusion or reordering can be performed [67, 15, 66].

The result of the loop unrolling process (as implemented by Gremlin++'s *RepeatUnrollStrategy* [31]) is shown in Figure 4.6. Applying *RepeatUnrollStrategy* boosts the cumulative speedup to 26% (Figure 4.9).

4.5.5 Pattern Detection

Like loop unrolling, *pattern detection*'s closest analogues are from imperative programming language rather than SQL [66, 75]. The pattern detection optimization groups a large number of steps and encapsulates them in a specific algorithm. While similar to fusion, pattern detection is able to take a completely different codepath, potentially even one outside the query

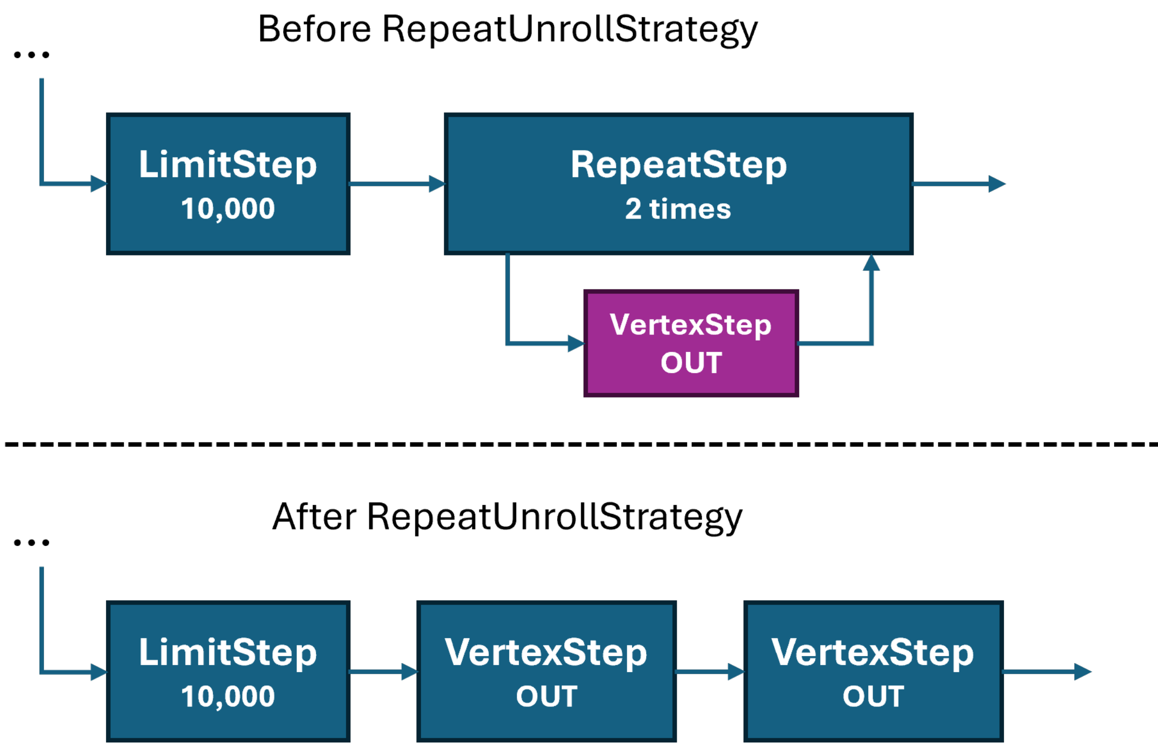


Figure 4.6: Application of RepeatUnrollStrategy to the query shown in Figure 4.3. Only the affected portion of the original query is shown. The original loop, set to run for two iterations, was split into two consecutive calls to the VertexStep. This eliminated the overhead associated with loop end condition checking and context management, resulting in significant speedup.

language, whereas fusion can only rewrite a sequence of steps using defined steps of the query language. Many types of dispatching (Figure 4.2) are combined with pattern detection; for instance, Java-Gremlin can dispatch some step combinations to a *vertex program* to be executed by Spark-Gremlin [15]. Graph databases can use pattern detection in conjunction with dispatching to execute algorithms in an external graph processing system [15, 25, 71].

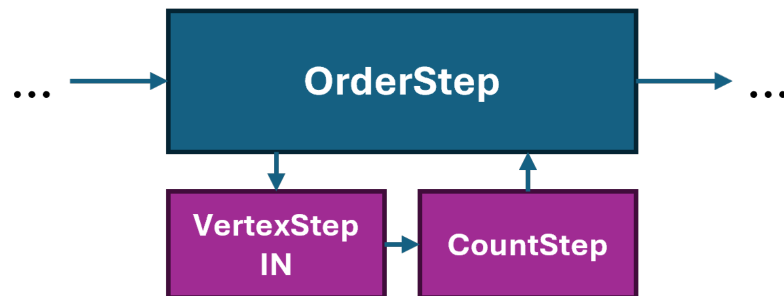
Continuing the optimization process from where we left off in Figure 4.6, Gremlin++'s *BasicPatternExtractionStrategy* [31] identifies patterns that can be replaced with new sequences of steps (Figure 4.7). This brings the cumulative speedup to 36% (Figure 4.9).

4.5.6 Backend-Specific Optimizations

As I briefly mentioned earlier, *backend-specific optimizations* are a critical requirement for Gremlin++ supporting backends [31]. For instance, the *V Step* in Gremlin++, which extracts vertices from the graph, is a stub that raises an exception when the query engine attempts to execute it. It is intended to be replaced by backends like BitGraph, since Gremlin++ does not know what the graph structure is, leaving it up to the backend to handle data storage and transfer. BitGraph uses the *BitGraphStrategy* to replace stub V Steps with *BitGraphVStep*. This backend-specific step knows how to access data in BitGraph efficiently.

In addition to this stub replacement, backends can also introduce other optimizations that are unique to their framework and API. BitGraph also introduces the *BitGraphSelectionStrategy* [31], which combines a BitGraphVStep with adjacent Limit Steps or Has Steps, executing these filters are part of the vertex query into BitGraph's canonical graph format (discussed in Chapter 3). Again, this is like a fusion operation, but is considered separate because it involves knowledge

Before BasicPatternExtractionStrategy



After BasicPatternExtractionStrategy

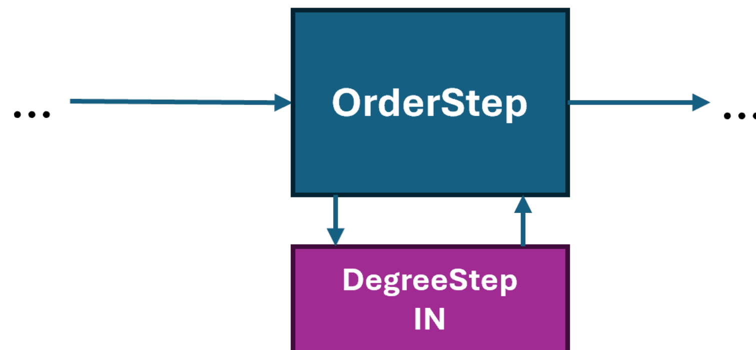


Figure 4.7: Application of BasicPatternExtractionStrategy to the query shown in Figure 4.3. Only the affected portion of the original query is shown. Here, the traversal within the OrderStep, which performs a VertexStep and CountStep to determine in-degree, is replaced with a new step that calculates the in-degree. This new step is not part of standard Gremlin++.

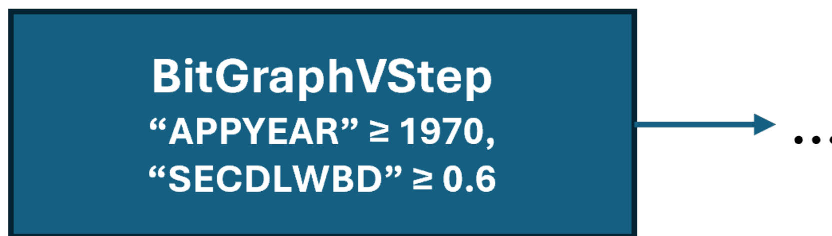


Figure 4.8: Application of BitGraphStrategy and BitGraphSelectionStrategy to the query shown in Figure 4.3. Only the affected portion of the original query is shown. Note that the reordering optimization is still applied here, even though it is not explicitly shown.

of the inner workings of BitGraph and its canonical graph format. Regular fusion operations are backend-agnostic.

Figure 4.8 shows the transformed query from Figure 4.3 after the BitGraphStrategy and BitGraphVStep are applied. These backend-specific strategies have the highest cumulative effect, bringing the total speedup to 102% (Figure 4.9).

4.5.7 Overall Impact

As illustrated by Figure 4.9, traversal strategies depend on each other to achieve optimal speedup. Many, such as reordering, would not be possible without prior fusion and loop unrolling optimizations [31]. Even backend-specific optimizations rely on non-backend-specific optimiza-

tions that eliminate loops or reorder filters. This applies not just to traversal strategies within Gremlin++, but broadly to the full ecosystem of graph query languages [67]. Now that I have introduced these optimizations and what they can achieve, the next step is to show how they apply to a specific use case.

4.6 RAG Case Study

4.6.1 Overview

Retrieval Augmented Generation (RAG) is the process by which a user prompt to a large language model (LLM) is combined with relevant context. In graph-based RAG, this context is stored in a knowledge graph which has been constructed from a large corpus of documents. Vertices in this knowledge graph typically represent documents and paragraphs. Edges represent linkages between documents, such as hyperlinks or references. Constructing this knowledge graph is a complex problem that itself could easily be the subject of a complete dissertation.

For this case study, I focused solely on the extraction component of the RAG, leaving out the LLM and knowledge graph construction pieces [31]. These are discussed in Chapter 5, as part of my research into a new type of RAG built on top of the BitGraph framework. This case study served as a critical test of whether the BitGraph framework could meet the challenge that my future work would pose. The pure RAG workflow used in this case study is show in Figure 4.10. This workflow is an example of a *direct input* RAG [76], meaning that textual context extracted from the RAG process is combined with the prompt and fed directly into the LLM. This is in contrast to combined RAGs, which encode the returned context and concatenate the resulting embeddings with those from the LLM [35, 41]. Again, combined RAGs are discussed

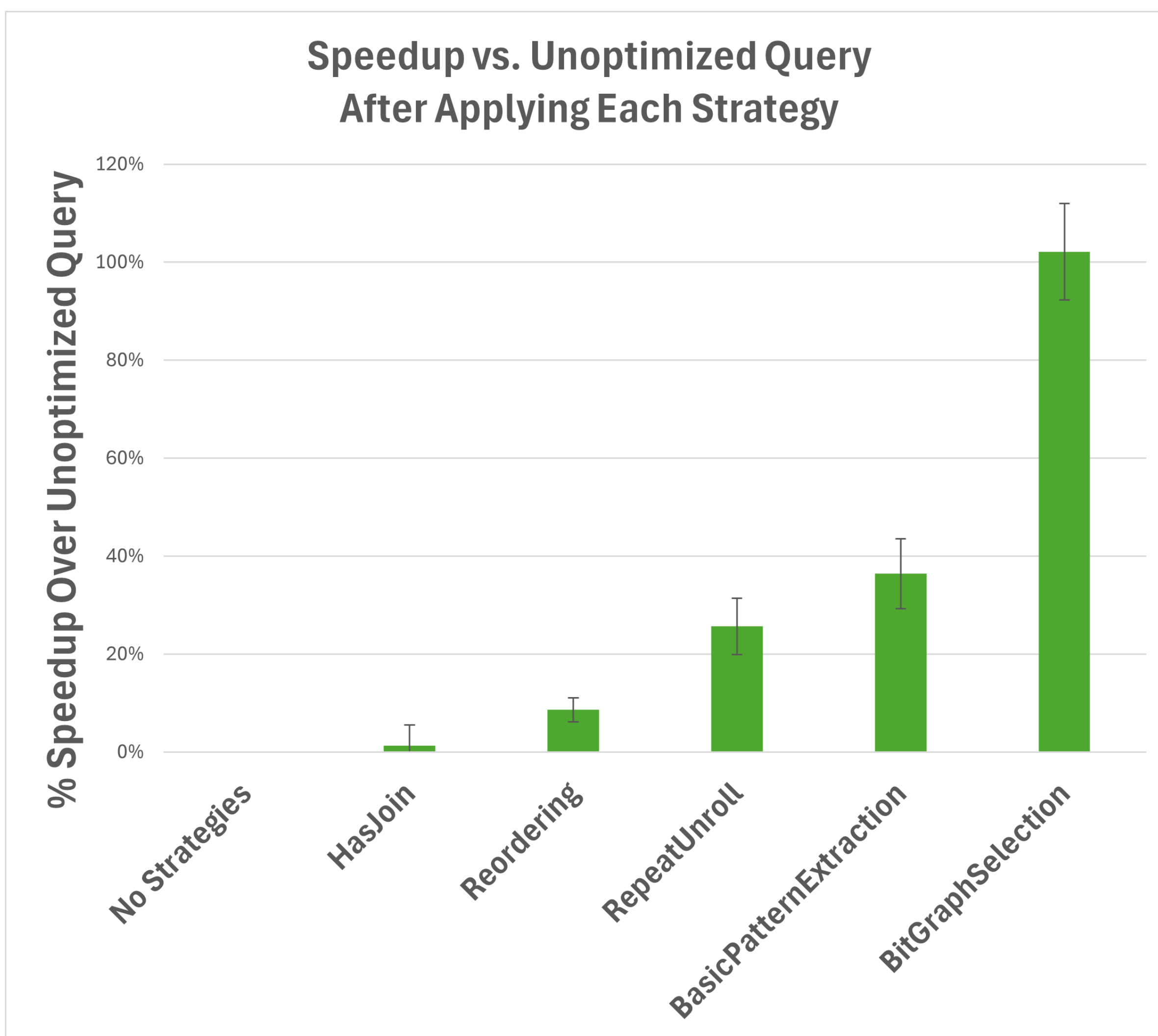


Figure 4.9: Cumulative speedup of each traversal strategy (optimization) on the query from Figure 4.3. This benchmark was run on the U.S. Patents 1975-1999 dataset [74, 73], which has 6 million vertices and 6.5 million edges. Each optimization builds on the previous, many achieving only a small impact on their own, but enabling more impactful downstream optimizations. Backend-specific optimizations, which have direct knowledge of the underlying graph structure, result in the highest speedup. After all strategies are applied, the query is over 100% faster [31]. This benchmark was run on using an overclocked Intel i7-12700K CPU and NVIDIA RTX A6000 GPU.

in more detail in the following chapter.

