Graph Analytics on Relational Databases

by

Praynaa Rawlani

S.B., Electrical Engineering and Computer Science, MIT (2013)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2014

© Massachusetts Institute of Technology 2014. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute
publicly paper and electronic copies of this thesis document in whole or in part in
any medium now known or hereafter created.

Author .................................................................................. Praynaa Rawlani
Department of Electrical Engineering and Computer Science
August 22, 2014

Certified by ................................................................. Samuel Madden
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ................................................................. Prof. Albert R. Meyer, Chairman
Masters of Engineering Thesis Committee
Graph Analytics on Relational Databases

by

Praynaa Rawlani

Submitted to the Department of Electrical Engineering and Computer Science on August 22, 2014 in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Graph analytics has become increasingly popular in recent years. Conventionally, data is stored in relational databases that have been refined over decades, resulting in highly optimized data processing engines. However, the awkwardness of expressing iterative queries in SQL makes the relational query-processing model inadequate for graph analytics, leading to many alternative solutions. Our research explores the possibility of combining a more natural query model with relational databases for graph analytics. In particular, we bring together a graph-natural vertex-centric query interface to highly optimized column-oriented relational databases, thus providing the efficiency of relational engines and ease-of-use of new graph systems. Throughout the thesis, we used stochastic gradient descent, a loss-minimization algorithm applied in many machine learning and graph analytics queries, as the example iterative algorithm.

We implemented two different approaches for emulating a vertex-centric interface on a leading column-oriented database, Vertica: disk-based and main-memory based. The disk-based solution stores data for each iteration in relational tables and allows for interleaving SQL queries with graph algorithms. The main-memory approach stores data in memory, allowing faster updates. We applied optimizations to both implementations, which included refining logical and physical query plans, applying algorithm-level improvements and performing system-specific optimizations.

The experiments and results show that the two implementations provide reasonable performance in comparison with popular graph processing systems. We present a detailed cost analysis of the two implementations and study the effect of each individual optimization on the query performance.

Thesis Supervisor: Samuel Madden
Title: Professor of Electrical Engineering and Computer Science
Acknowledgements

This thesis won't have been possible without the help, support and guidance of many people. I want to take this opportunity to express my gratitude for each of them.

First and foremost, I would like to thank my professor and thesis supervisor, Sam Madden, for giving me the chance to do my MEng thesis with him. I really appreciate your friendliness and supportive attitude, which made it very easy for me to express my ideas and opinions to you. You made sure that I could work on a project of my interest and were always available whenever I needed your advice. Your readiness to respond to emails and your guidance throughout the course of the thesis helped tremendously. I learnt a lot from the technical discussions we had regarding relational databases and different graph processing systems, as well as from the classes I have taken with you.

I would also like to thank my post-doctoral supervisor, Alekh Jindal. Your mentorship was an invaluable resource for my thesis. You were always available to answer my questions and made sure that I could continue my thesis smoothly. Your technical knowledge of different big data systems was really inspiring and I am really fortunate to contribute to this research work with you.

I am indebted to my family for whatever I have accomplished in my life. Thank you so much Mumy and Papa for loving, guiding and supporting me unconditionally throughout this journey. You have struggled more than I have to make sure I receive good education and live an easy life. I have learnt to take new challenges just because I know you will always be there for me if I fail. Thank you my sister Preeyaa, my brother Arun and my brother-in-law Sanjay for showing faith in me at times when I gave up. When I got admitted to MIT and was afraid of leaving home, my family believed in me more than I did and encouraged me to step out and accept the opportunity. I don't have enough words to thank you guys for your support and I can't imagine my life without you all.

Being thousands of miles away from home is not easy and especially when the academic life is as challenging as MIT's, it's natural to get homesick. I won't have survived at MIT had I not met some amazing people whom I can call my friends. Thank you so much for being my strength and making these five years the best time of my life. Special thanks to Afrah, Radhika, Isra, Rik and Divya for making sure that I remain focused and sane during the past few months when I was wrapping up my thesis.

I would also like to thank my academic advisor Srini Devadas for mentoring me throughout the five years at MIT. Last but not the least, thanks Anne Hunter, Linda Sullivan, all my professors, TAs, LAs and MIT admissions -- all of you have contributed to my learning and experience at MIT. I can't believe it's getting over so soon but I'm so glad that I can call MIT my second home.
Contents

CHAPTER 1: INTRODUCTION ................................................................. 9
  1.1 OVERVIEW ........................................................................ 9
  1.2 THESIS OUTLINE .............................................................12

CHAPTER 2: BACKGROUND AND RELATED WORK .................................. 14
  2.1 SPECIALIZED GRAPH ANALYTICS SYSTEMS .........................14
    2.1.1 Why Graph Analytics Systems? .....................................15
    2.1.2 Specialized Graph Systems ............................................17
  2.2 GRAPH ANALYTICS USING RELATIONAL SYSTEMS ..................22
    2.2.1 Why consider relational systems? ...................................22
    2.2.2 Solutions taking a relational approach ............................24
    2.2.3 Our Approach ..........................................................26
  2.3 PREGEL AND GRAPHLAB ..................................................27
    2.3.1 Pregel/Giraph ..........................................................27
    2.3.2 PowerGraph (GraphLab 2.2) .........................................29
  2.4 EXAMPLE ALGORITHM: STOCHASTIC GRADIENT DESCENT .........29
    2.4.1 Collaborative Filtering and SGD .....................................30
    2.4.2 Why SGD? ............................................................31

CHAPTER 3: DISK-BASED, VERTEX CENTRIC QUERY INTERFACE FOR
  RELATIONAL DATABASES .............................................................33
  3.1 MOTIVATION AND OVERVIEW ............................................33
  3.2 ARCHITECTURE OF THE DISK-BASED APPROACH ....................34
    3.2.1 Data loading and storage ..............................................34
    3.2.2 Stored Procedure/Coordinator .......................................36
    3.2.3 Workers/Threads ......................................................37
    3.2.4 Update/Compute function .............................................38
  3.3 QUERY PLAN .......................................................................39
  3.4 PREGEL-LIKE API ............................................................40
  3.5 ADVANTAGES AND LIMITATIONS ........................................41

CHAPTER 4: MAIN-MEMORY BASED, VERTEX-CENTRIC QUERY INTERFACE
  FOR RELATIONAL DATABASES .......................................................45
  4.1 MOTIVATION .....................................................................45
  4.2 ARCHITECTURE OF THE MAIN-MEMORY APPROACH .................47
    4.2.1 Data Storage ............................................................47
    4.2.2 Coordinator .............................................................50
    4.2.3 Worker .................................................................51
    4.2.4 Update/Compute function ............................................54
  4.3 QUERY PLAN .....................................................................54
  4.4 PREGEL-LIKE API ............................................................55
  4.5 ADVANTAGES AND LIMITATIONS ........................................56

CHAPTER 5: OPTIMIZATIONS ..............................................................58
  5.1 GENERAL OPTIMIZATIONS ................................................58
    5.1.1 Physical Optimizations ................................................58
      5.1.1.1 Load balancing / Partitioning ....................................59
    5.1.2 Logical Optimizations ................................................60
      5.1.2.1 Unions of input tables instead of using joins ..................60
      5.1.2.2 Sorting the input before feeding it to the workers ............61
CHAPTER 6: EXPERIMENTS AND RESULTS ........................................... 71
  6.1 BENCHMARK ENVIRONMENT............................................. 71
    6.1.1 Dataset ........................................................................ 71
    6.1.2 Testbed ........................................................................ 72
  6.2 PERFORMANCE ..................................................................... 73
    6.2.1 Comparison with GraphLab and Giraph .................... 73
    6.2.2 Effects of Optimizations ............................................. 75
  6.3 COST ANALYSIS .................................................................. 76
    6.3.1 Methodology ............................................................. 77
    6.3.2 Results ........................................................................ 79
      6.3.2.1 Cost Analysis of Disk-based Approach ............. 80
      6.3.2.2 Cost Analysis of Main-Memory Approach ....... 81
  6.4 GENERAL OPTIMIZATIONS APPLIED TO DISK-BASED APPROACH 82
    6.4.1 Better Load Balancing .............................................. 82
    6.4.2 String vs Floats for representing latent vectors .......... 84
    6.4.3 Simplifying union of input tables ........................................................................ 85
    6.4.4 Compiler 03 optimizations ....................................... 86
  6.5 MAIN-MEMORY APPROACH SPECIFIC OPTIMIZATIONS ........ 87
    6.5.1 Synchronization wait time ....................................... 88
    6.5.2 Radix vs Quick sort ............................................... 89
    6.5.3 Reducing input size .................................................. 89