4.6.2 Benchmark Description

Based on the use case illustrated in Figure 4.10, Figure 4.11 shows a simple query that a direct input graph-based RAG (DGR) might execute during the additional document retrieval stage [31]. This query is parameterized - its behavior can be adjusted based on two variables: *fanout* and *vertex_ids*. Fanout refers to the maximum number of vertices visited per hop, where "hop" here means to a single move from one vertex to a neighboring vertex across an edge (i.e. a fanout of $[5, 10]$ means 5 vertices are selected in the first hop, and 10 are selected in the second hop). The number of hops is fixed in this query. The vertex_ids parameter comes from the set of starting vertices that were found using vector similarity. These variables are similar to those in neighborhood sampling and subgraph extraction algorithms, which are often used by downstream models like GNNs to produce a set of input batches. One advantage of using a query language here is that these variables are easy to manipulate, and can be extended as necessary. I benchmarked three combinations of these variables: first, 10 input vertices (seeds) with a fanout of $[10, 10]$, then 5 input vertices (seeds) with a fanout of $[5, 5]$, and finally 5 input vertices (seeds) with a fanout of $[10, 10]$.

4.6.3 Benchmark Results

Figure 4.12 shows the results from running fully-optimized and non-optimized version of the query in Figure 4.11 with the specified number of vertices and fanout values [31]. Speedup was highly-dependent on the values of these parameters. The best-performing query was the

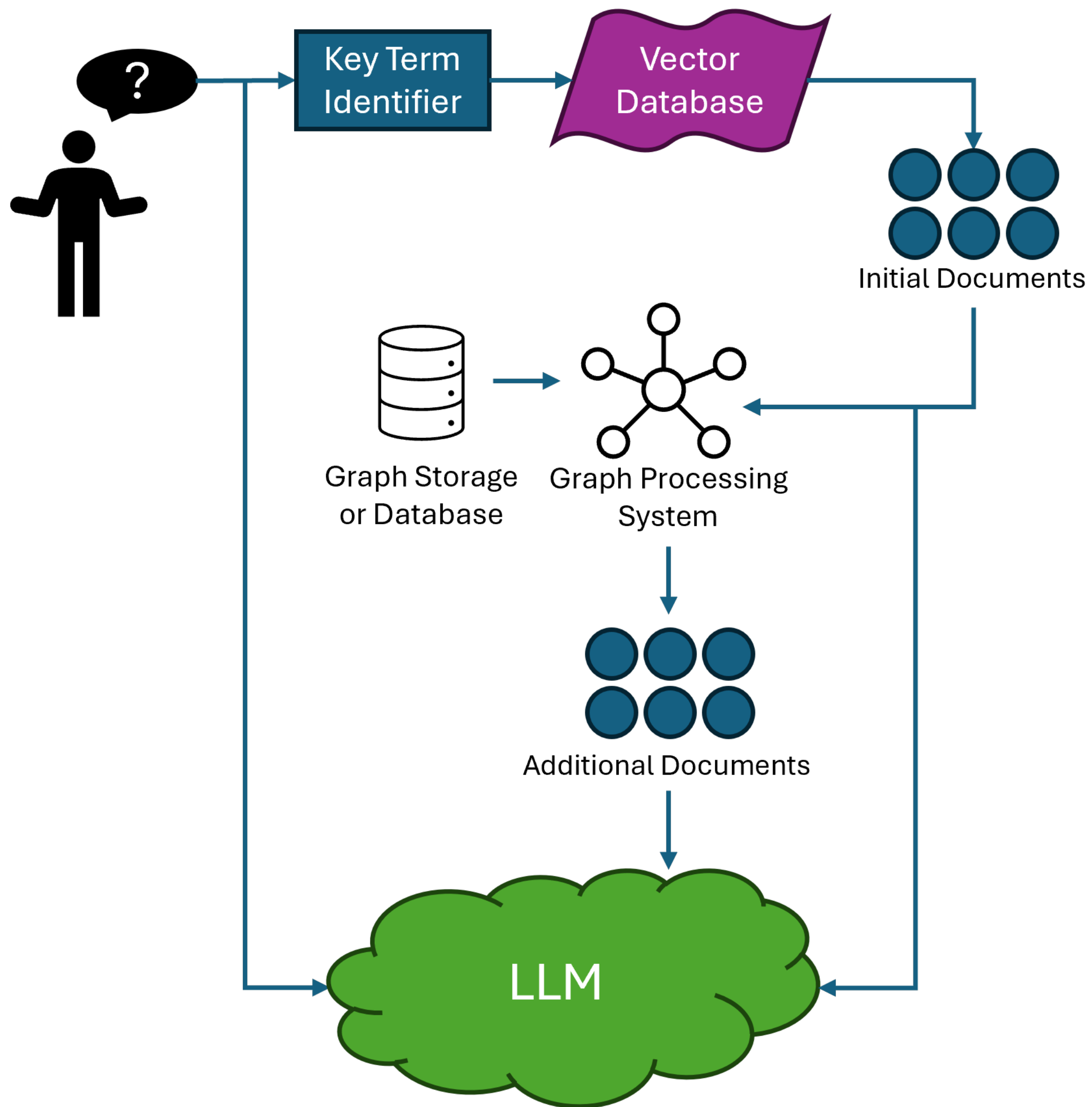


Figure 4.10: Workflow diagram showing a direct input graph-based RAG. The user prompt first passes through a named entity recognition (NER) pipeline [77], which outputs a result of key terms. These key terms are converted to vector embeddings, which are used to identify initial documents through a basic vector similarity search. These initial documents correspond to vertices in the graph. From these initial vertices, a graph query is executed to identify other vertices of interest in the neighborhood of the initial vertices. The resulting set of vertices corresponds to a set of final documents, which are used as additional context for the prompt. This additional context is combined with the prompt, and the combined text is fed into the LLM. A more complex version of this workflow was used for the query-based RAG described in Chapter 5.

```
g.V(vertex_ids)
  .emit(identity())
  .repeat(
    out().dedup()
    .order()
    .by(out().count())
    .limit(fanout)
  ).times(2)
  .dedup()
  .order()
    .by(out().count())
```


Figure 4.11: Basic parameterized query that retrieves additional documents from a set of initial documents (vertices) as part of a direct input RAG. This query uses the *vertex_ids* and *fanout* parameters to control which vertices and edges are traversed, and therefore which documents are selected as the final context.

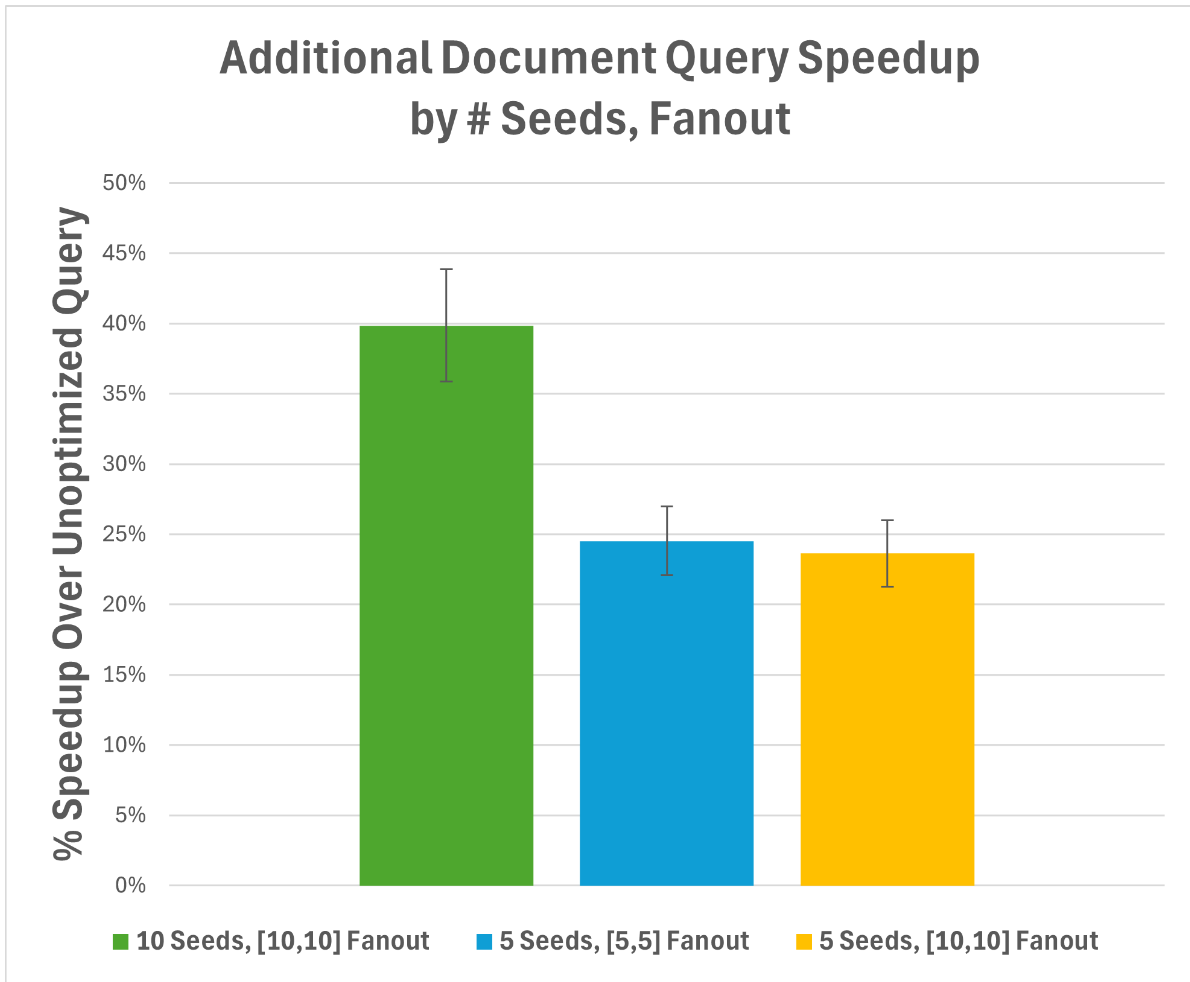


Figure 4.12: Simple RAG Query Benchmark Results - this benchmark was run using an overclocked Intel i7-12700K CPU and NVIDIA RTX A6000 GPU.

one with a higher number of starting vertices and fanout, which resulted in a larger subgraph. This is consistent with other results discussed in Chapter 3, where larger, more connected graphs generally saw more speedup from using the BitGraph framework. All versions of the query still achieved noticeable speedup, with the best query achieving 40% speedup over the unoptimized version.

4.7 Discussion

In this component of my dissertation, I formally introduced the concept of graph query optimization and walked through each type of optimization, using Gremlin++ to illustrate how each optimization is applied to the intermediate representation of a query. I showed how optimizations interact with each other to achieve a higher speedup than what each could achieve individually, again using Gremlin++ to show the cumulative speedup after each optimization, starting with a tiny 1% speedup and growing to a sizeable 102% speedup after all optimizations were applied. I then showed how these optimizations work in practice, using a RAG use case to benchmark the effects of query optimization on a parameterized RAG query. This benchmark showed that the optimization process as implemented in Gremlin++ performs well on a real-world query, and that the BitGraph framework was ready to be further extended to solve the important real-world problem of large-scale retrieval augmented generation.

Chapter 5: Development and Performance of a Query-Based Graph RAG

5.1 Overview

This component of my dissertation discusses the development of a new type of RAG that extends the framework of G-Retriever [35] to an end-to-end graph query-based RAG incorporating *prize-aware graph traversal*, which I named QRAG. It is mostly based on work published in [43], and builds on top of the work discussed in Chapter 3 and Chapter 4. Unlike the two preceding chapters, the work analyzed in this chapter goes beyond simply accelerating existing queries and algorithms, and offers a new, unique model for graph-based retrieval augmented generation with enhanced tunability.

5.2 Motivation

Over the past year, G-Retriver [35] quickly became a foundational framework in graph-based RAG. G-Retriever was the first framework to effectively use GNNs for graph-based RAG, and it proved that encoding the graph structure as additional input to the LLM was worthwhile, further igniting the battle between direct-input and combined RAGs. One of the most interesting components of G-Retriever was its incorporation of the *prize-collecting steiner tree* (PCST) algorithm for pruning the initial subgraph [78]. G-Retriever used a naive approach for selection

of the initial subgraph, and pruned it with PCST before encoding it to reduce noise. This made me wonder whether there was a way to get a better subgraph in the first place. The problem with existing subgraph extraction algorithms is that they are not *prize-aware*; in other words, they hop around the graph and grab vertices and edges without knowing if they are gaining any useful information at all. Perhaps for small graphs this is not a major concern, but for massive-scale enterprise knowledge graphs, this could quickly run into issues, because there are just so many possible vertices and edges to choose from. The naive approach would take too long, and grab too much data; even though the PCST algorithm is fast [35, 78], the graph extraction leading up to it would be impossibly slow.

In [31], I showed that RAG queries could be accelerated well, but I wanted to go beyond this. I decided to create an alternative to the naive-subgraph-and-PCST solution used by G-Retriever that would use a graph query capable of evaluating the prize as it traversed the graph. Doing so would involve further modifications to the BitGraph framework, as well as additional work to integrate it with PyTorch, PyTorch Geometric (PyG), and FAISS [43]. These modifications are discussed in detail in this chapter.

5.3 Background and Related Work

5.3.1 Retrieval Augmented Generation

5.3.1.1 RAG Overview

Retrieval Augmented Generation (RAG) is an information retrieval and extraction process used by large language models to prevent hallucination and avoid expensive model retraining.

RAG works by analyzing a user prompt, identifying information that would be relevant in answering the prompt, such as news articles or academic papers, and *augmenting* the user's prompt with this additional context [41].

RAG is most frequently used to keep pretrained LLMs up-to-date [41, 79, 80]. LLM training is very expensive, requiring thousands of GPUs [37], but information indexing is relatively cheap in comparison [35, 41, 80]. News breaks in a matter of minutes, new papers are published daily, and outdated content is purged instantly across the internet, so keeping a good index is critical. In the field of LLMs, this index is usually referred to as a *corpus* of documents, where documents are passages extracted from larger texts. RAG can also reduce the size of LLMs by acting as a non-parametric data store, allow pretrained models to answer questions on a proprietary nonpublic knowledge base [80], and speed up model training [41].