CHAPTER 7: ONGOING AND FUTURE WORK ....................................... 91
  7.1 EXTENDING SHARED MEMORY ......................................... 91
  7.2 IN-DATABASE GRAPH ANALYTICS .................................... 92

CHAPTER 8: CONCLUSION .................................................................. 94

APPENDIX A ............................................................................... 96

APPENDIX B ............................................................................... 97

REFERENCES .............................................................................. 99
List of Figures

FIGURE 1 SUMMARY OF GRAPH ANALYTICS TOOLS AND RELATIONAL SYSTEMS .......... 16
FIGURE 2 QUERY PLAN FOR GIRAPH/PREGEL................................................................. 28
FIGURE 3 DATA-FLOW PER SUPERSTEP FOR THE DISK-BASED IMPLEMENTATION. ....... 35
FIGURE 4 SCHEMA FOR NODES, VERTICES AND MESSAGES TABLES FOR RELATIONAL
EMULATION OF VERTEX-CENTRIC MODEL ............................................................. 36
FIGURE 5 UDTF EXECUTION BY THE COORDINATOR .................................................... 37
FIGURE 6 QUERY PLAN FOR DISK-BASED APPROACH .............................................. 40
FIGURE 7 CODE SNIPPET FOR SGD FOR VERTEX-CENTRIC MODEL........................... 42
FIGURE 8 MESSAGE STRUCT FOR IN-MEMORY ARCHITECTURE ............................... 49
FIGURE 9 MESSAGE STORE STRUCT FOR IN-MEMORY APPROACH ............................. 49
FIGURE 10 INTERACTIONS BETWEEN WORKERS DURING A SUPER-STEP. WORKER 3
sends a message for node 15, which resides in worker 1's node array. Thus
the message is written to the message store for w3 in w1's set of message
stores. Message store also exists for local messages e.g. w2 writing
message for node 2. The write pointers are incremented after the
message is written and the read pointers are incremented as compute
processes the input messages. ................................................................. 49
FIGURE 11 OVERVIEW OF THE ARCHITECTURE OF THE IN-MEMORY APPROACH ....... 52
FIGURE 12 QUERY PLAN FOR MAIN-MEMORY ARCHITECTURE .................................. 55
FIGURE 13 SIMPLER QUERY PLAN FOR DISK-BASED APPROACH WITH UNION INSTEAD
OF UNIONS ............................................................................................... 63
FIGURE 14 SIMPLER QUERY PLAN FOR MAIN-MEMORY APPROACH WITH UNION OF
BETWEEN WORKERS DURING A SUPER-STEP. WORKER 3
sends a message for node 15, which resides in worker 1's node array. Thus
the message is written to the message store for w3 in w1's set of message
stores. Message store also exists for local messages e.g. w2 writing
message for node 2. The write pointers are incremented after the
message is written and the read pointers are incremented as compute
processes the input messages. ................................................................. 49
FIGURE 11 OVERVIEW OF THE ARCHITECTURE OF THE IN-MEMORY APPROACH ....... 52
FIGURE 12 QUERY PLAN FOR MAIN-MEMORY ARCHITECTURE .................................. 55
FIGURE 13 SIMPLER QUERY PLAN FOR DISK-BASED APPROACH WITH UNION INSTEAD
OF UNION OF JOINS .................................................................................. 63
FIGURE 14 SIMPLER QUERY PLAN FOR MAIN-MEMORY APPROACH WITH UNION OF
NODES AND EDGES TABLE ........................................................................... 69
FIGURE 15 COMPARISON OF POPULAR GRAPH PROCESSING SYSTEMS, GIRAPH AND
GRAPHLAB, WITH OUR IMPLEMENTATIONS OF VERTEX-CENTRIC QUERY INTERFACE
WITH VERTICA. THE TIMES SHOWN ARE FOR RUNNING 10 ITERATIONS OF SGD .... 74
FIGURE 16 EFFECTS OF OPTIMIZATION FOR DISK-BASED AND SHARED MEMORY
APPROACH FOR RUNNING 10 SUPERSTEPS. NOTE: THE FIRST VERSION OF MAIN-
MEMORY APPROACH ALREADY INCORPORATES THE GENERAL OPTIMIZATIONS AS
APPLIED TO DISK-BASED VERSION I.E. METIS PARTITIONING, COMPILER
OPTIMIZATIONS AND UNIONS OF TABLES. ...................................................... 75
FIGURE 17 COST BREAKDOWN OF 10 SUPERSTEPS IN DISK-BASED IMPLEMENTATION WITH
AND WITHOUT OPTIMIZATIONS ........................................................................ 75
FIGURE 18 COST BREAKDOWN OF 10 UPDATES OF SGD WITH MAIN-MEMORY APPROACH
BEFORE AND AFTER OPTIMIZATIONS (10 ITERATIONS WITHOUT OPTIMIZATION AND
11 ITERATIONS WITH OPTIMIZATIONS) ............................................................ 80
FIGURE 19 WORKLOAD WITH HASH PARTITIONING .................................................. 83
FIGURE 20 WORKLOAD WITH METIS PARTITIONING ................................................ 83
FIGURE 21 COMPARISON OF PER SUPERSTEP TIMING WITH HASH AND METIS
PARTITIONING ................................................................................................ 84
FIGURE 22 COMPARISON OF PER SUPERSTEP TIMING WITH STRINGS AND FLOATS FOR
STORING LATENT VECTORS ............................................................................ 85
FIGURE 23 COMPARISON OF PER SUPERSTEP TIMING WITH AND WITHOUT A SIMPLIFIED
LOGICAL QUERY PLAN .................................................................................. 86
FIGURE 24 COMPARISON OF PER SUPERSTEP TIMING WITH AND WITHOUT COMPILER OPTIMIZATIONS.................................................................87
FIGURE 25 CHANGE IN SYNCHRONIZATION COST WITH REDUCED WAIT TIME FOR 10 ITERATIONS...............................................................88
FIGURE 26 CHANGE IN SORTING COSTS BY USING QUICK SORT INSTEAD OF RADIX SORT FOR 10 ITERATIONS........................................89
FIGURE 27 THE PLOT SHOWS THE GAIN IN PERFORMANCE AS THE IMPLEMENTATION-SPECIFIC OPTIMIZATIONS WERE APPLIED INCREMENTALLY TO THE MAIN-MEMORY APPROACH. THE TIMES SHOWN ARE FOR 10 ITERATIONS OF SGD. ..........................90
List of Tables

TABLE 1 SUMMARY OF OPTIMIZATIONS ................................................................. 70
TABLE 2 COST BREAKDOWN COMPONENTS FOR DISK-BASED .......................... 77
TABLE 3 COST BREAKDOWN COMPONENTS FOR MAIN-MEMORY ...................... 77
Chapter 1: Introduction

1.1 Overview

Over the years, graph analytics has received immense attention. Graphs are often used to capture data dependencies and interactions between commodities. E-commerce, social networks, genetic data and transportation, to name a few, use graphs to store relationships between users, items, genes, etc. Analyzing the topology and properties of these graphs can be extremely useful in deriving key insights and extracting useful information. Some applications include recommendation systems, web search and social graph analytics. Thus, developing tools for large-scale graph analytics, with high performance and ease of use, has captured the interest of many researchers.

Traditionally, for many real-world applications, data is collected and stored in relational databases that are optimized for sequential scans, suggesting a good storage model for graphs. However, since most of the graph algorithms are iterative in nature and representing these algorithms using traditional SQL queries can be non-trivial and quite complex e.g. it involves multiple self-joins, traditional relational database systems are often considered inadequate for graph analysis. The awkward query model of relational databases for graph analytics has led to many alternate solutions being proposed. These include vertex-centric systems like Pregel [1], Giraph [2], GraphLab[3] and graph databases e.g. Neo4j [4], TAO [5] and FlockDB [6].
Another property of graphs is that both nodes and edges often have rich information associated with them. Graph analytics often involve selecting a subset of a graph based on these properties, or compute some statistics on the output of the analytics. These operations are very well served by relational database systems. Hence, these new graph processing systems are typically used alongside a relational database, which acts as the primary data store and supports all relational queries. This means that to run one of these new graph systems, users first must export all data from the RDBMS into the supported input format for the graph processing system before analytics can happen. Hence, the key question we address in this thesis is: Can we use relational databases as a single platform that supports end-to-end data analysis, rather than using several systems?

Specifically, this thesis aims to find a middle ground between relational databases and the more popular graph processing systems in the hope of developing a complete data analytics platform. We studied some popular graph analytics systems, mainly focusing on the vertex-centric systems e.g. Pregel and GraphLab, which have become very popular. These systems allow users to think of graph computations in terms of vertices and edges, which makes the implementation of graph algorithms more natural. Our research sought to build a vertex-centric query interface on top of a relational engine that can provide both the efficiency of a relational engine and the ease-of-use of a new graph system. Our results show scalable and efficient graph analytics is possible with relational databases.
Throughout the course of this thesis, we use stochastic gradient descent (SGD) as the representative algorithm for iterative computations. SGD is a general optimization technique that minimizes an objective function. It is used as a training algorithm for many machine learning applications, e.g. support vector machines, neural networks and linear regression. Specifically, we study SGD in the context of collaborative filtering, a technique used by recommendation systems to predict what items might be liked by users. Because the algorithm is computationally intensive, widely used, and not much explored as a use-case by many graph-processing systems, it serves as a good example algorithm.

The major contributions of this thesis are as follows:

a) We study different graph processing systems and the features provided by column-oriented stores such as Vertica [7], to explore the possibility of developing a vertex-centric interface on top a relational engine.

b) Using SGD as a representative algorithm running on Vertica, we prototyped and implemented two different techniques, disk-based and shared memory-based, for running typical iterative algorithms in a relational database.

c) We explore four broad classes of optimizations: (1) optimized physical designs (2) optimizations for logical query plans, (3) algorithm-specific optimizations (4) optimizations that improve the performance of the column-oriented relational engines e.g. Vertica. Some of these optimizations were applicable to both disk-based and shared-memory
approaches, while others were more implementation-specific. Specific techniques we study include using different methods for storing data, pipelining results, partitioning the input data, synchronization of intermediate super-steps and SGD-related data representations.

d) We perform a cost analysis for both approaches to further understand the advantages and limitations. We use different experiments to analyze how the individual components and their interactions with each other contribute to the cost breakdowns.

e) We evaluate our designs by comparing our approaches with popular vertex-centric systems like GraphLab and Pregel. We find that the disk-based approach outperforms Giraph by a factor of 2, but is 9 times slower than the shared-memory based GraphLab. However, for graphs that fit in memory, our shared-memory approach performs 22 times better than Giraph and is also slightly faster than GraphLab.

f) We analyze how each of the optimizations affects the performance in both the disk-based and shared-memory implementations.

1.2 Thesis outline

In Chapter 2, we motivate the thesis by comparing the specialized graph processing systems with relational databases for graph analytics. We also survey recent popular graph processing systems and include an overview of SGD in context of collaborative filtering. Chapter 3 introduces the architecture of our disk-based approach to provide a vertex-centric interface for relational databases.
For each iteration, the disk-based approach stores input and output data in relational tables, thus making it possible to execute SQL operations between iterations. Chapter 4 then introduces our novel shared-memory approach, which stores the input data in main memory, allowing for faster data access and updates. For both the disk-based and shared-memory based approach, we applied several optimizations to logical and physical query plans as well as used some algorithm-specific and system-level improvements, which are presented in Chapter 5. Chapter 6 covers the evaluation of the two techniques in contrast with graph processing systems and provides a cost breakdown to summarize the effect of different optimizations. We present on-going and future work in Chapter 7 and conclude in Chapter 8.
Chapter 2: Background and Related Work

This chapter presents a detailed discussion of the need for graph processing systems and the potential advantages that relational databases can provide for graph analytics, developing a motivation for our approach to find a hybrid solution. We present an overview of the existing graph data management systems, including graph databases, MapReduce based solutions and the more popularly used, vertex-centric model based systems. We also discuss the efforts by other researchers to perform graph analytics on relational databases. At the end of this chapter, we provide a brief description of stochastic gradient descent as the training algorithm for collaborative filtering.

2.1 Specialized Graph Analytics Systems

Many graph analytical solutions have been proposed in the recent years that are non-relational in nature. Thus, it is important to understand what challenges relational systems present for graph analytics and what benefits are provided by these specialized systems. In this section, we discuss the motivation behind the purpose-built graph analytical systems. In the second part of the section, we delve into the existing graph based systems, which can be roughly categorized into two groups: transactional and analytical.
2.1.1 Why Graph Analytics Systems?

Graphs are used to represent connections between different entities, e.g. relations between people in social networks or between documents in web search, thus storing useful information about interactions. However, it is hard to efficiently capture the topology or locality information of graphs in relational databases. Traversing the graphs stored in relational tables require random data access and since all neighboring nodes might not be located physically close to each other on disk, it can lead to random I/O, making the algorithms extremely expensive to run. On a similar note, some algorithms require accessing only a subset of the nodes. However, since the physical layout of the nodes in RDBMS doesn’t capture the graph structure well, one can end up doing complete table scans even when referring to a small sub-graph. The relational data model poses difficulty in efficiently storing the graph data physically, which is one of the reasons why specialized graph systems were proposed.

Most graph algorithms are iterative in nature and expressing them in SQL efficiently is both non-intuitive and time consuming. This is because instead of nodes and edges, users have to think about querying relational tables, which is not a natural way to think about graphs. Moreover, often algorithms involve following some defined paths or doing complete graph traversals. However, because relational databases aren’t aware of the underlying graph structure,
traversing graphs in RDBMS results in doing costly multi-joins and self-joins. Thus, using relational databases can result in over-complicated and inefficient queries even for simple algorithms like depth or breadth first search, which are often the basis for more complex graph problems. Specialized analytical tools, on the other hand, capture the graph structure when storing data and provide an abstraction layer that gives an easy-to-use yet powerful query interface for expressing graph algorithms. With these systems, users can write simple programs in terms of nodes and edges, without worrying about the underlying

---

**Figure 1** Summary of graph analytics tools and relational systems
specifics of the system. Moreover, users don't have to worry about partitioning graphs, distributing sub-graphs across machines or coordinating the communication between machines.

2.1.2 Specialized Graph Systems

Given the demand of graph analytics and awkwardness of expressing iterative algorithms in SQL, many specialized systems have been proposed in the past few years to capture the properties of a graph and facilitate graph queries. As shown in Figure 1, these systems can be roughly categorized into graph databases, map-reduce based systems, array-based approaches and vertex-centric tools.

**Graph databases**

Graph databases store data using native data structures such that the graph structure is preserved naturally. Data is stored as nodes and edges and data is often indexed upon attributes or properties of entities. Each node has a pointer to the adjacent nodes, which makes graph traversal easier. Thus, it avoids cost of doing multiple joins and lookups that would otherwise occur in relational databases e.g. if one wants to find information about a node’s 3-hop neighbors.

Neo4j [4] is a popular open-source graph database that stores data in nodes connected by directed edges. It aims to provide full ACID properties, an easily expressible query language along with scalability and durability of data. TAO [5] is used by the social networking website, Facebook, and represents objects as
nodes and associations as directed edges. It is heavily optimized for reads, as that is one of the needs for the website and favors efficiency and scalability over consistency. However, the system does not provide an advanced graph processing API and is more focused on simpler queries and faster updates. FlockDB [6] wraps around MySQL to build a distributed graph processing system supporting high rate of add/update/remove operations, but doesn't support multi-hop queries. Other popular transactional graph processing systems include key-value stores like HypergraphDB[8], similar to Neo4j, and RDF stores like AllegeroGraph[9]. In general, the graph databases are designed for online, low latency transactions.

**MapReduce/Hadoop-based Frameworks**

In order to support offline, high throughput graph analytics, several frameworks have been proposed and optimized upon. Due to the simplicity of defining certain algorithms as map and reduce tasks, MapReduce/Hadoop has gained a lot of popularity as a platform for large-scale data processing and several Hadoop-based frameworks have been proposed to support iterative queries. Systems like HaLoop [10] and Twister [11] are built around Hadoop to support iterative queries. While these systems provide efficient query processing, writing these queries as MapReduce jobs almost as non-intuitive as writing them in SQL. Also, since the state of graphs is not shared, the system requires to explicitly send the graph structure across tasks or iterations, making the process non-scalable.
Array-based approaches

Pegasus [12] also uses the Hadoop architecture to provide a graph mining package for large graphs. However, unlike the other Hadoop-based systems, Pegasus proposes that many graph problems can be generalized to matrix-vector multiplications and thus allows users to express their graph queries as matrix operations.