RAGs can be divided into two types: *direct-input* and *combination*. Direct-input RAG works by directly adding text to the original prompt. Combined RAG uses a separate model to encode additional information and concatenate the resulting embeddings with those from the LLM. I'll discuss these two types in the context of graph-based RAG.

5.3.1.2 Graph-Based RAG

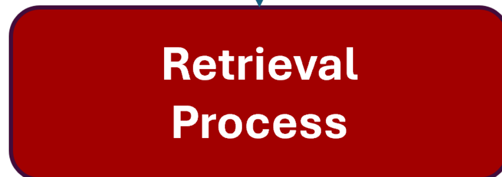
Many RAG implementations use a form of naive vector search to determine which documents to select from their index [81]. This can often miss the complete context required by a prompt, especially for *multi-hop* queries, which involve adjacent information [82, 83]. Simply retrieving the closest document is not sufficient; the full context requires more information. For instance, if a user asked an LLM about composers of the Baroque period, it would need to re-

trieve not just the article on the Baroque period, but presumably articles about Johann Sebastian Bach and Jean-Baptiste Lully [84] as well. Graph-based RAG helps overcome this challenge by linking related articles together in a knowledge graph, so those "hops" can be made more easily [2, 35, 41, 42].

There are both direct-input and combination versions of graph-based RAG [2, 31, 41, 42]. *Direct input graph-based RAG* (DGR) (Figure 5.1) incorporates vector search for finding the initial vertices, which correspond to documents in the index/corpus [2, 41]. From these initial vertices, edges corresponding to linkages between documents are traversed to find additional documents [31, 2, 41]. As illustrated in Figure 5.1, DGR uses named entity recognition [31, 77] on the input prompt to get a list of entities to search for in the document index. Using a vector search method, such as top-k cosine similarity, these entities are matched to documents, which form the list of starting vertices in the knowledge graph [2, 31]. Then, a graph traversal is executed starting from these seed vertices to select more vertices in the neighborhood of the seeds. Types of traversals used for this task include random walks [85], neighborhood sampling algorithms [20], as well as query-based parameterized traversals [3, 31]. The text in the documents corresponding to the retrieved vertices is then joined with the input prompt and inputted into the LLM.

In *combination graph-based RAG* (CGR) (Figure 5.2), the process begins just as it does in DGR, with a set of seed vertices selected through vector search to select more vertices [2, 3]. Unlike DGR, however, the output of the traversal is used not only to obtain more vertices, but to encode the structure of the subgraph resulting from the traversal [2, 3, 35, 42]. This can be done using a *graph attention network* (GAT) [86] or other GNN model. The encoded graph is concatenated with the embedding of the original text, or in some cases both the original and

Question: Do both directors of films The Big Bang (1989 Film) and Tender Fictions share the same nationality?



Question: Given the information below, Do both directors of films The Big Bang (1989 Film) and Tender Fictions share the same nationality?

- The Big Bang is a 1989 documentary film, directed by Academy Award- nominated screenwriter James Toback
- James Toback(born November 23, 1944) is an American screenwriter and film director
- Tender Fictions is a 1996 autobiographical documentary film directed by American experimental filmmaker Barbara Hammer
- Barbara Jean Hammer(May 15, 1939 – March 16, 2019) was an American feminist filmmaker

Figure 5.1: Illustration of a direct-input graph-based RAG (DGR) using data from 2WikiMultihopQA [83]. The user prompt is fed through a graph-based retrieval process, which identifies additional relevant documents to combine with the input prompt. The combined text is then inputted into the LLM.

retrieved text [35], and is taken as input to the LLM.

5.3.2 G-Retriever

G-Retriever [35] is a graph-based RAG framework that uses a GNN (specifically GAT [86]) to produce an encoding of a graph containing relevant context to a user prompt. The authors of the paper describe this as "chat with your graph" [35]. The G-Retriever paper tests their model against several datasets, including *ExplaGraphs* [87], *SceneGraphs* [88], and *WebQSP* [35, 89, 90]. These datasets contain input graphs along with related prompts; the process of extraction is performed on these input graphs, and therefore on a much smaller scale than the complete knowledge graph derived from 2WikiMultihopQA [83] I used to train QRAG.

G-Retriever incorporates the prize-collecting steiner tree algorithm [35, 78] into the retrieval process, using it to prune the size of the input subgraphs. This algorithm finds a fully-connected subgraph with the highest amount of information contained within its vertices (prize) while minimizing the number of edges based on an assigned edge cost. G-Retriever uses cosine similarity between the embedding of the document corresponding to each node and the embedding of the original prompt to determine node prize values; the edge cost is a fixed value that can be tuned by the user [35].

G-Retriever is a CGR, taking the original prompt plus a textual representation of the graph, and augmenting it with the GNN-created embedding [35]. To build the G-Retriever framework, the authors first fine-tuned the LLM on ground truth data similar to the input of G-Retriever using LoRA [38]. Then they trained the LLM using the complete retrieval process [35]. This was proven to be the most effective approach; fine-tuning before training yielded up to a 5% increase

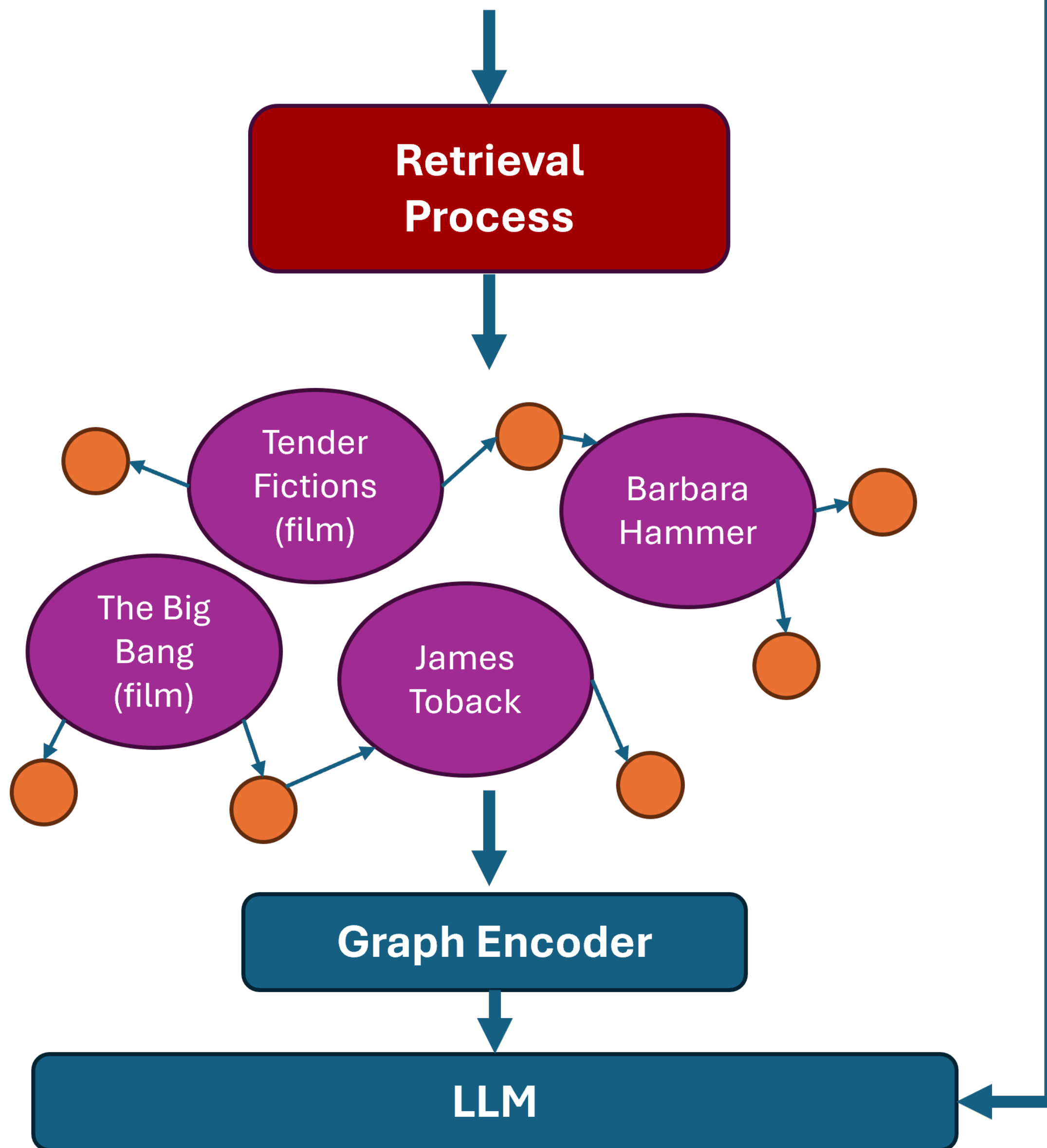


Figure 5.2: Illustration of a combined graph-based RAG (CGR) using data from 2WikiMultihopQA [83]. The user prompt is fed through a graph-based retrieval process, which creates a subgraph comprised of the neighborhood around the starting (seed) vertices. This subgraph is then transformed into an embedding using a graph encoder, and fed into the LLM along with the encoded prompt. In some cases, the same process used for DGR is performed in addition to CGR, resulting in both an encoded augmented prompt and graph embedding taken as input to the LLM.

in model accuracy [35]. The fully-trained G-Retriever (GNN+LLM) model resulted in a 2% accuracy increase on the ExplaGraphs benchmark, 12% accuracy increase on the SceneGraphs benchmark, and 10% accuracy increase on the WebQSP benchmark [35].

Over the past year, G-Retriever has quickly become a fundamental framework for graph-based RAG. The QRAG process described in this chapter is directly derived from G-Retriever, but is extended to a much larger knowledge graph, and includes a few other modifications, such as replacement of the naive-subgraph-and-PCST method with a query-based subgraph extraction method.

5.3.3 HotpotQA and 2WikiMultihopQA

HotpotQA [82] is a dataset for benchmarking multi-hop question answering performance. It was created in response to the lack of benchmark datasets available for multi-hop question answering. It also comprises a large enough corpus to construct a proper knowledge graph. HotpotQA was constructed from the articles comprising the English version of Wikipedia, with only the hyperlinks from the first paragraph of each article included (this was done to reduce noise, since the authors determined that these links were the most relevant) [82]. Then, a question generation process was used to create the prompts. This was primarily done by sending random pairs of articles to users on Amazon Mechanical Turk, and asking them to write either a bridge or yes/no question between them while also listing the supporting facts [82]. Bridge questions were of the form *"When was the singer and songwriter of Radiohead born?"*; the bridge entity in this case would be the singer in question, *"Thom Yorke"* [82]. Yes/no questions used a list of categories; for instance *mountains* was a category derived from the list of highest mountains from

Wikipedia [82]. Two entities from the same list of categories were randomly chosen and sent to the Amazon Mechanical Turk users; for instance *"Michael Jordan"* and *"Kobe Bryant"* from the basketball category. Users then generated the yes/no questions of the form *"Who has played for more NBA teams, Michael Jordan or Kobe Bryant?"* [82].

HotpotQA was not directly used in the benchmarks described in this chapter. Instead, I used *2WikiMultihopQA* [83], which is directly derived from HotpotQA. In addition to the original Wikipedia data included in HotpotQA, 2WikiMultihopQA adds additional linkage information obtained from *WikiData*. This linkage information is used to construct *evidence*, which is an explanation of the path taken from the prompt to the answer; in other words, a more graph-friendly set of supporting facts. Using work done by other researchers [91], the authors of 2WikiMultihopQA eliminated prompts from HotpotQA that they did not consider to be true multi-hop prompts. 2WikiMultihopQA is comprised of four types of prompts: *comparison*, *inference*, *compositional*, and *bridge-comparison* [83].

1. Comparison prompts compare two or more entities from the same group; e.g. *"Who was born first, Albert Einstein or Abraham Lincoln"* [83].

2. Inference prompts are combinations of two triples (equivalent to edges in the knowledge graph representation); e.g. *(Abraham Lincoln, mother, Nancy Hanks Lincoln); (Nancy Hanks Lincoln, father, James Hanks)* → *(Abraham Lincoln, maternal grandfather, James Hanks)*. These triples are translated into question form; e.g. *"Who is the maternal grandfather of Abraham Lincoln?"* [83].

3. Compositional prompts are similar to inference prompts, but there is no "combined" relation between entities. For instance, while "maternal grandfather" is a defined combined

relation, no such relation exists for the triples *(La La Land, distributor, Summit Entertainment); (Summit entertainment, founded by, Bernd Eichinger)*. The resulting question lists both relations: *"Who is the founder of the company that distributed La La Land?"* [83]

4. Bridge-comparison prompts follow the same structure as those from HotpotQA [82, 83].

The prompts for 2WikiMultihopQA were generated by applying a set of rules (i.e. $spouse(a, b) \wedge mother(b, c) \rightarrow mother_in_law(a, c)$) to the filtered set of prompts from HotpotQA, cross-referencing the data with WikiData, and transforming the input prompts into the output prompts based on the ruleset and WikiData-derived relations [83].

5.3.4 FAISS

FAISS [92] is a library for large-scale vector similarity search. It indexes a large set of *embeddings*, which are vector representations of data such as text, graphs, or media typically produced by CNNs [92]. Embeddings are typically used as input to downstream models, such as image classification models, or, as is the case in this chapter, large language models. Embeddings are increasingly stored in large *database management systems* (DBMS) as properties of entities such as users, products, or images, and can also be used to search for matching entities in a cost-effective manner [92]. This type of search is powered by *approximate nearest-neighbor search* (ANNS) algorithms. The goal of FAISS is to accelerate these algorithms.

FAISS provides both a Python and C++ API (the work described in this chapter used the C++ API) [43, 92]. FAISS also offers GPU acceleration [93] for some types of index methods. There are many index methods available in FAISS, such as *IndexFlat*, which implements brute force nearest-neighbors, to IVFPQ which is a highly-compressed index [92, 94]. *Inverted file*

(IVF) indexing was the most relevant type of indexing to this work. It works by clustering the vectors stored in a database at the time of indexing [92, 94] using a set of centroids produced by a coarse quantizer. The resulting centroids are stored into inverted lists to form the inverted file.