Vertex-centric models

Vertex-centric based systems offer an interface that allows a user to think through the algorithm as computation on a vertex or a node. Each vertex has a state and is aware of its outgoing edges. Computations usually involve processing messages sent by neighbors with incoming edges to the node and/or updating the value of the node. Giraph [2] is based on the Google’s Pregel [1] model, which follows the Bulk Synchronous Parallel (BSP) protocol, according to which vertices use messages from a previous iteration to update their values. GraphLab, on the other hand, is an asynchronous distributed graph processing platform, where the data on vertices is updated directly. We used Giraph and GraphLab for comparisons throughout the thesis. Section 2.3 gives a detailed overview of the two systems.

GPS[13], an open-source distributed graph processing systems, also belongs to the BSP domain. It extends the Pregel API with a master compute option to perform global computations or cascade multiple vertex-centric computations. It
is optimized for different node degree distributions and provides support for dynamic repartitioning of nodes across machines during iterations.

**Other vertex-centric extensions**

One of the major problems with graph algorithms is that the computations are often I/O intensive. The problem can be mitigated if the graph topology resides in memory instead of disk, thus limiting the number of random I/Os that are expensive. However, doing so limits the systems in terms of scalability. Trinity [14] tries to address this problem by implementing a random data-access abstraction on top of a distributed memory cloud. Trinity uses graph access patterns to optimize memory management and network communication, thus allowing for faster graph explorations. This allows the system to handle both online, low latency query processing tasks as well as high throughput, offline graph analytics. Realizing the diverse data and network communication models followed by graph algorithms, the system provides the user with the flexibility to define graph schema, data models, communication protocols and computations, unlike Pregel and GraphLab. However, the graph engine doesn’t support ACID properties and doesn’t guarantee serializability in concurrent threads.

Systems based on the BSP model, e.g. Giraph, impose synchronization barriers, which can lead to slower convergence of certain algorithms. Some algorithms can be executed asynchronously with faster convergence rates. Grace [15], a single machine parallel processing system, extends the Pregel API such that the programming logic is separated from the execution logic. Since the underlying
model is BSP, the simplicity of the programming model is preserved, while providing more flexibility. Users can relax data dependencies, specify which messages to process and in what order, and define scheduling policy for vertices. Thus, it allows users to control the consistency and isolation level of messages generated and received by vertices. However, Grace requires that the graph structure is immutable.

For applications that perform small computations per vertex, Xie et al. [16] extended Grace to add block-oriented computation capability that allows such applications to scale well. This extension allows users to update blocks or partitions of a graph instead of vertices and define different scheduling policies for blocks than that for vertices. These features improve locality and overhead of scheduling, making it possible to achieve faster convergence for algorithms.

Rather than using “think-like-a-vertex” strategy, Giraph++ [17] uses a “think-like-a-graph” model. The motivation is that the vertex-centric models partition the original graph but does not exploit the sub-graph topology in the partitions. Giraph++ proposes that computation and communication can be much faster if the system utilizes the sub-graphs in a partition. For instance, it can allow us to apply message-batching optimizations for faster propagation of information than the pure vertex-centric model.
2.2 Graph Analytics using Relational Systems

Although the graph systems above offer several different approaches to do graph analytics, relational systems are widely used for data storage and transactional query processing. Being such stable and well-researched systems, it is important to realize the advantages that relational databases can provide for graph analytics queries. This section is divided into two parts; the first part discusses why relational database systems are important and what benefits they provide for query processing in general. The second part gives an overview of the different attempts that have been done so far at using relational systems to do graph analysis.

2.2.1 Why consider relational systems?

The database community has spent decades of research to optimize RDBMS, maturing the databases as time has progressed. As a result, relational databases offer many features including ACID properties, fault-tolerance, recovery and checkpointing, which make them complete, robust systems. Because of these advantages, most of the real-world data naturally resides in relational tables. Using a different system for graph analysis requires users to export input data from these relational tables in another format. It also requires reloading the data every time updates are made to the source tables. Moreover, users cannot take advantage of the advanced and optimized features of RDBMS while graph computations are being performed on another system.
Graph processing systems, like Pregel and GraphLab, use a static query and execution plan based on the underlying architecture of these systems. On the other hand, relational databases allow users to define their computations declaratively while the query optimizers and cost estimators consider multiple ways to run a query before choosing the optimal plan. Hence, with RDBMS, unlike graph processing systems, users can write their queries without worrying about optimizing them on the physical data.

Rich metadata is often associated with graph nodes. For example, one can imagine vertices in a social graph storing useful information about a user's likes and dislikes, location, etc. Often graph algorithms like shortest path or Page Rank doesn't involve searching the entire graphs, but instead require preprocessing the graphs to extract sub-graphs or post-processing the results to derive some useful information. Thus many data analysis queries on graphs are relational in nature and relational databases provide an adequate logical interface for carrying out these queries. For instance, if one wants to distinguish between friends and siblings, a graph management system will possibly keep separate labels on the edges indicating how two people are related [18], dwarfing graphs even more. However, this information can be easily represented as a separate table with RDBMS.

Similarly, if one wants to run queries on some subset of vertices, relational databases have indexing mechanisms to speed up the search. For graph processing systems, one might need to either pre-process or post-process data...
externally, possibly by using other scripting languages, making the whole process too complex and defeating the simplicity of these systems. Data processing can be easily done with RDBMS by using relational operators like filters, selections, projections and aggregations. Also, column-oriented databases like Vertica are highly optimized for relational data analytics and reads, which makes loading and querying data much faster than graph processing systems.

2.2.2 Solutions taking a relational approach

There has been prior research done on similar lines as our research. Welc et al. [19], argue that, contrary to popular belief, many graph problems can be solved using SQL and in-memory databases, instead of specialized systems like graph databases, with equal or better performance. They run Dijkstra on a social network graph using Neo4j, SQL-based solutions and Green-Marl Domain Specific Language and observe that graph databases like Neo4j cannot compete in performance with in-memory graph engines like Green-Marl DSL in many scenarios. However, usually graphs are large enough to not fit in memory. Also, Welc et al. noticed that SQL based versions work well for most social graph queries; however, some graph problems several graph problems are very tricky and time consuming to express in SQL. Welc, et al. suggest a hybrid approach that uses Green-Marl compiler that generates C++ code for in-memory representation and SQL code for transactional processing. However, that still requires users to learn another system, export their data to these systems and not
take advantage of an easy-to-use interface like the vertex-centric approach discussed above.

Myers et al. [18] also believe that relational engines can work as well as specialized systems. To test this hypothesis, they use code generation techniques to generate fast query plans over C++ data structures and use analogous algorithms for parallel computation on Grappa, a parallel query system. Their approach is different from ours, as they don’t aim to use relational databases for their purpose.

Lawande et al. [20] present two use cases on social network graphs, where Vertica solved graph problems with high performance using simple SQL operations. They also compare Vertica’s performance with Hadoop and Pig for counting triangles algorithm and observe that Vertica performed much better. The performance gain comes in because of pipelined execution and optimized code generation in Vertica. Moreover, if the problem space involves running on a subset of data, Vertica’s index search and projections are very efficient in that case in comparison to other systems.

Systems like Giraph have high memory consumption and large infrastructure requirements to scale well on larger graphs, which makes them resource-intensive. Moreover, they have less flexibility in the physical design of their architecture as discussed above. Pregelix [21] tries to address these problems by using a dataflow design that follows the same logical plan as Pregel but runs on
the dataflow engine, Hyracks. The system is implemented to handle both in-memory and out-of-core workloads and provides more physical flexibility.

MadLib [22] is an open-source project that aims to provide a scalable analytics library for SQL-based relational databases. The library currently implements many commonly used machine learning and statistical algorithms including linear regression and least squares using SQL statements.