FAISS allows IVF indexing to be used along with *product quantization* (PQ) [92, 94, 95]. Product quantization splits vectors horizontally into chunks, creating a list of m subvectors for each vector in the index [94]. Clustering is then performed on each subcomponent to get a list of centroids for each subcomponent. Product quantization significantly reduces the complexity of learning the quantizer with minimal information loss [95]; when used with IVF, the new centroids are the concatenation of the PQ centroids [92, 95]. The authors of FAISS strongly recommend this combination, called IVFPQ, for large-scale GPU indexing [92, 93], and it is what I used when integrating FAISS with BitGraph [43].

5.4 High-Level Design and API

A primary goal in building QRAG was to encapsulate as much of the RAG process into Gremlin++ so it would be accessible to a typical user [43]. However, when I began this work, Gremlin++ lacked several important features needed to enable RAG. First, the Gremlin++ language and its structure API did not have a concept of embeddings. Embeddings, as discussed earlier, are non-sparse learnable encodings for vertices and edges in the graph. They are stored in a tensor data structure rather than a map data structure, as is the case for properties. To incorporate embeddings into the API of Gremlin++, I first added new steps to access and modify them.

The *Encode Step* [43], shown in Figure 5.3 maps a vertex or edge in the graph to its embed-

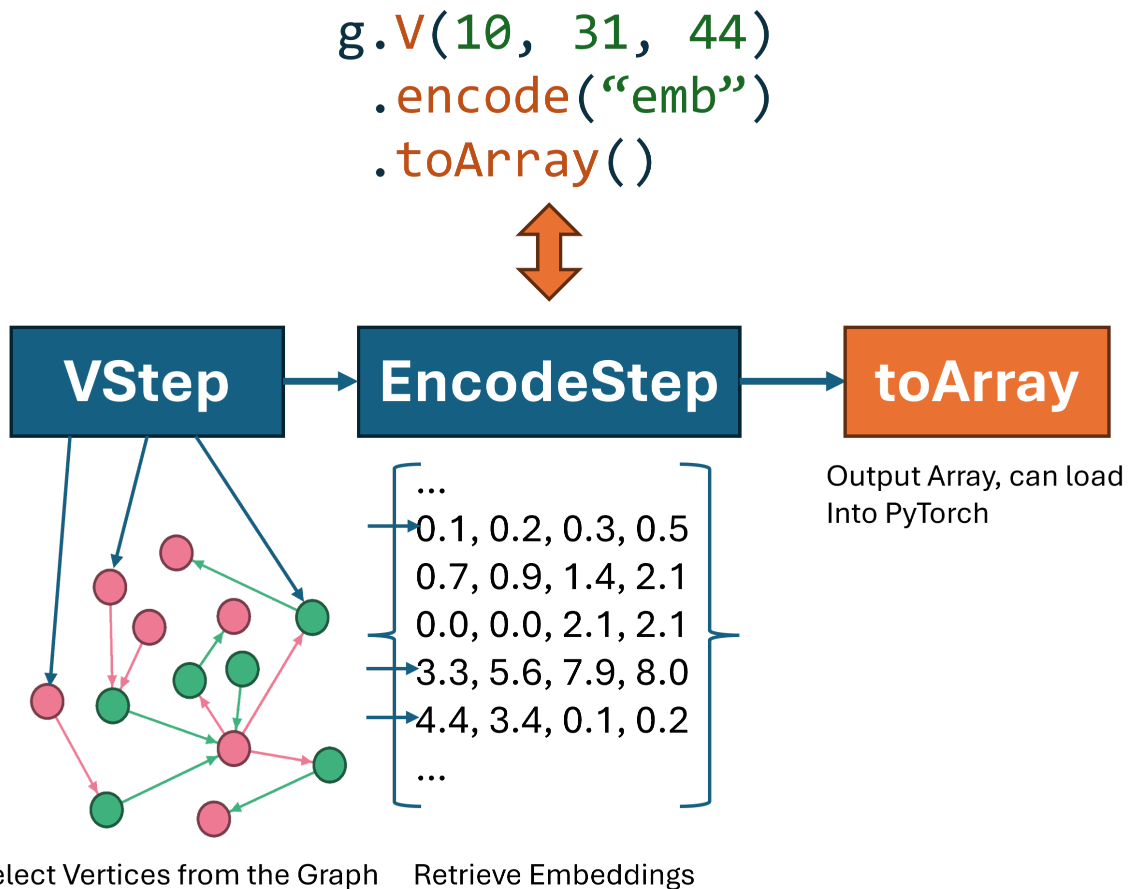


Figure 5.3: The Encode Step in Gremlin++. This step maps vertices or edges to their embeddings, which are stored as a contiguous tensor. Theoretically, implementing backends could store the embedding values in any format, so long as the final output is converted into a tensor. The encode step can be used in conjunction with a *toArray* finalizer to output the extracted embeddings in a format that can be loaded directly into a PyTorch tensor.

ding tensor value, similar to how the *Value Step* maps a vertex or edge to a property value. The *Embedding Step* [43] can add or update an existing embedding, similar to the *Property Step* for updating properties. I believe these two additions make Gremlin++ the first graph query language to explicitly support tensor-based values [23, 43].

Simply storing embeddings is not enough to make Gremlin++ useful for RAG. These embeddings still need to be searchable, so that a set of starting vertices can be generated from a

list of matched entities, ideally from within a single graph query. Enabling vector search from with a graph query also allows for something potentially even more useful - controlling the graph traversal based on vector similarity of encountered vertices and edges to the embedding of a user's prompt. This is the most important feature required for prize-aware graph traversal [43].

To allow vector search within a Gremlin++ query, I began by adding two new algorithms to Maelstrom. Like all algorithms in the Maelstrom API, these new algorithms were type-erased and compute-erased, allowing them to run on data stored on the CPU or GPU [28].

1. The *similarity* algorithm takes a set of source embeddings, and a set of target embeddings. For each source embedding vector, the algorithm calculates the similarities to the target vectors and takes the maximum similarity score, resulting in an output vector containing the maximum similarity for each source vector. Similarity scores are calculated using *cosine similarity*, L2 norm, or a similar algorithm.

2. The *topk* algorithm returns the indices of the top k elements in a given vector.

Using these algorithms, I then constructed the *Like Step* [43] in Gremlin++ (Figure 5.4), which performs fuzzy matching on embedding vectors. These steps can be used in the middle of a Gremlin++ query to control which vertices or edges are traversed based on similarity to the prompt embedding; in other words, this directly allows prize-aware graph traversal. As far as I know, no other graph engine currently supports this. The query in Figure 5.4 can be extended into a complete prize-aware RAG traversal by adding multiple hops, and its output is ready to be submitted to a DGR or CGR without any additional manipulation. No additional filtering is required; nor is the PCST required because the subgraph has already been pruned [35].

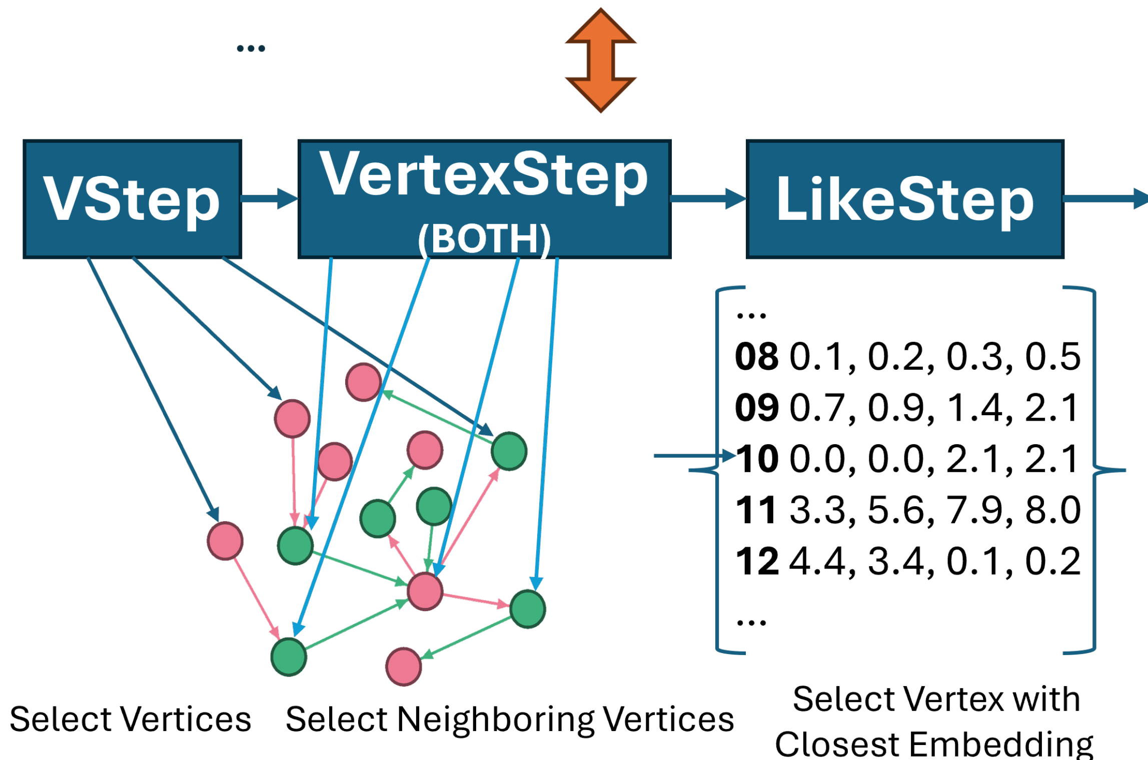


Figure 5.4: The Like Step in Gremlin++. This step compares embeddings against vertex or edge embeddings in the graph. It also has top-k functionality, allowing the user to request the vertices or edges with the *k* closest embeddings, as well as a limit that returns only matches with a similarity score above a certain score. Finally, it allows the user to choose between several similarity types, such as cosine similarity and L2 norm.

5.5 Incorporating Vector Indexing

5.5.1 Memory Usage and Behavior of Vector Search

While the achievement of prize-aware graph traversal is impressive, there is still one more major problem to be solved. Like most graph processing frameworks, the BitGraph framework was originally designed for sparse graph properties with a random access pattern [28]. Tensor-based embeddings have very different memory requirements; for one, they are much larger than sparse properties, and are usually accessed in large chunks [43]. For instance, the RAG query used by QRAG begins by selecting the closest starting vertices from the given set of target embeddings [43]. Without any index, this operation accesses the embedding of every single vertex in the graph.

On the CPU, this would take a prohibitively long time even for a single query. Using Gremlin++/BitGraph can help with this problem, since the GPU can search many embeddings at once, but there is another problem: for any reasonably-sized knowledge graph, the embedding tensor size likely exceeds the capacity of device (GPU) memory. The next logical step is to store the embedding tensor in either managed memory (UVM) or host-pinned memory (UVA) [28, 96]. As discussed earlier, this type of memory performs very well for sparse, randomly-accessed properties, but again, it performs very poorly for embedding tensors. Managed memory in particular is especially poor; so poor in fact that my attempts to benchmark it timed out. Accessing the embedding of every single vertex touched the entire graph and resulted in the maximum possible number of page faults. And furthermore, the managed memory system cached too much of the embedding data in managed memory, leaving too little memory for actually

training the LLM.

Using host-pinned memory instead of managed memory yields better results when executing the Like Step [43]. This held true not just for the initial vertex selection, but also during the other parts of the query that executed that step. When traversing QRAG's 4096-byte Roberta embeddings, and executing the Like Step in the middle of a traversal, using host-pinned memory cut the runtime from several minutes per Like Step to a fraction of a second per Like Step. However, there was not enough improvement when executing the initial Like Step to select the seed vertices. Seed selection still took an unreasonably long amount of time and was not usable. Even using host-pinned memory, and taking advantage of the full capabilities of the GPU, brute-force vector search was not practical. Therefore, a vector index was needed to enable the initial Like Step.

5.5.2 Incorporating FAISS into BitGraph

Nearly every vector database faces the same problem I encountered when working on QRAG [92, 97]. There is always a point at which the cost of a naive, brute-force search becomes too expensive, no matter how much compute is available [94, 95, 97]. This is why other vector databases support non-exhaustive vector search using an index. FAISS can work with these vector databases to accelerate vector search [92, 97], just as the BitGraph framework accelerates graph queries. Since I had essentially turned the BitGraph framework into a vector database, it made sense to integrate with something like FAISS to handle vector search. To keep Gremlin++ backend-agnostic, I integrated FAISS directly into the BitGraph library, adding an option for users to add an embedding index to the graph [43, 97]. Embedding indexes in BitGraph enable

accelerated vector search with a slight cost to accuracy, while also keeping the process of index management and querying transparent to the user. Users of BitGraph that want to use accelerated vector search do not need to be aware of FAISS; they only have to declare an embedding index using the BitGraph API [43].

Embedding indexes in BitGraph use FAISS's GPU IVFPQ index [93, 94], which was discussed earlier in this chapter. This type of index keeps the index size small, allowing it to be stored completely in device memory, thus taking full advantage of the GPU. This dramatically cuts the time needed to run the initial like step, resulting in 840x speedup over the non-indexed query (Figure 5.5).

5.6 Assembling QRAG

5.6.1 The QRAG Process

With the additions to the BitGraph complete, and queries running in a reasonable time, the complete version of QRAG (Figure 5.6) could now be assembled. QRAG begins by performing named entity recognition (NER) [77] to extract named entities from the prompt using a pretrained BERT model [98]. Using a fairly complex model for NER ensured that the resulting entities were accurate, and that the correct seed vertices could be selected in the next step [43]. After NER, a Roberta [99, 100] tokenizer generated embeddings for the initial prompt and named entities, so they could be used by the RAG query. The RAG query, written Gremlin++ and executed against the knowledge graph stored in BitGraph, starts by selecting seed vertices, and then performs a prize-aware traversal of the local neighborhood. The seed vertex selection process uses a pretrained BitGraph embedding index, which is backed by a FAISS IVFPQ index. FAISS is

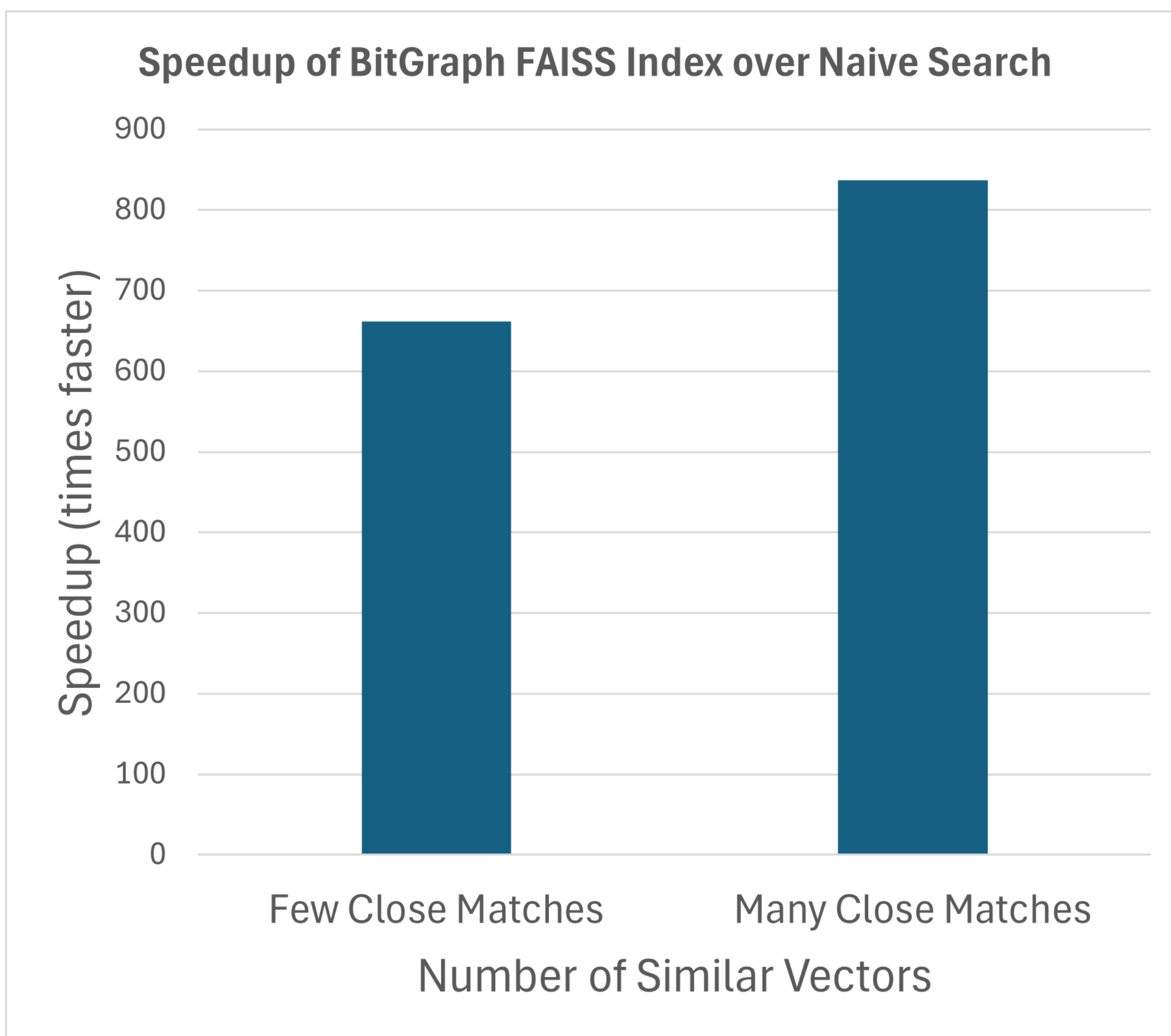


Figure 5.5: Speedup over the non-indexed seed selection step delivered by BitGraph's IVFPQ FAISS indexes. Two versions of this query were tested - one where many embeddings were close to the target embedding, and one where only a few embeddings were close to the target embedding (the more common use case). Speedup on both use cases was excellent, reaching nearly 700x when there were few close matches, and over 800x when there were many close matches. Adoption of the IVFPQ FAISS indexes was critical in making QRAG possible. Without them, it would have taken far more GPU-hours to train QRAG then were available.

completely hidden behind the BitGraph API. The final output of the Gremlin++ query is either an output subgraph and Roberta embeddings (for CGR), or a list of vertices corresponding to documents of interest (for DGR). This output is finally passed to either a pure LLM or GNN+LLM model, which returns an answer to the prompt [43].

5.6.2 Query Tuning

The Gremlin++ queries used in QRAG have tunable parameters, which can be modified as needed (Figure 5.8) when training the final model. The 2WikiMultihopQA [83] dataset is especially good for query tuning because it contains both prompt answers and supporting facts within its ground truth data. This was one of the deciding factors in selecting this dataset over other competing datasets like HotpotQA. To tune the parameters of the Gremlin++ queries, I executed a random grid search over several different parameter values and compared the output vertex embeddings to the output vertex embeddings of the ground truth supporting facts [43]. I then selected the parameters that resulted in the highest average similarity scores across the whole ground truth dataset, and kept these parameters fixed when training the LLM and GNN+LLM models [43].

5.7 Performance Analysis

5.7.1 The Knowledge Graph

To measure the performance and accuracy of QRAG, I constructed a knowledge graph in BitGraph using the 2WikiMultihopQA dataset [83], from the publicly-available version of the dataset on GitHub [101]. I assigned a sequential vertex id to each article and paragraph in the

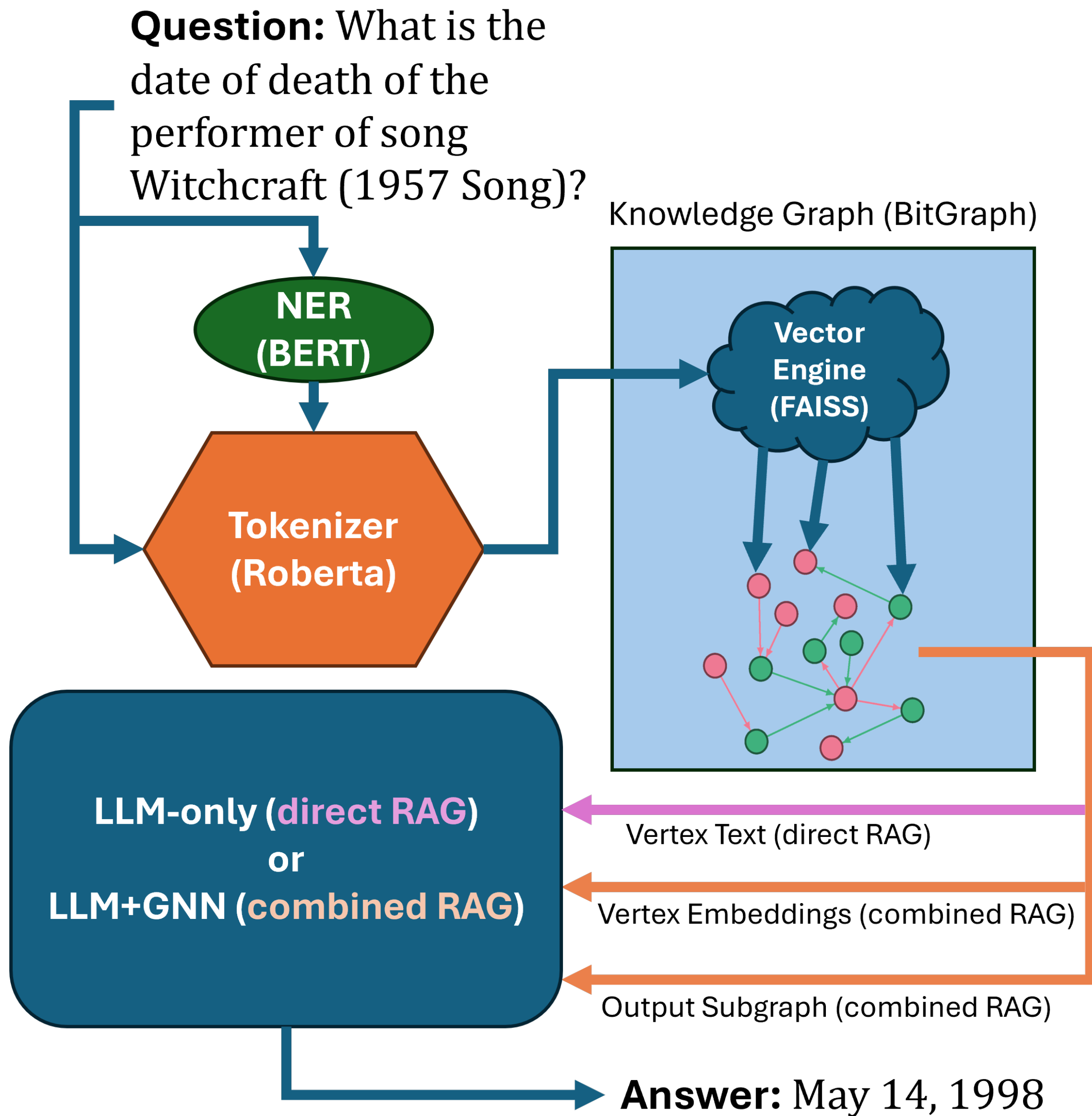


Figure 5.6: End-to-end QRAG process. A pretrained Roberta (Sentence-Bert) [99, 100] tokenizer produces output embeddings from the user prompt, and named entities extracted from the user prompt using a pretrained BERT [98] model. These embeddings are used to select initial vertices using a FAISS IVFPQ index, which BitGraph hides behind its API, allowing users to simply call the Gremlin++ Like Step so long as they have instructed BitGraph to index for the target embedding [43]. Then, a prize-aware RAG query traverses the graph to produce a final output subgraph containing vertices corresponding to documents of interest. QRAG supports both direct-input (direct) and combined RAG [43]. In the direct-input case, the text from the extracted vertices is joined to the original prompt and fed directly into the LLM. In the combined case, the whole subgraph and embeddings of the extracted vertices are concatenated with the tokenized prompt and fed into the combined GNN+LLM model.

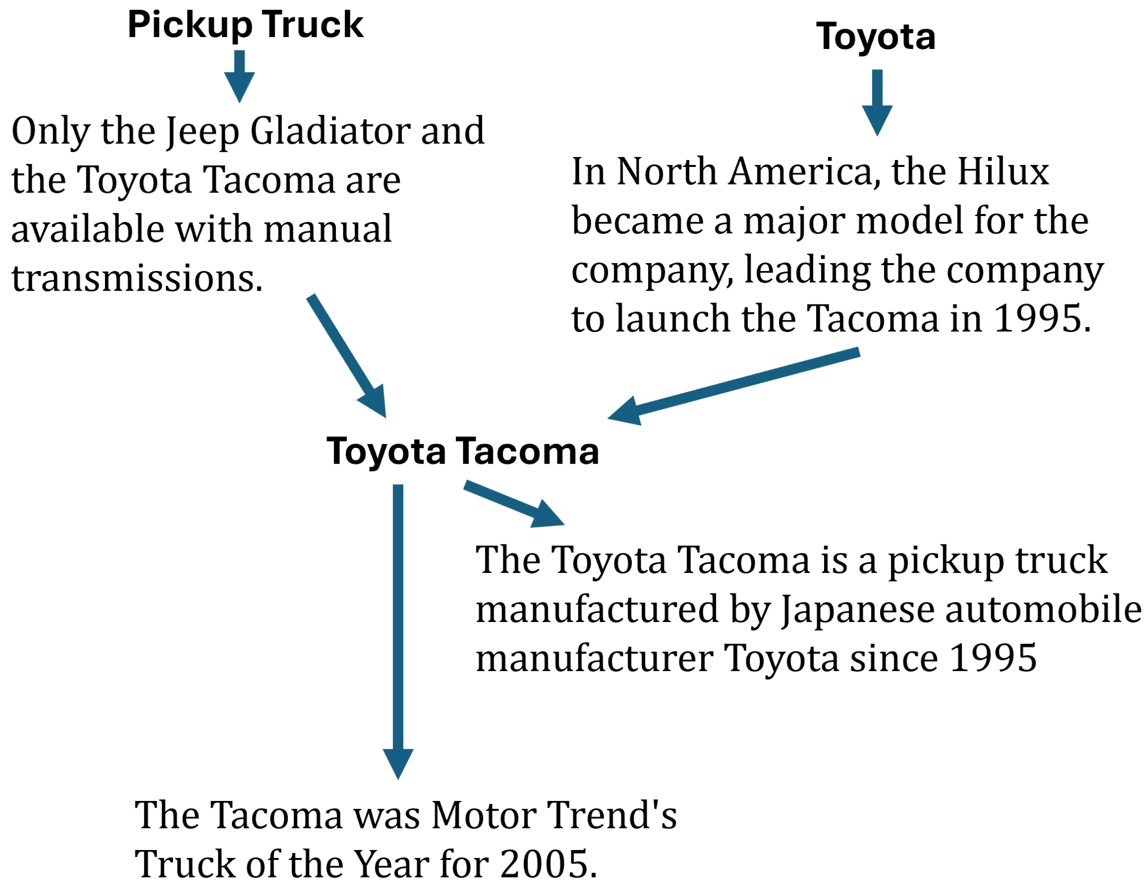


Figure 5.7: Sample knowledge graph resembling the knowledge graph constructed from 2Wiki-MultihopQA [83]. The Wikipedia article on the *Toyota Tacoma* has outgoing edges to paragraphs in the article and incoming edges from paragraphs in other articles that link to the Tacoma article [102, 103, 104].

dataset, added ownership edges from each article to its contained paragraphs, and added linkage edges from paragraphs to the articles they linked to. Figure 5.7 illustrates the structure of this knowledge graph.

5.7.2 Model and Training Pipeline Structure

To give the most complete picture of QRAG's performance, I constructed a training pipeline comprised of five stages, and trained four different models, as shown in Figures 5.8 and 5.9 [43].

RAG Type	*LLM Type*	*Textual Graph*	*System (CPU + GPU)*
None	Prompt Only	No	AMD EPYC 9374F + H100 NVL 94GB
DGR	Prompt with Context	No	AMD EPYC 9374F + H100 NVL 94GB
CGR	Prompt Only	No	AMD EPYC 9374F + H100 NVL 94GB
CGR	Prompt Only	Yes	AMD EPYC 9374F + H100 NVL 94GB

Table 5.1: QRAG Benchmark Models

Table 5.1 lists the four models, which were comprised of a baseline LLM-only model, LLM with direct-input RAG, a LLM with combined RAG, and a LLM with combined RAG and additional textual graph input (as described in the original G-Retriever paper [35]). I used the PyTorch Geometric (PyG) [70] version of G-Retriever to handle all four models. The LLM used across all models was *OpenLlama-3B-V2* [37, 40]. The GNN used to produce graph embeddings for the combined RAGs was the PyG implementation of GAT [86].