While the above systems realize the advantages that relational engines can bring in doing graph analysis, there is no system that preserves the easy-to-use programming interface provided by vertex-centric approaches and use a single system to perform both relational and graph queries. Our approach is to extend the ideas and come up with a system that brings the best of both worlds together, providing not only a user-friendly solution but also an efficient, parallel graph processing system on top of a relation engine.

2.2.3 Our Approach

Given the general advantages of relational databases and the simplicity and speed of graph processing systems for analyzing graphs, it is natural to ask if there is a way to combine the best of both worlds. We aim to come up with a simple query interface that works on top of a relational database to provide an end-to-end data analytics system. For the purpose of this work, we used the vertex-centric model for the query interface and used highly optimized, column-oriented Vertica as
data store. We explain the implementation details of our approaches in more detail in the later chapters.

### 2.3 Pregel and GraphLab

We used GraphLab and Giraph, which is based on Google’s Pregel model, over the course of this thesis. The following sections give a brief overview of the two systems.

#### 2.3.1 Pregel/Giraph

Giraph [2] is an implementation of Pregel [1], a vertex-centric programming model, that runs on Hadoop. Inspired by Bulk Synchronous Parallel (BSP) model, Pregel executes each iteration as a super-step. During a super-step, a user-defined compute function is run on each vertex to update its state. The input data is stored in HDFS and the nodes and edges are initially read and partitioned by workers. After, the initial setup, each worker maintains nodes, edges and message store, where the latter stores messages received from other workers from the previous superstep.

Workers use the incoming messages to do the actual computation by running the UDF and the new state is propagated to neighboring or known vertices via messages. A super-step is completed only when all workers are done running computations on the vertices in their respective partition. When all active vertices have been processed and all workers have finished, the updates are stored and
next iteration or super-step begins. This ensures consistency and makes the programming model synchronous in nature. The algorithm ends when all vertices vote to stop or when there aren't any messages to process. Giraph execution can be expressed using a simplified logical query plan as illustrated in Figure 2. The input nodes (V) and edges (E) table are joined for the nodes that have incoming messages in the messages table (M). The input data is then partitioned and each workers executes compute function on every node in the partition assigned to it. The output is a union of new messages (M') and updated nodes (V') table.

\[ \text{M'}UV' \]

\[
\begin{align*}
\text{compute} \\
\text{partition} \\
V & \quad E \\
M
\end{align*}
\]

Figure 2 Query plan for Giraph/Pregel

The Pregel programming interface allows parallel execution, with scalability and fault tolerance. When running in a distributed setting, vertices are placed across machines and messages between nodes are sent over on the network.
2.3.2 PowerGraph (GraphLab 2.2)

GraphLab [3] is a parallel, distributed shared memory abstraction framework designed for machine learning algorithms. Unlike Pregel, which follows a message-passing model, GraphLab uses a vertex-update approach that decomposes graph problems using a Gather-Apply-Scatter (GAS) model. In the gather phase, vertices collect information about their neighboring nodes and compute a generalized sum as a function of these values. The result from the gather phase is used to update the vertex value in the apply phase. Finally, adjacent nodes are informed about the new value of the vertex in the scatter phase.

Unlike Giraph, GraphLab supports both synchronous and asynchronous computations, thus exposing greater parallelism for algorithms like shortest path where vertices can be scheduled asynchronously. It also provides more flexibility to users to define the different consistency levels e.g. vertex-consistency and edge-consistency.

2.4 Example Algorithm: Stochastic Gradient Descent

We used stochastic gradient descent (SGD), a loss-minimization optimization technique, as the example iterative algorithm to test our hybrid approaches. This section gives an overview of collaborative filtering and SGD. The section also discusses the reasons for choosing SGD to evaluate our implementation.
2.4.1 Collaborative Filtering and SGD

Collaborative Filtering (CF) is a popular machine learning method used by recommendation engines. The main objective is to learn patterns of users with similar interests and use that information to predict a user’s preferences. CF problems can be seen as matrix completion problems, with \( m \) users and \( n \) items, where the matrix values represent ratings given by a user to an item. The technique aims to fill out missing entries in the matrix i.e. predict ratings for items not yet consumed by a user.

There are several approaches to solving collaborative filtering problem, but we will focus on the commonly used latent factor model-based technique [23]. According to this approach, a vector represents each user and item, where an element in the vector measures how much a factor is important to a user or the extent to which a factor is possessed by an item. The predicted rating is calculated by taking the dot product of the user and item vectors:

\[
\text{rating}_{ui} = u \cdot i
\]

where \( u \) and \( i \) represent the user and item vectors respectively.

To ensure that the predicted rating is close to its true value, stochastic gradient descent (SGD) is used as the training algorithm for minimizing the error between predicted and known ratings. SGD is provided with an input of user-item pairs and known ratings. It is an iterative algorithm that starts with some initial latent vectors representing users and items and they are trained in each iteration according to the error between predicted and known rating for every input user-
item pair. The learning rate is controlled by $\alpha$, which decides the extent to which the prediction error will affect the latent vectors. To avoid overfitting the data to the known ratings, a regularization factor $\beta$ is used. At each iteration, SGD updates the latent user and item vectors as follows:

$$u = u - \alpha(error_{ui} \cdot i + \beta \cdot u)$$

$$i = i - \alpha(error_{ui} \cdot u + \beta \cdot i)$$

where $error$ is the difference between the predicted and known rating. The algorithm is either run for a fixed number of iterations or is allowed to converge within some epsilon.

### 2.4.2 Why SGD?

Stochastic gradient descent is an optimization technique for minimizing an objective function, used commonly in many machine learning algorithms and statistical estimation problems. The technique is used to train core machine learning models with widespread applications like support vector machines, collaborative filtering, neural networks and least squares. Given that the algorithm is much more general and sophisticated than, e.g. Page Rank or shortest paths, it serves as a good example of an iterative algorithm to test our hybrid system on.

As explained in the previous section, SGD computation involves calculating scalar and dot products per user-item pair it is trained on. This makes each iteration of
the algorithm computationally intensive than other graph problems. Thus, it can provide a good measure of how the system will perform where vertex updates are not simply taking the minimum of values. Additionally, unlike shortest paths and PageRank, SGD has been rarely focused upon when evaluating graph processing systems, despite being a general optimization technique. Because it's computationally expensive and more complex, the algorithm also has the potential of emphasizing problems in scaling to data, etc. Hence, we believe that not only would it be interesting, but also necessary to explore a more real and relevant example of iterative algorithms like SGD.
Chapter 3: Disk-based, Vertex Centric Query Interface for Relational Databases

In this section, we describe a disk-based approach to emulate the vertex-centric computation model in a relational database system.

3.1 Motivation and Overview

The goal of this section is to describe our first emulation of a vertex-centric API in a relational engine for query processing. We chose to use column-oriented relational database management system, Vertica, which is highly optimized for sequential reads and is known for fast query performance.

We implemented a generally functioning vertex-centric query interface, using a logical query plan similar to Pregel (see Figure 2). For simplicity, we used external loops to emulate the iterations or supersteps. Inside the loop, a call was made to a Vertica user-defined transform function (UDTF) that wraps the “compute” method used in vertex-centric systems. We adopted a disk-based approach: intermediate results from UDTF computations are stored in relational tables after each iteration and reloaded from disk in the next iteration. We experimented by implementing stochastic gradient descent and collaborative filtering, using the vertex-centric API.

In this chapter, we first give an overview of the architecture of the disk-based approach and provide the implementation details of emulating a disk-based,
vertex-centric query interface on Vertica. Then, we present with the details of implementing SGD in this vertex-centric model. Finally, we discuss the advantages and limitations of the disk-based approach.

3.2 Architecture of the Disk-Based Approach

This section gives an overview of the implementation details of our disk-based approach. The system has four main components: i) physical storage (the relational database), ii) a coordinator that manages the iterations, iii) workers that execute the vertex computations and iv) the user-defined vertex compute function that contains the core logic for the algorithm. The following sections describe in detail what each of these components are and how they interact with each other to provide a vertex-centric query interface with Vertica. Figure 3 illustrates the interaction between each of the components in the system.

3.2.1 Data loading and storage

The system maintains three major input tables to store the relevant graph data: a nodes table, an edges table, and a messages table. The nodes table generally contains columns for node ID, node value, node state and group ID. A node value stores some information associated with the node and the state indicates whether or not the vertex is active i.e., if it will perform computation in the next iteration.
Nodes are distributed amongst the workers based on the group ID assigned to them by the coordinator, which is discussed in more detail in the later sections. The edges table stores source and destination IDs for each edge, their corresponding group IDs and optionally some edge value. Finally, the messages table contains columns for the message source, destination, destination group and message value. Figure 4 summarizes the schema for the input tables.
The output of each UDF call includes messages sent by each vertex and updates to each vertex during a super-step. The output is stored in an intermediate table and is used to update the nodes and messages tables for the next iteration.

### 3.2.2 Stored Procedure/Coordinator

The coordinator drives the iterations by providing logic for partitioning nodes and running the super-steps. Prior to the first iteration, the coordinator uses a partitioning function to assign a group ID (dst_grp) to each node. Currently, we support hash and METIS partitioning, the latter of which is a standard, well-researched algorithm [24] for computing a balanced partitioning of nodes in a graph. We assign equal weights to all nodes and edges in the graph.

Figure 5 shows the SELECT SQL query statement that the coordinator uses to invoke the Vertica UDTF, which contains the logic for workers and performs the vertex computation. For the input to the UDTF, the coordinator takes the union of the input nodes, edges and messages tables and sorts the union by dst_id, dst_edge_id and src_id using the UDTF’s ORDER BY clause. Thus, the node data, outgoing edges and incoming messages for a vertex are grouped together before passing the input to the UDTF. The coordinator uses the UDTF’s
CREATE intermediate_output_table FROM
SELECT sgd_pregel(...) OVER (PARTITION BY dst_grp
ORDER BY dst, dst_edge_id, src_id)
FROM union_of_input_tables

Figure 5 UDTF execution by the coordinator

PARTITION OVER clause on the dst_grp to partition the input data among the
workers. The UDTF spawns the worker threads, usually equal to the number of
cores.

These workers perform the computations on each node using the compute
function provided by user. The coordinator waits for all workers to finish
computation before propagating any updates to the destination nodes. Once all
the workers are done with an iteration, the coordinator collects the results and
updates the nodes and messages tables, which serve as input to the workers in
the next iteration. Appendix A includes a complete example code snippet
showing the implementation of the coordinator for disk-based approach.

3.2.3 Workers/Threads

Our current implementation works on a single machine and the coordinator
executes a SELECT SQL query to invoke the UDTF, which starts each worker on a
separate core. The processPartition method of the UDTF contains the logic
for a worker and calls the compute method. Appendix B presents an example
code snippet for processPartition for SGD.
Each worker receives a partition of nodes, edges and messages table from the coordinator where the data for each node is already grouped together by the coordinator. It uses Vertica UDTF’s PartitionReader to read the input stream. The worker identifies and parses the input data for each vertex in the partition, i.e. the node information, its corresponding outgoing edges and its incoming messages and then executes the user-defined compute method on each of the nodes serially. Finally, the worker sends the new messages and updates generated by the compute function to the coordinator using Vertica’s PartitionWriter. Once the output is passed to the coordinator, the worker processes the next node in the partition. When all workers finish processing their partitions, the coordinator sends the resulting intermediate output table as the input to the next iteration.

The worker also exposes some other API methods similar to Pregel including modifyVertexValue(...), getVertexValue(), sendMessage(...), getOutEdges(), getMessages() and voteToHalt(), described in Section 3.3. The user can define message and vertex value type, based on the graph algorithm.

3.2.4 Update/Compute function

The Update/Compute function is defined by the user and has the core logic for the graph computation that needs to be performed. It is the equivalent of
Pregel's `compute` function. This function follows the vertex-centric model and is called by a worker during each super-step on all nodes that have at least one incoming message. The function takes as input a list of messages intended for the node, processes the messages and updates the vertex state and value. At the end, it generally makes a call to the `sendMessage` method (see Section 3.3 below) to propagate the updates or messages from the current node to its neighboring vertices. A user can use the API provided by the worker to get and update node and message values. The node can declare itself as inactive by calling `voteToHalt` to indicate that it will not participate in the computation in the next iteration, unless it receives messages from other nodes, which will turn it active. Thus, the `update/compute` function provides the simple yet powerful vertex-centric approach to solving graph analytics problems.

### 3.2 Query Plan

Figure 6 illustrates the query plan for the first version of the disk-based approach. We take a union of the input tables, where we select the nodes and edges data for only the active vertices in the current superstep, by joining both the nodes and edges tables with the messages table individually. The input data is then partitioned across the workers by the coordinator, on which the UDF or compute function is called. The output is a join of new messages and updated vertex values. These updates, stored in an intermediate output table M2, are incorporated into the nodes, edges and messages table after all workers finish their computations.
3.3 Pregel-like API

In addition to the compute function, the following Pregel-like API is exposed by the worker to the user. The methods are defined according to the graph algorithm that needs to be implemented.

1. `getMessages()`:

   Returns an array/vector of messages for the current node that is being processed. The user specifies the return type. For SGD, each message is represented as a vector.

2. `getOutEdges()`:

   Returns a vector of all node IDs that form outgoing edges with the current node that is being processed.
3. modifyVertexValue(vertex_value):
   Updates the vertex value for the current node being processed. The user
defines the type of the vertex value. For SGD, vertex value is the latent
vector associated with the vertex.

4. sendMessage(dst_id, message_value):
   This function is used by the compute function to collect all the outgoing
messages that will be sent by the current node. The dst_id corresponds
to the destination node that the message is addressed to. The updates are
then sent to the worker, which pipelines them to the coordinator.

5. voteToHalt():
   A vertex calls this function to declare itself inactive for the next superstep.
If all vertices declare themselves as inactive, no more iterations are
performed.

6. compute(vector<?> messages):
   This is the function with the core logic for the graph algorithm that is
being run. We discussed this in detail in the previous section.

3.4 Use case: SGD with vertex-centric model
As described in Chapter 2, we use SGD as an example graph algorithm
throughout this thesis. In context of collaborative filtering, SGD aims to minimize
the error between the known rating and predicted rating for a user-item pair. We
discuss here how we represent SGD in a vertex-centric model, as used in our
system.
virtual void compute(vector<VectorXd>& messages, PartitionWriter& outputWriter) {
    if (ITERATION < MAX_SUPERSTEPS) {
        VectorXd selfLatentVector = getVertexValue();
        vector<vfloat>::iterator edgeval_it = edgevalues.begin();
        Eigen::VectorXd newLatentVector = selfLatentVector;
        for (vector<VectorXd>::iterator it = messages.begin();
            it != messages.end(); ++it) {
            vfloat known_rating = *edgeval_it;
            VectorXd msgLatentVector = *it;
            double predictedRating = selfLatentVector.dot(msgLatentVector);
            double known_rating = std::max((double)1, known_rating);
            predictedRating = std::min((double)5, predictedRating);
            double error = known_rating - predictedRating;
            edgeval_it++;
        }
        modifyVertexValue(newLatentVector, outputWriter);
        vector<vint>::iterator it2 = edges.begin();
        while (it2 != edges.end()) {
            sendMessage(*it2, new_value);
            it2++;
        }
    }
}

Figure 7 Code snippet for SGD for vertex-centric model

We used both Stylianou's thesis on Graphlib[25], and GraphLab's implementation of SGD as a starting point. Vertices in the graph represent users and items and the vertex values are the latent vectors associated with them. There is an edge for each item rated by a user, with the edge value ranging on a scale of 1.0 to 5.0. For SGD compute function, a message represents a latent vector. Thus, the input messages are a list of latent vectors of neighboring vertices, e.g. in case of a user vertex, it will be an array of latent vectors of all items that have been rated by the user. The algorithm iterates over this list and accumulates the delta changes from each neighbor, as described in Chapter 2. The vertex's value is adjusted using the final delta value and the new vector is sent out as a message to
all its outgoing edges, which in the case of a user vertex, will be all the items that the user had rated.

Both the Graphlib and GraphLab implementations of SGD use alternative super-steps to train the users and items latent vectors, i.e., even-numbered super-steps train user nodes while odd-numbered super-steps train items. Thus, the implementations use one-directional edges and reverse the direction every alternate superstep. In contrast, we performed the updates simultaneously, in the same iteration, by making a copy of each edge, such that for each rating, there was an edge going from a user to an item and from the item that was rated to the user who rated it. Hence, we doubled the number of edges and increased input size but cut down the number of iterations to half. This works well with our implementation, as the column-oriented Vertica is scalable and thus can perform extremely fast sequential reads for large tables.