The training pipeline started with *embedding generation* [43], the process of producing Roberta embeddings for every sentence and article stored in the knowledge graph and saving them to disk, so they could be reused by other pipeline stages. Next was *retrieval tuning* [43], the query tuning process discussed earlier. A random grid search was used to select the parameters of the prize-aware graph traversal. These parameters were used for both the DGR and CGR versions of QRAG. The third stage took the pretrained LLM and finetuned it using ground truth data from 2WikiMultihopQA, following the same process used by G-Retriever [35]. Two versions of the LLM were finetuned: one for DGR, which incorporated the context, and another for CGR and the non-RAG model, which did not include textual context. I used LoRA [38] for this finetuning process to freeze the model weights and optimize the rank decomposition matrices of the transformer model, again following the process used in G-Retriever [35]. These initial stages are shown in in Figure 5.8.

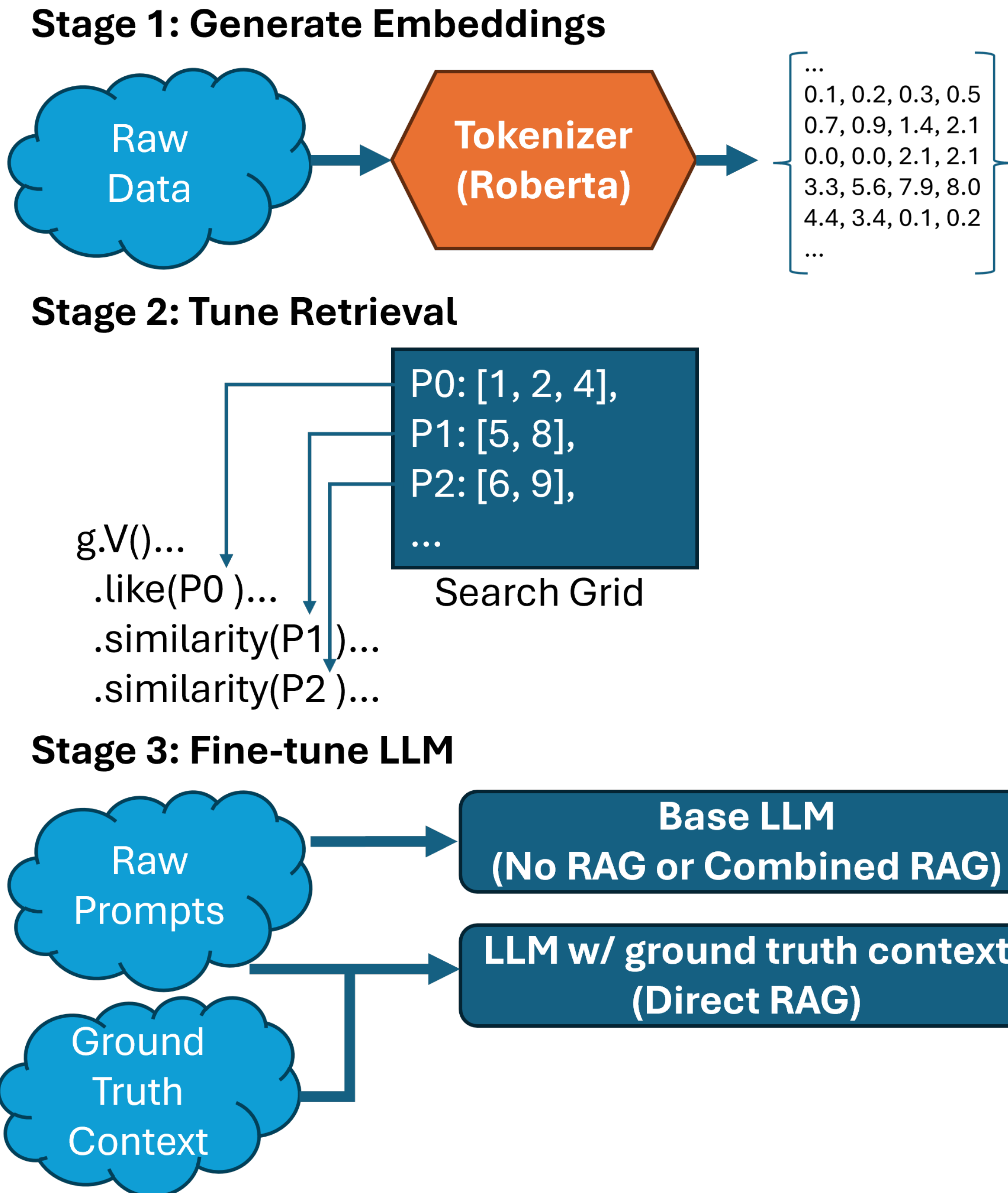


Figure 5.8: Stages 1-3 of the QRAG training pipeline. The first stage, embedding generation, produces embeddings for the entities stored in the knowledge graph. The second stage, retrieval tuning, tunes the Gremlin++ query used for prize-aware graph traversal for both the CGR and DGR versions of QRAG. The third stage, LLM fine-tuning, fine tunes the LLM so it is better-adjusted to the problem and dataset being presented to it during this benchmark. During the fine-tuning process, the model weights are frozen and the rank decomposition matrices are optimized using LoRA [38].

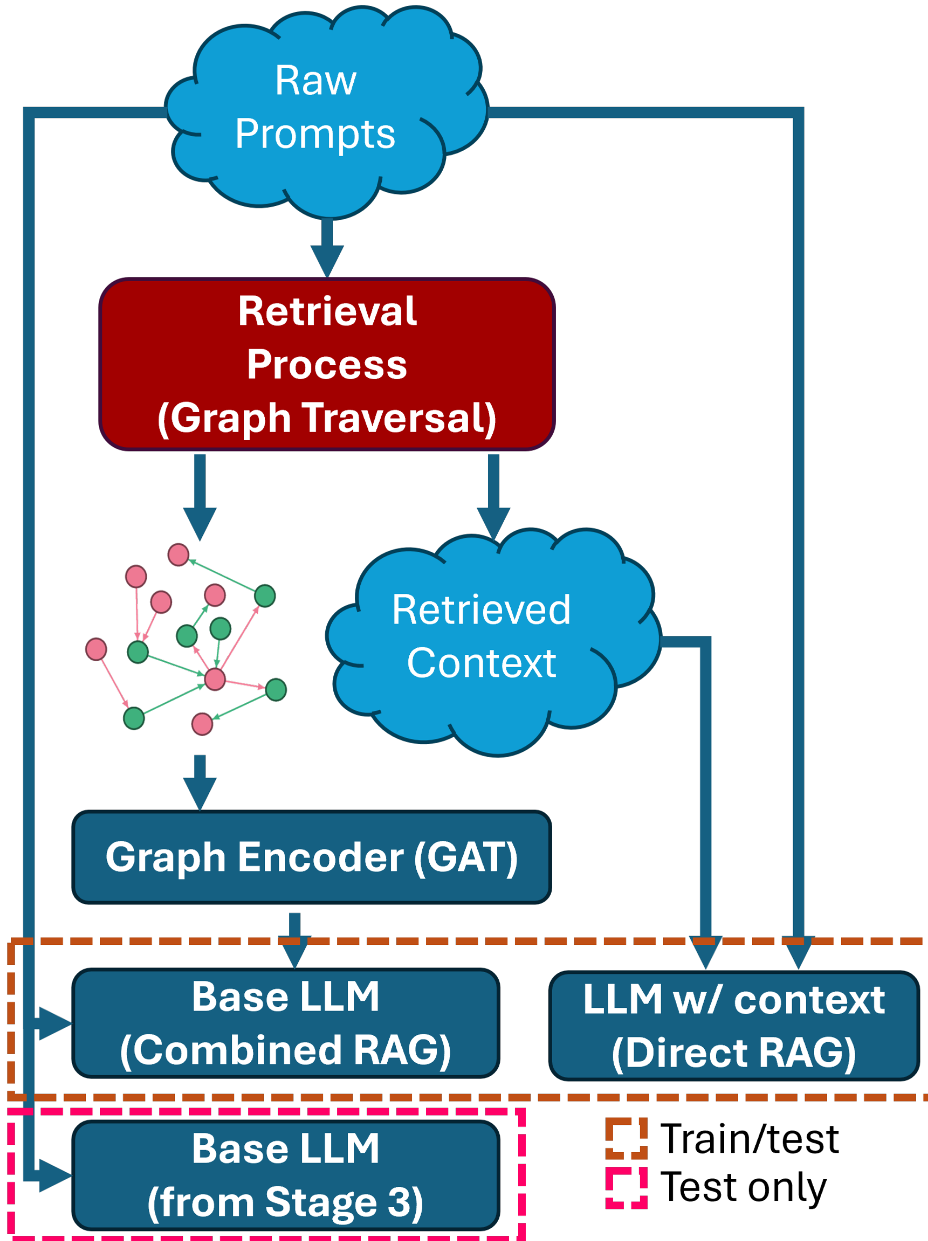


Figure 5.9: The fourth and fifth stages of the QRAG training pipeline. The fourth stage, *model training* [43], takes each fine-tuned LLM and trains it using ground truth data from the 2Wiki-MultihopQA dataset, using backpropagation to update the model weights. Models that used the same fine-tuned LLM had their own copy of the LLM so their training did not interfere with the training of other models. All models except the baseline non-RAG model used the Gremlin++ query to extract data from the knowledge graph. The DGR model used only text from the vertices in the output subgraph, while the CGR model used an encoding of the graph structure generated by GAT. The CGR models trained the GNN and LLM together using PyG. The fifth stage, *model evaluation* [43], evaluated the final model on unseen data to produce the mean embedding difference score, which was used to rate model performance.

The final two stages train and evaluate (test) the complete model [43]. For the CGR version, the training stage added the GNN portion to the previously-tuned LLM, and trained the two models together using PyG. Note that there were two versions of the CGR trained, one with the textual graph added to the prompt, and one without the textual graph. For the DGR and non-RAG models, their corresponding fine-tuned LLM from the previous stage was the only model to be trained. Training, as usual, consists of passing in each batch consisting of raw input to the model, comparing the output to the target variable, and backpropagating to update the model weights. In the testing (evaluation) stage, the model is run against unseen data to calculate the mean embedding difference from the expected answer embeddings. This mean is the final performance score of the model.

5.7.3 Benchmark Results

Figure 5.10 shows the mean difference metric for each of the four versions of QRAG that were evaluated [43]. Training took about 8 hours per model. Both the direct-input RAG and no-RAG models outperformed the combined GNN+LLM model based on G-Retriever [43]. Adding the textual graph to the CGR model improved its performance, but it still lagged behind the DGR and no-RAG models. This result was consistent with my original expectations; I was not surprised to see the DGR model perform the best. The DGR model combined the best of both worlds, using the prize-aware graph traversal to augment the proven process of direct-input, which is simpler and faster to train than combined models. I believe the structure of the knowledge graph used for the benchmarks also strongly influenced the results. QRAG used a much larger knowledge graph encompassing the whole corpus, as opposed to G-Retriever, which used

relatively small, disjoint input graphs. I went this way because I believe the knowledge graph I used is much more like a real-world enterprise graph that an actual deployed RAG will have to interact with. It is much hierarchical, and has lower connectivity than the graphs used by G-Retriever, which I believe made the structure information added by the GNN less relevant. It's possible, however, that a tool like LangChain [105], which can produce more edges between documents, could be used to augment my knowledge graph and make the structural information more relevant, thus improving the performance of CGR.

5.7.4 Opportunities for Improvement

While the performance of DGR QRAG was good, there is still significant room for improvement. There were several issues affecting the retrieval process that need to be resolved before this system could be used in production. First, the retrieval query is not always accurate [43]. This comes from error in the ANNS that was used to select the seed vertices from the retrieved entities. On one hand, ANNS is what made QRAG feasible; but on the other hand, it is limiting the performance of QRAG by introducing noticeable retrieval error. Further tuning of BitGraph's embedding indexes could help resolve this issue. It might also be worthwhile to investigate alternatives to FAISS.

Another issue hindering DGR performance is that I used a less aggressive query so that the same query would also work for CGR. Even the 2WikiMultihopQA dataset introduces some noise with additional context beyond what is actually needed to answer the question [83]. Well-trained LLMs are definitely capable of sifting through a list of facts and determining which ones are relevant, so long as the list of facts is presented as textual input. In a future experiment, I

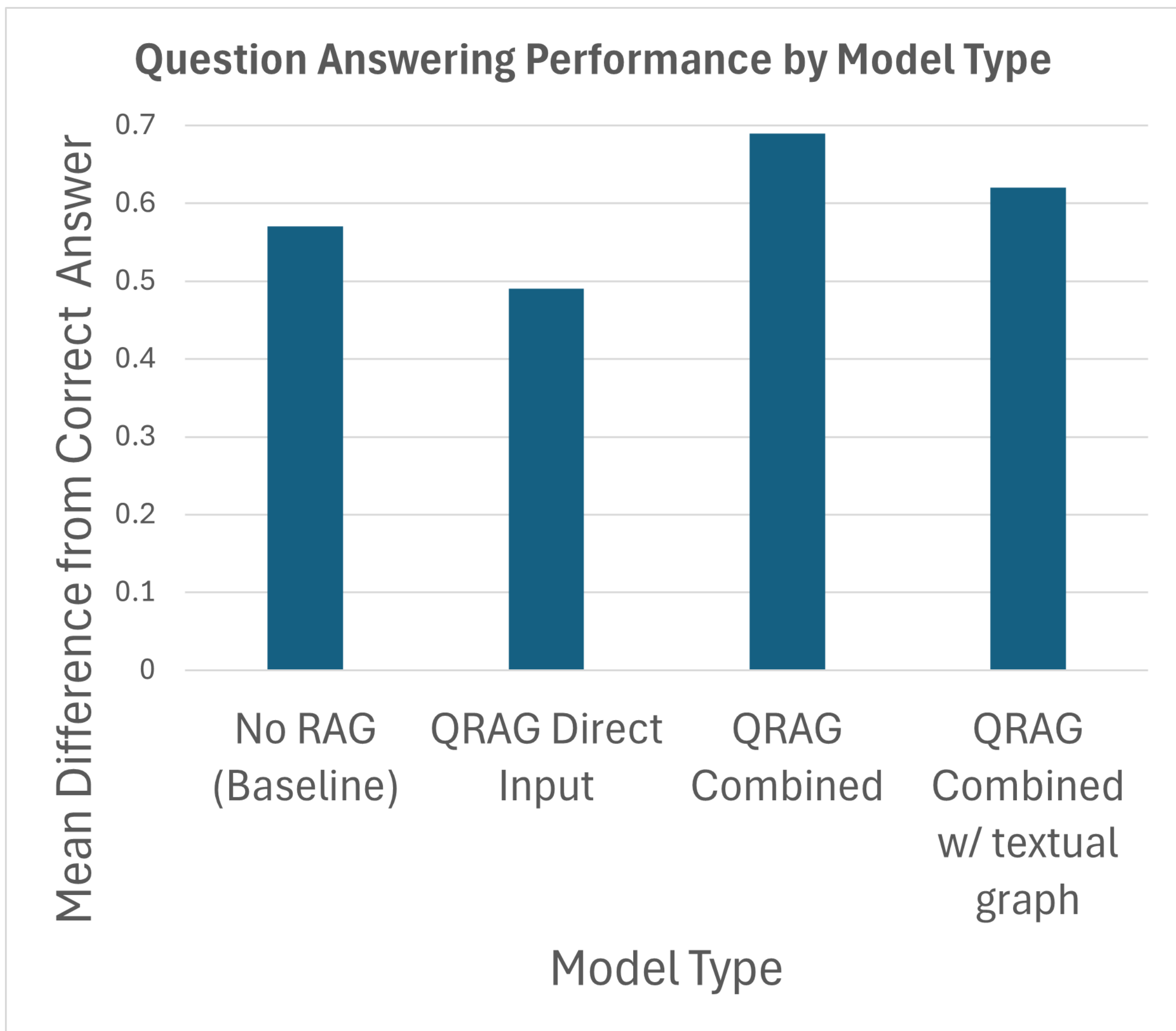


Figure 5.10: Mean embedding difference of each model. Lower is better. The direct-input version of QRAG performed the best, followed by the baseline LLM without any RAG. The two combined RAG models did not perform well, largely due to the structure of the knowledge graph.

might try re-running the pipeline for DGR with an aggressive query that grabs more vertices, erring on the side of too much rather than too little information, and trusting the LLM to be a good judge of relevance.