Figure 7 presents the code used for SGD’s compute function. As is clear from the figure, the user does not have to worry about the underlying architecture of the system, and can easily express the algorithm in a similar way as with Pregel or GraphLab.

3.5 Advantages and Limitations

The disk-based approach provides a simple solution that combines the vertex-centric model with the relational engine. This was the first step for us to prove
that it is possible to make graph queries easier on relational databases. The implementation works for input tables of all sizes, whether or not they fit in memory, and for all synchronous algorithms that can be expressed with any other vertex-centric approach like GraphLab and Pregel. The disk-based approach performs better than Pregel, taking about half the time for running SGD. However, as we show in Chapter 6, it is 9 times slower than shared-memory systems like GraphLab, since we use table reads and writes from disk in each iteration.

On the other hand, the disk-based implementation provides several useful features. Since the intermediate output is written to a relational table at the end of each superstep, it gives an opportunity to run any SQL operations in between. For example, if one wants to filter out data using certain attribute or calculate some statistical measure like maximum or average after each superstep, one can easily write some SQL queries for the intermediate table. Similarly, the implementation allows to interleave different graph algorithms with each other. Thus, if one wants to run shortest path after every iteration of PageRank, both algorithms can be easily combined in one computation. The fact that the intermediate results are stored in relational tables also provides check-pointing, requiring us to redo only some of the iterations in case of failure.
Chapter 4: Main-memory based, Vertex-Centric Query Interface for Relational Databases

In this chapter, we describe how we emulate the vertex-centric computation model using shared-memory approach with relational database system.

4.1 Motivation

As noted in Chapter 3, the disk-based implementation has performance limitations. In this section, we show how using shared memory to buffer results between supersteps can improve performance when graphs fit in memory. The main-memory technique has similar logical query plan as the disk-based approach, however with a different physical implementation.

The major problem with the disk-based approach is that it does not pipeline results efficiently between super-steps. The messages, nodes and edges are not kept in a shared memory space among the workers and are instead provided as input by the coordinator at the start of each super-step. For this reason, we used external looping to run the super-steps instead of doing the iterations inside the UDF/workers. This resulted in repetitive table scans. Interestingly enough, most of the nodes and edges data e.g. group IDs and node IDs are copied over from one super-step to another without any changes. This raises the question if there is a better way for pipelining each iteration's output and avoiding unnecessary table scans.
For synchronous algorithms, graph processing systems like Giraph and GraphLab allow workers to communicate the updates with each other, instead of sending all of them to the coordinator, as done in our disk-based solution. In contrast, as a result of the inability for UDFs to communicate in our disk-based approach, the coordinator in our disk-based implementation writes out the results to intermediate relational tables, which is costly. While writing out the results to tables provides fault-tolerance and allows for interleaving with other kinds of SQL operations, inter-worker communication can give better performance for graphs that can fit in memory.

There are some interesting tradeoffs that can occur if there is some shared block of memory to store the initial input data, which can be used by the workers for communicating updates. In general, if we have smaller sized graphs and some persistence across iterations, it might be better to use a main-memory based technique, which is the motivation to explore this approach.

This chapter gives an overview of the architecture of this main-memory approach. We explain the logical query plan in detail and show how the different components interact with each other. We also discuss the advantages and limitations of this approach.
4.2 Architecture of the Main-Memory Approach

This section describes how we implemented the main memory approach to provide vertex-centric querying in Vertica. In contrast to the disk-based approach, there is a block of memory shared by all workers that allows for more efficient pipelining and better performance for graphs that can fit in memory. The basic architectural plan is very similar to the disk-based approach with the same four major components: data storage, a coordinator, workers, and user-defined functions. However, the tasks performed by each of these components are different to leverage shared-memory. In the rest of this section, we discuss the role of each of the components and describe how they interact with each other.

4.2.1 Data Storage

Unlike disk-based approach, the main-memory system does not maintain a table for storing intermediate output and instead introduces some additional customized data structures. However, the schema for the initial input tables is exactly the same as in the disk-based approach. After the input tables are scanned once, the nodes, edges and messages reside in memory as C++ data structures that can be customized by the user, depending on what information needs to be communicated to the workers.

For most algorithms, including SGD, the messages data structure keeps track of source and destination unique identifier, message value, destination edge group
as well as the edge value associated with the edge between source and destination as shown in Figure 8. In disk-based approach, we didn’t need to keep a column for edge values in the intermediate messages table because the incoming edges data was joined with the new messages table and was passed to the workers in every superstep. However, in this approach, the entire edge data is read only once initially. In the specific case of the SGD computation, the messages keep track of the known rating between the source and destination of the message.

The messages are sequentially stored in message stores, which are C++ data structures (see Figure 9), that also keep track of the number of messages read and written and the read and write offset pointers. Each message store is divided into two parts, where one half stores the messages received from the previous superstep while the second half is used to write new messages that will be used in the next super-step. Each message store can hold twice the maximum number of messages expected to be sent or received by the worker, where the maximum is currently hard-coded for proof-of-concept purposes. Each worker has an array of message stores of size equal to the number of workers stored in its memory. The memory interactions between memory blocks on different nodes in a superstep is illustrated in Figure 10.
struct message{
    vint dst_id;
    vint src_id;
    VectorXd value;
    vfloat dst_edge_value;
    vint dst_edge_grp;
};

Figure 8 Message Struct for In-memory Architecture

struct message_store{
    struct message *messages;
    int *roff, *woff;
    int *num_rmessages,
        *num_wmessages;
};

Figure 9 Message Store Struct for In-Memory Approach

Figure 10 Interactions between workers during a super-step. Worker 3 sends a message for node 15, which resides in worker 1's node array. Thus the message is written to the message store for w3 in w1's set of message stores. Message store also exists for local messages e.g. w2 writing message for node 2. The write pointers are incremented after the message is written and the read pointers are incremented as compute processes the input messages.
4.2.2 Coordinator

The coordinator's tasks can be divided into three phases: setup, super-step management and finalization. The setup phase starts with scanning the input nodes and edges table, sorting them by node ID and partitioning the data amongst the workers. On a single-node system, a thread is spawned for each worker and a partition is assigned to it, as in the disk-based approach. The coordinator initializes a worker by allocating an array of message stores, such that each worker has a message store for itself as well as one for each of the other workers. It also records the memory address of the worker's message store, this serves as a unique identifier to locate the worker in the message store array. After synchronizing the setup for all the workers, i.e. once the coordinator hears back from all workers, it exits the setup phase and starts off the workers to proceed with the execution of super-steps.

Once the setup is done, in the super-step management phase, the coordinator is responsible for coordinating the communication between the workers during the super-steps. Unlike the disk-based approach, there is no external looping and all message updates are stored in the message stores that reside in memory. The coordinator imposes a two-step synchronization barrier at the end of each super-step. The first one is to ensure that all workers are done writing their messages to the relevant message store. Once that is done, the coordinator synchronizes the swapping of newly written messages with the messages received from the previous step, for each of the message stores in the worker's memory. If the synchronization succeeds, the coordinator allows the workers to move to the next
super-step. As in the disk-based approach, the algorithm runs until there are no more messages, which can be either after the computation converges or when the user-defined maximum number of super-steps is reached.

In the final stage, the coordinator collects all the node and edge updates that were made by each worker to its assigned partition and writes them into a relational table, with the schema that can be defined by the user. For SGD, the output table is essentially an updated vertex table.

4.2.3 Worker

As in the disk-based approach, the system contains a number of workers, each of which assists in the execution of the graph computation. Each worker is assigned a partition of nodes, its outgoing edges and initial incoming messages from the graph and is responsible for carrying out the computation on its partition. The workers read the partition data and store the nodes and edges in a contiguous block of memory. Since the coordinator already sorts the initial input data based on node IDs, the workers can directly parse all the data related to a vertex and call the compute function on its incoming messages in the first superstep, similar to the disk-based approach. However, for the rest of the super-steps, the worker is responsible for also sorting all the incoming messages in its message stores before parsing the information and beginning the computations.
Unlike the disk-based approach, where the workers read and write updates to the coordinator, a worker in the shared memory approach own a set of message stores. There is a message store associated with each of the workers that are present in the system, including itself. The first half of a message store for worker A in worker B's set of message stores contains incoming messages from worker A.
for B in the current superstep, while the second half contains messages received in the previous superstep. Computations are performed on the messages received in the second half, i.e. from previous superstep. The coordinator swaps the two halves at the end of each superstep as discussed above.