5.8 Discussion

In this chapter, I discussed the extension of the BitGraph framework the problem of retrieval augmented generation (RAG). I explained the key concepts of DGR and CGR, and explained how G-Retriever implements a graph-based CGR. I discussed the concept of prize-aware graph traversal, and outlined the tools needed to make it possible, following up with an explanation of how I incorporated each of these tools into the BitGraph framework. I then examined the memory usage and access patterns of vector search, justified the need for integration with a vector search engine, and explained how I made this integration transparent to the user. Finally, I outlined the process of constructing and training the QRAG model, analyzed the performance of four different versions of the QRAG model, and outlined how this performance could be improved through future work.

Chapter 6: Conclusion and Future Work

6.1 Conclusion

In this dissertation I described the BitGraph framework, the result of a complex software and systems engineering project aimed at accelerating graph analytics. I began by introducing the framework and describing its history, from my original idea from 2018, to its current composition of three separate libraries, each with an important purpose. The first component of my dissertation discussed the structure and performance of these three libraries. The lowest-level library, Maelstrom, accelerates basic vector and matrix operations, provides basic data structures like arrays, sparse matrices, and hash tables, and supports type-erased and compute-agnostic operations through a single API. Gremlin++, a higher-level library, is a high-performance OLAP graph query language based on Gremlin that translates query steps into a series of primitive operations in the Maelstrom API, while leaving some details to implementing graph backends. The highest-level API, BitGraph, is such a backend, implementing the Gremlin++ API through use of Maelstrom data structures and algorithms. I showed with benchmarks that the BitGraph framework can achieve up to 35x speedup over the fastest publicly-available CPU implementation of a Gremlin graph backend, and finally, discussed where there was room for improvement, emphasizing the need for better query optimization.

In the second component of this dissertation, I focused on query optimization, and the

traversal strategies used by Gremlin++ for just-in-time optimizations. I described the fundamentals of query optimization, and how my work implemented the fundamental types of optimizations: *fusion*, *reordering*, *loop unrolling*, *pattern detection*, and *backend-specific optimizations*. I broke down each optimization's key contribution to query speedup, as well as additional speedup achieved by multiple optimizations working together, illustrating how a sample query ran 100% faster after all optimizations were applied. Finally, I examined a specific use case, retrieval augmented generation (RAG) and showed how a just-in-time optimizations accelerated a RAG query by up to 40%.

In the third and final component of this dissertation, I described QRAG, an accelerated graph-based RAG built using the BitGraph framework. I emphasized how QRAG was different from other types of graph-based RAG because it implemented *prize-aware graph traversal*, which allowed for more intelligent subgraph extraction. I then discussed how QRAG required two additional features to be added to the BitGraph framework to support prize aware graph traversal: *native embedding support* and *vector indexing*. I explained the need for these features by highlighting the differences between traditional vertex and edge properties and embeddings, and analyzing the memory properties of embeddings that made these two features critical. After a short discussion on how these two features were integrated, I then described the full QRAG pipeline, comprised of *embedding generation*, *retrieval tuning*, *LLM fine-tuning*, *model training*, and *model evaluation*, and analyzed the accuracy of four different versions of the pipeline, including a no-RAG model, direct-input RAG, combined RAG with textual graph, and combined RAG without textual graph. Of these, the direct-input RAG performed the best, delivering answers that were on average 14% closer to the ground-truth set of answers. Finally, I analyzed how to achieve even higher accuracy improvement, highlighting the need to test a broader array

of ANNS algorithms and potentially expand the graph traversal to output larger subgraphs.

Through Python bindings included with the Gremlin++ and BitGraph libraries, as well as scripts and documentation in the official BitGraph repository [106], anyone can use or adopt my work for their own purpose, as well as reproduce the results I have described in this dissertation and related papers [28, 31, 43, 106].

6.2 Future Work

6.2.1 Beyond PCI-e

Throughout this dissertation, every benchmark I have described was performed on a system where the CPU and GPU were connected via a PCI-e bus, and there was a clear distinction between device memory and host memory. This is no longer the case on many systems, such as Grace Hopper and Grace Blackwell devices, which have an ARM-based CPU and NVIDIA GPU connected together through a chip-to-chip interconnect, and a single shared memory. On these systems, memory allocated with *malloc* can be accessed by the GPU directly, as opposed to PCI-e systems, which require whatever is in that allocated memory to be copied to device memory before it can be read by the GPU.

One of the problems limiting QRAG was the difficulty of maintaining a large vector index on the device, necessitating the use of an index with high compression (IVFPQ). Having direct access to a large unified memory space, such as in the system described in [107], makes a larger index with less compression (and therefore more accurate vector retrieval) feasible. Furthermore, a lot of applications that frequently use host-pinned memory (again, QRAG is a good example here) would be accelerated through direct memory access. It would be interesting to see just how

much faster QRAG and other BitGraph-based applications are on such a system.

6.2.2 Alternatives to FAISS

I chose FAISS because it was very well-documented, it had a good C++ API, and it supported GPU acceleration. However, it is not the only vector search backend. *Milvus* [108] is another well-known, well-documented database that supports a C++ API and GPU acceleration, and *cuVS* [109] is yet another. These libraries offer a variety of different search algorithms and index implementations, and it would be valuable to see if anything they offer can beat what BitGraph currently uses. There are also so many different types of vector search workloads, that it is feasible one library or index type may make more sense for a particular application. Allowing more choice would make the BitGraph framework more attractive for use in RAG.

6.2.3 Multi-GPU Processing

Multi-GPU processing is something that I considered when working on all three components of my dissertation. The obvious motivation for running on multiple GPUs is scalability. At some point a single GPU is not enough; one GPU only has so much memory, after all, and datasets continue to grow. I have already made some progress in adding support for multi-GPU algorithms and data structures to Maelstrom, which is the first step in enabling multi-gpu execution throughout the framework. Once all the required algorithms have multi-GPU support, the next step is to modify the BitGraph library to add bookkeeping for distributed graphs. This in turn may require changes to Maelstrom's hash table and sparse matrix data structures.

6.2.4 Other Query Languages

Gremlin is not the only graph query language out there. As discussed in this dissertation, there are many competing graph query languages, and attempts at standardization have not been successful. Gremlin is arguably the easiest to parallelize, but despite this, I still had to adapt the original version of Gremlin into Gremlin++, which has its own semantics and paradigms not present in its ancestor. Why not, then, adapt the semantics of other languages, like Cypher? One answer is that Cypher's syntax focuses more on pattern matching, which is harder to decompose into a series of vector operations, but there are some ways to overcome this. Figuring out how Cypher, GQL, or another language can be translated into Maelstrom API calls would be an interesting research problem that would help further serve the needs of the graph analytics community.

Bibliography

[1] Ananth Kalyanaraman and Partha Pratim Pande. "A brief survey of algorithms, architectures, and challenges toward extreme-scale graph analytics." In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1307–1312.

[2] Tyler Procko. "Graph retrieval-augmented generation for large language models: A survey." In: *Available at SSRN* (2024).

[3] Boci Peng et al. "Graph retrieval-augmented generation: A survey." In: *arXiv preprint arXiv:2408.08921* (2024).

[4] James R Groff, Paul N Weinberg, and Andrew J Oppel. *SQL: the complete reference*. Vol. 2. McGraw-Hill/Osborne, 2002.

[5] Jean-Francois Boulicaut and Cyrille Masson. "Data mining query languages." In: *Data Mining and Knowledge Discovery Handbook* (2010), pp. 655–664.

[6] Sabri Pllana et al. "A survey of the state of the art in data mining and integration query languages." In: *2011 14th International Conference on Network-Based Information Systems*. IEEE. 2011, pp. 341–348.

[7] Hendrik Blockeel et al. "A practical comparative study of data mining query languages." In: *Inductive databases and constraint-based data mining* (2010), pp. 59–77.

[8] Michael Armbrust et al. "Spark sql: Relational data processing in spark." In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1383–1394.

[9] Nadime Francis et al. "Cypher: An Evolving Query Language for Property Graphs." In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, 1433–1445. ISBN: 9781450347037. DOI: 10.1145/3183713.3190657. URL: https://doi.org/10.1145/3183713.3190657.

[10] M.A. Rodriguez. "The Gremlin Graph Traversal Machine and Language." In: *Proceedings of the 15th Symposium on Database Programming Languages*. Symposium on Database Programming Languages. Pittsburgh, PA, USA, 2015, pp. 1–10. DOI: 10.1145/2815072.2815073. URL: https://doi.org/10.1145/2815072.2815073.

[11] TigerGraph. *GSQL: Graph Query Language*. 2024. URL: https://www.tigergraph.com/gsql/ (visited on 07/06/2024).

[12] Keith W. Hare. *ISO/IEC 39075 Database Language GQL*. 2024. URL: https://jtc1info.org/wp-content/uploads/2024/04/2024-Article-39075-Database-Language-GQL.docx.pdf (visited on 07/06/2024).

[13] W3C. *SPARQL 1.1 Query Language W3C Recommendation 21 March 2013*. 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/ (visited on 09/10/2023).

[14] Thomas H. Cormen et al. *Introduction to Algorithms, Fourth Edition*. Cambridge, Massachusetts, United States of America: The MIT Press, 2022. ISBN: 9780262046305.

[15] Apache TinkerPop. *TinkerPop Documentation*. 2023. URL: https://tinkerpop.apache.org/docs/3.7.0/reference/ (visited on 02/17/2025).

[16] Jyothish Soman, Kothapalli Kishore, and P J Narayanan. "A fast GPU algorithm for graph connectivity." In: *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 2010, pp. 1–8. DOI: 10.1109/IPDPSW.2010.5470817.

[17] Sergey Brin and Lawrence Page. "The anatomy of a large-scale hypertextual web search engine." In: *Computer networks and ISDN systems* 30.1-7 (1998), pp. 107–117.

[18] Linton C. Freeman. "A Set of Measures of Centrality Based on Betweenness." In: *Sociometry* 40.1 (1977), pp. 35–41. ISSN: 00380431, 23257938. URL: http://www.jstor.org/stable/3033543 (visited on 02/17/2025).

[19] Leo Katz. "A new status index derived from sociometric analysis." In: *Psychometrika* 18.1 (1953), pp. 39–43.

[20] Will Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs." In: *Advances in neural information processing systems* 30 (2017).

[21] Maciej Besta et al. "Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries." In: *ACM Computing Surveys* 56.2 (2023), pp. 1–40.

[22] Deepak Singh Rawat and Navneet Kumar Kashyap. "Graph database: a complete GDBMS survey." In: *Int. J* 3 (2017), pp. 217–226.

[23] Renzo Angles et al. "Foundations of Modern Query Languages for Graph Databases." In: *ACM Comput. Surv.* 50.5 (2017). ISSN: 0360-0300. DOI: 10.1145/3104031. URL: https://doi.org/10.1145/3104031.

[24] Neo4j Inc. *Neo4j Graph Database & Analytics*. 2025. URL: https://neo4j.com/ (visited on 02/17/2025).

[25] TigerGraph. *TigerGraph: Drive Competitive Advantage with the Enterprise-Scale Graph Data Platform for Advanced Analytics and Machine Learning*. 2024. URL: https://tigergraph.com (visited on 07/06/2024).

[26] Amazon Web Services. *Amazon Neptune*. 2025. URL: https://aws.amazon.com/neptune/ (visited on 02/17/2025).

[27] Paul Hudak. "Conception, evolution, and application of functional programming languages." In: *ACM Computing Surveys (CSUR)* 21.3 (1989), pp. 359–411.

[28] Alexandria Barghi. "BitGraph: A Framework For Scaling Temporal Graph Queries on GPUs." In: *Temporal Graph Learning Workshop @ NeurIPS 2023*. 2023. URL: https://openreview.net/forum?id=B7Wd1K0l4I.

[29] Mohamed Nadjib Mami et al. “The query translation landscape: a survey.” In: *arXiv preprint arXiv:1910.03118* (2019).

[30] Harsh Thakkar et al. “Let’s build Bridges, not Walls: SPARQL Querying of TinkerPop Graph Databases with Sparql-Gremlin.” In: *2020 IEEE 14th International Conference on Semantic Computing (ICSC)*. 2020, pp. 408–415. DOI: 10.1109/ICSC.2020.00080.

[31] Alexandria Barghi. “Accelerating Graph Query Languages for Machine Learning and Retrieval Augmented Generation.” In: *2024 IEEE International Conference on Big Data (BigData)*. 2024, pp. 3367–3374. DOI: 10.1109/BigData62323.2024.10825641.

[32] Todd Hricik, David Bader, and Oded Green. “Using RAPIDS AI to accelerate graph data science workflows.” In: *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2020, pp. 1–4.

[33] NVIDIA Corporation. *RAPIDS Graph Documentation*. 2025. URL: https://docs.rapids.ai/api/cugraph/stable/ (visited on 02/17/2025).

[34] Yangzihao Wang et al. “Gunrock: GPU Graph Analytics.” In: *ACM Transactions on Parallel Computing* 4.1 (Aug. 2017), 3:1–3:49. DOI: 10.1145/3108140. URL: http://escholarship.org/uc/item/9gj6r1dj.

[35] Xiaoxin He et al. “G-retriever: Retrieval-augmented generation for textual graph understanding and question answering.” In: *arXiv preprint arXiv:2402.07630* (2024).

[36] Shervin Minaee et al. “Large language models: A survey.” In: *arXiv preprint arXiv:2402.06196* (2024).

[37] Hugo Touvron et al. “Llama: Open and efficient foundation language models.” In: *arXiv preprint arXiv:2302.13971* (2023).

[38] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL]. URL: https://arxiv.org/abs/2106.09685.

[39] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL]. URL: https://arxiv.org/abs/2303.08774.

[40] Xinyang Geng and Hao Liu. *OpenLLaMA: An Open Reproduction of LLaMA*. May 2023. URL: https://github.com/openlm-research/open_llama.

[41] Penghao Zhao et al. *Retrieval-Augmented Generation for AI-Generated Content: A Survey*. 2024. arXiv: 2402.19473 [cs.CV]. URL: https://arxiv.org/abs/2402.19473.

[42] Yijun Tian et al. *Graph Neural Prompting with Large Language Models*. 2023. arXiv: 2309.15427 [cs.CL]. URL: https://arxiv.org/abs/2309.15427.

[43] Alexandria Barghi. “QRAG: Using Learnable Graph Queries for Retrieval Augmented Generation.” In: *2025 IEEE 15th Annual Computing and Communication Workshop and Conference (CCWC)*. 2025, pp. 00561–00568. DOI: 10.1109/CCWC62904.2025.10903812.

[44] Federico Busato et al. “Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs.” In: Sept. 2018. DOI: 10.1109/HPEC.2018.8547541.

[45] Oded Green and David Bader. "cuSTINGER: Supporting dynamic graph algorithms for GPUs." In: Sept. 2016. DOI: 10.1109/HPEC.2016.7761622.

[46] Blazegraph. *Blazegraph Database*. 2016. URL: https://blazegraph.com (visited on 02/17/2025).

[47] Carnegie Mellon Database Group. *Database of Databases - BlazeGraph*. 2025. URL: https://dbdb.io/db/blazegraph (visited on 02/17/2025).

[48] Alexandria Barghi. "Gremlin++ & BitGraph: Implementing The Gremlin Traversal Language and a GPU-Accelerated Graph Computing Framework in C++." MA thesis. ProQuest, Ann Arbor, MI: University of Maryland, 2019.

[49] Salman Salloum et al. "Big data analytics on Apache Spark." In: *International Journal of Data Science and Analytics* 1 (2016), pp. 145–164.

[50] PyTorch Foundation. *PyTorch*. 2025. URL: https://pytorch.org/ (visited on 02/17/2025).

[51] Alexander Ocsa. "SQL for GPU Data Frames in RAPIDS Accelerating end-to-end data science workflows using GPUs." In: *LatinX in AI Research at ICML 2019*. 2019.

[52] Randy Gelhausen. *Announcing the General Availability of Dask-SQL on GPUs*. 2022. URL: https://medium.com/rapids-ai/announcing-the-general-availability-of-dask-sql-on-gpus-5abe5312c5d5 (visited on 10/06/2022).

[53] Yuhao Zhang and Arun Kumar. "Lotan: Bridging the Gap between GNNs and Scalable Graph Analytics Engines." In: *Proceedings of the VLDB Endowment* 16.11 (2023), pp. 2728–2741.

[54] NVIDIA Corporation. *Thrust: The C++ Parallel Algorithms Library*. 2025. URL: https://nvidia.github.io/cccl/thrust/ (visited on 02/18/2025).

[55] Alexandria Barghi. *Maelstrom: High-performance type-erased structures and algorithms*. 2024. URL: https://github.com/bgamer50/maelstrom (visited on 02/19/2025).

[56] Mark Harris. *Unified Memory for CUDA Beginners*. 2017. URL: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/ (visited on 09/10/2023).

[57] Yunsong Wang et al. *cuCollections*. 2023. URL: https://github.com/NVIDIA/cuCollections (visited on 09/24/2023).

[58] NVIDIA Corporation. *cuSPARSE*. 2023. URL: https://docs.nvidia.com/cuda/cusparse/index.html (visited on 09/24/2023).

[59] G Satyanarayana Reddy et al. "Data Warehousing, Data Mining, OLAP and OLTP Technologies are essential elements to support decision-making process in industries." In: *International Journal on Computer Science and Engineering* 2.9 (2010), pp. 2865–2873.

[60] Jure Leskovec and Julian Mcauley. "Learning to discover social circles in ego networks." In: *Advances in neural information processing systems* 25 (2012).

[61] Lars Backstrom et al. "Group formation in large social networks: membership, growth, and evolution." In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2006, pp. 44–54.

[62] Jure Leskovec et al. “Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters.” In: *Internet Mathematics* 6.1 (2009), pp. 29–123.

[63] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[64] M. Strohmeier et al. “Crowdsourced air traffic data from the OpenSky Network 2019–2020.” In: *Earth System Science Data* 13.2 (2021), pp. 357–366. DOI: 10.5194/essd-13-357-2021. URL: https://essd.copernicus.org/articles/13/357/2021/.

[65] Shenyang Huang et al. *Temporal Graph Benchmark for Machine Learning on Temporal Graphs*. 2023. arXiv: 2307.01026 [cs.LG].

[66] Li Su et al. “Banyan: a scoped dataflow engine for graph query service.” In: *arXiv preprint arXiv:2202.12530* (2022).

[67] Bingqing Lyu et al. *A Graph-Native Query Optimization Framework*. 2024. arXiv: 2401.17786 [cs.DB]. URL: https://arxiv.org/abs/2401.17786.

[68] Wenfei Fan et al. “GraphScope: A Unified Engine For Big Graph Processing.” In: *Proceedings of the VLDB Endowment* 14 (Aug. 2021), pp. 2879–2892. DOI: 10.14778/3476311.3476369.

[69] Holden Karau and Mika Kimmins. *Scaling Python with Dask*. " O’Reilly Media, Inc.", 2023.

[70] PyG Team. *Scaling Up GNNs via Remote Backends*. 2024. URL: https://pytorch-geometric.readthedocs.io/en/latest/advanced/remote.html (visited on 07/15/2024).

[71] ArangoDB. *ArangoDB, the database for graph and beyond*. 2024. URL: https://arangodb.com (visited on 07/06/2024).

[72] Surajit Chaudhuri. “An overview of query optimization in relational systems.” In: *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS ’98. Seattle, WA, United States of America: Association for Computing Machinery, 1998, 34–43. ISBN: 0897919963. DOI: 10.1145/275487.275492. URL: https://doi.org/10.1145/275487.275492.

[73] Bronwyn H Hall, Adam B Jaffe, and Manuel Trajtenberg. *The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools*. Working Paper 8498. National Bureau of Economic Research, 2001. DOI: 10.3386/w8498. URL: http://www.nber.org/papers/w8498.

[74] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. “Graphs over time: densification laws, shrinking diameters and possible explanations.” In: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. KDD ’05. Chicago, Illinois, USA: Association for Computing Machinery, 2005, 177–187. ISBN: 159593135X. DOI: 10.1145/1081870.1081893. URL: https://doi.org/10.1145/1081870.1081893.

[75] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. MORGAN KAUFMANN PUBL Incorporated, 2001. ISBN: 9781493303540. URL: https://books.google.com/books?id=X1QfogEACAAJ.

[76] Yu Wang et al. "Knowledge graph prompting for multi-document question answering." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. 17. 2024, pp. 19206–19214.

[77] Erik F. Tjong Kim Sang and Fien De Meulder. "Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition." In: *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*. 2003, pp. 142–147. URL: https://www.aclweb.org/anthology/W03-0419.

[78] Daniel Bienstock et al. "A note on the prize collecting traveling salesman problem." In: *Math. Program.* 59.1–3 (Mar. 1993), 413–420. ISSN: 0025-5610.

[79] Alex Mallen et al. "When not to trust language models: Investigating effectiveness of parametric and non-parametric memories." In: *arXiv preprint arXiv:2212.10511* (2022).

[80] Xi Ye et al. "RNG-KBQA: Generation Augmented Iterative Ranking for Knowledge Base Question Answering." In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Smaranda Muresan, Preslav Nakov, and Aline Villavicencio. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 6032–6043. DOI: 10.18653/v1/2022.acl-long.417. URL: https://aclanthology.org/2022.acl-long.417.

[81] Vladimir Karpukhin et al. *Dense Passage Retrieval for Open-Domain Question Answering*. 2020. arXiv: 2004.04906 [cs.CL]. URL: https://arxiv.org/abs/2004.04906.

[82] Zhilin Yang et al. "HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering." In: *Conference on Empirical Methods in Natural Language Processing*. 2018. URL: https://api.semanticscholar.org/CorpusID:52822214.

[83] Xanh Ho et al. "Constructing A Multi-hop QA Dataset for Comprehensive Evaluation of Reasoning Steps." In: *Proceedings of the 28th International Conference on Computational Linguistics*. Ed. by Donia Scott, Nuria Bel, and Chengqing Zong. Barcelona, Spain (Online): International Committee on Computational Linguistics, Dec. 2020, pp. 6609–6625. DOI: 10.18653/v1/2020.coling-main.580. URL: https://aclanthology.org/2020.coling-main.580.

[84] Wikipedia contributors. *Baroque music — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Baroque_music&oldid=1276068475. [Online; accessed 21-February-2025]. 2025.

[85] Feng Xia et al. "Random walks: A review of algorithms and applications." In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 4.2 (2019), pp. 95–107.

[86] Petar Veličković et al. *Graph Attention Networks*. 2018. arXiv: 1710.10903 [stat.ML]. URL: https://arxiv.org/abs/1710.10903.

[87] Swarnadeep Saha et al. *ExplaGraphs: An Explanation Graph Generation Task for Structured Commonsense Reasoning*. 2021. arXiv: 2104.07644 [cs.CL]. URL: https://arxiv.org/abs/2104.07644.

[88] Drew A. Hudson and Christopher D. Manning. "GQA: A New Dataset for Real-World Visual Reasoning and Compositional Question Answering." In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 6693–6702. DOI: 10.1109/CVPR.2019.00686.

[89] Linhao Luo et al. *Reasoning on Graphs: Faithful and Interpretable Large Language Model Reasoning*. 2024. arXiv: 2310.01061 [cs.CL]. URL: https://arxiv.org/abs/2310.01061.

[90] Wen tau Yih et al. "The Value of Semantic Parse Labeling for Knowledge Base Question Answering." In: *Annual Meeting of the Association for Computational Linguistics*. 2016. URL: https://api.semanticscholar.org/CorpusID:13905064.

[91] Sewon Min et al. *Compositional Questions Do Not Necessitate Multi-hop Reasoning*. 2019. arXiv: 1906.02900 [cs.CL]. URL: https://arxiv.org/abs/1906.02900.

[92] Matthijs Douze et al. *The Faiss library*. 2025. arXiv: 2401.08281 [cs.LG]. URL: https://arxiv.org/abs/2401.08281.

[93] Jeff Johnson, Matthijs Douze, and Hervé Jégou. "Billion-scale similarity search with GPUs." In: *IEEE Transactions on Big Data* 7.3 (2019), pp. 535–547.

[94] Pinecone Systems. *Product Quantization: Compressing High-Dimensional Vectors by 97%*. 2024. URL: https://www.pinecone.io/learn/series/faiss/product-quantization/ (visited on 11/01/2024).

[95] Herve Jégou, Matthijs Douze, and Cordelia Schmid. "Product Quantization for Nearest Neighbor Search." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.1 (2011), pp. 117–128. DOI: 10.1109/TPAMI.2010.57.

[96] Nikolay Sakharnykh. *Maximizing Unified Memory Performance in CUDA*. Nov. 2017. URL: https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/ (visited on 11/14/2024).

[97] Pinecone Systems. *Introduction to Facebook AI Similarity Search (FAISS)*. 2024. URL: https://www.pinecone.io/learn/series/faiss/faiss-tutorial/ (visited on 11/01/2024).

[98] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[99] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. arXiv: 1907.11692 [cs.CL]. URL: https://arxiv.org/abs/1907.11692.

[100] Nils Reimers and Iryna Gurevych. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. 2019. arXiv: 1908.10084 [cs.CL]. URL: https://arxiv.org/abs/1908.10084.

[101] Xanh Ho et al. *2WikiMultihopQA: A Dataset for Comprehensive Evaluation of Reasoning Steps*. 2021. URL: https://github.com/Alab-NII/2wikimultihop.

[102] Wikipedia contributors. *Toyota — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-November-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Toyota&oldid=1256782570.

[103] Wikipedia contributors. *Toyota Tacoma — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-November-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Toyota_Tacoma&oldid=1251968772.

[104] Wikipedia contributors. *Pickup truck — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-November-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Pickup_truck&oldid=1256096237.

[105] LangChain Inc. *How to construct knowledge graphs*. Nov. 2024. URL: https://python.langchain.com/docs/how_to/graph_constructing/ (visited on 11/15/2024).

[106] Alexandria Barghi. *BitGraph: A C++ Backend for the Gremlin Traversal Language with GPU Acceleration*. 2024. URL: https://github.com/bgamer50/bitgraph (visited on 02/24/2025).

[107] NVIDIA Corporation. *NVIDIA Puts Grace Blackwell on Every Desk and at Every AI Developer's Fingertips*. 2025. URL: https://nvidianews.nvidia.com/news/nvidia-puts-grace-blackwell-on-every-desk-and-at-every-ai-developers-fingertips (visited on 02/24/2025).

[108] Jianguo Wang et al. "Milvus: A Purpose-Built Vector Data Management System." In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 2614–2627.

[109] NVIDIA Corporation. *cuVS: Getting Started*. 2025. URL: https://docs.rapids.ai/api/cuvs/nightly/getting_started/ (visited on 02/24/2025).