In order to perform the computations, a worker sequentially scans through each of the message stores, collecting all the messages sent to the current node by all the workers. The nodes are processed in the order of their node IDs. As the computation is completed, the worker writes outgoing messages to its message store in the destination workers. For example, if worker 1 wants to send a message to worker 3, it will write the message to the write half of worker 1's message store in worker 3's message stores. The receiving worker's unique identifier or group can be either specified by the user as a hash function or can be part of the message, depending on the partitioning technique used by the coordinator. For SGD, we used METIS partitioning and hence stored the worker partition ID in the message data structure. The memory block associated with message stores and the process of sending and receiving messages is illustrated more clearly in Figure 10.

After each superstep, workers notify the coordinator and proceed to the next superstep only after the coordinator has completed the two-step synchronization process. Once the worker is done with all the supersteps, it passes all its final updates to the coordinator. Figure 11 gives a general architectural overview of the main-memory approach.
4.2.4 Update/Compute function

In the main memory implementation, the compute function is just as described in Section 3.2.4, except that the input to the function is a list of message data structures, instead of just the message identifiers in the messages table.

4.3 Query Plan

The logical query plan for the main-memory approach is shown in Figure 12. Initially, we take the union of the nodes, edges and messages table, which is sorted and partitioned by the Vertica UDTF. Note that we do not join the input nodes and edges tables with the messages table, unlike disk-based approach, since initially all vertices are active and thus we do not need to filter out the inactive ones. However, we do check for inactive nodes inside the UDF loops at the beginning of a superstep. All iterations are done internally in the UDF, with the output from each superstep stored directly into the messages stores. The updates are sorted and fed into the UDF until either the maximum number of supersteps is reached or there are no more active vertices. The output is stored in a table with same schema as the nodes table, with all the updated values.
4.4 Pregel-like API

The API for main-memory approach extends the disk-based approach's API. We list the modifications and additional functions below.

1. `getVertexId()`
   
   Returns the node ID of the current vertex that is being processed.

2. `getVertexValue()`

   Returns the vertex value of the current node that is being processed.
3. `sendMessage(dst_id, message)`

The function differs from that of the disk-based approach. This method directly writes the message to the message store of the worker that `dst_id` belongs to and increments the write pointer for the message store.

4. `getPartitionId(id)`

This optional function takes as input a node ID (`id`) and retrieves the partition or worker ID of the worker that the node belongs to. For a graph that is hash partitioned by the coordinator, this method is basically the hash function. The worker ID is used in the `sendMessage` function to locate the set of message stores of the worker to which the message's destination node belongs. For SGD, instead of using this function, we carried over the worker ID in the message data structure, as we pre-calculated the partition IDs using METIS.

4.5 Advantages and Limitations

Our main-memory implementation of a vertex-centric query interface on Vertica provides an efficient solution for graphs that can fit in memory. Fast data access is very important for graph analytics and since the data resides in memory, the number of I/Os is reduced significantly, making all the updates and accesses very efficient. Also, since the input and output tables reside in relational tables, one can perform transactional queries to process data before or after running a graph algorithm. The performance of the system is slightly faster than GraphLab and much better than Giraph for a sample of Netflix data, as presented in Chapter 6.
Also, the implementation allows us to control more parameters for optimization, which can be particularly useful when we come up with a query optimizer in the future.

Our current implementation involves only one machine and utilizes the different cores to assign a thread to each worker. While this implementation works well for small graphs, many real-world graphs are large and cannot fit in memory. This will require using multiple machines and managing communication for nodes that reside in physically different locations, not sharing the same memory space. Also, unlike the disk-based approach, the main-memory solution runs iterations within the UDTF, without storing results on disk. This makes the system less fault-tolerant, as there is no check pointing between super-steps. Thus, if the system fails in between super-steps, there is no recovery process and once will need to start the algorithm all over again. Not storing intermediate results in tables also make it hard to run other graph algorithms in between super-steps, which is not the case in disk-based approach.

Thus, the implementation is faster than disk-based for small graphs, but has some limitations associated with it. The proper choice of implementation depends on a number of factors including the workload and whether or not we want to interleave different graph algorithms.
Chapter 5: Optimizations

We applied a number of optimizations to obtain performance gains for both disk-based and main-memory approaches. This chapter gives an overview of the ways in which we optimized the different components of the two implementations. Since there is some overlap in the basic architecture of the two approaches, we start with a discussion of some techniques that were used generally for both the implementations. Later sections delve into the more implementation-specific optimizations for disk-based and main-memory solutions. The optimizations can generally be categorized into physical, logical, system-level and algorithmic optimizations. The performance gains from these optimizations are detailed in Chapter 6.

5.1 General Optimizations

When designing the architecture for the disk-based approach, we used several optimizations to make the system more efficient. Most of these optimizations also applied to the main-memory approach. This section discusses some of these major factors that had a significant effect on the performance of the two systems.

5.1.1 Physical Optimizations

These optimizations are lower level and involve decisions taken when carrying out operations like distributing data across workers, ways of accessing data or the
technique used to store the input tables. This section discusses the decisions made for the physical design of the implementations.

5.1.1.1 Load balancing / Partitioning

All workers run in parallel, which makes load balancing important, to avoid a worker becoming the bottleneck during a super-step. Otherwise none of the other workers can proceed due to the synchronization barrier. Thus, the choice of partitioning for input data can prove to be one of the most important optimizations. We hashed on node IDs to perform partitioning for algorithms like PageRank and shortest path, which worked quite well for them. However, we obtained poor performance results for SGD, which made us to reconsider our partitioning technique.

In SGD, if the error tolerance level is not too high, which is usually the case, vertices send messages on most or all of their edges and process messages from most or all their neighbors. Moreover, the number of these outgoing messages stays nearly constant throughout all super-steps, making the computations edge-heavy. Moreover, for SGD, the computations per node are more complex than for the other algorithms. Thus, if a worker gets nodes with fewer edges, it will complete faster than a worker that gets the same number of nodes but with more edges. This implies that a partitioning technique will be needed that performs edge cuts. We used k-way partitioning with METIS [26][24], a standard, well-researched package of partitioning methods and libraries. METIS performed the
edge cuts such that each worker was assigned almost the same number of edges, thus ensuring a balanced workload among all workers.

Since our implementation runs a worker per core and runs workers in parallel, our balanced partitioning technique allows us to evenly distribute the work keeping all workers busy throughout their computation.

5.1.2 Logical Optimizations

This section presents the logical optimizations made to the query plan for the disk-based approach. We adopted several optimizations to make the pipelining of input data from coordinator to workers more efficient.

5.1.2.1 Unions of input tables instead of using joins

To perform vertex computations for SGD and in general, the system needs information about the node on which the computation is being performed, its incoming messages from other nodes and the outgoing edges. Passing this input data to the workers means combining the data from the nodes, edges and messages table. One natural way to achieve that is to perform a three-way join, however joins are extremely expensive, especially when we are dealing with large graphs.

We realized that instead of joining the tables, we could take a union of the three tables and sort the data on node id before passing it to the workers. The worker
reads the input data sequentially and parses and collects all the information related to a node, before calling the compute function. Once the computation is completed and output is written back, the worker moves on to the next node. Since unions are much cheaper than joins, the performance gains are significant.

5.1.2.2 Sorting the input before feeding it to the workers

By sorting the data on node IDs, the coordinator groups the data for each node together before feeding a partition to a worker. For SGD, we also require information about the edge values corresponding to the source of the message. Since the edges are bidirectional, after sorting the data on dst_id, we order it on dst_edge_id and src_id, which ensures that the list of edges and messages for the node correspond to the same neighboring node. The workers are then fed this sorted partition of the union of nodes, edges and messages tables. Grouping data for each node allows the worker to sequentially read the input entries, which makes the process faster.

5.1.3 System-level Optimizations

The choice of Vertica naturally provides us with some system-level optimizations. Being a column-store, Vertica offers superior data compression, making loading and storing tables for large graphs much efficient. We used Vertica's C++ library for UDTFs, which is optimally written to support UDFs on top of the relational tables. Vertica also allows using g++ compiler level optimizations, in particular
the -o3 flag when compiling the UDTF. We enabled the flag for our system to further optimize the SGD computation and observed significant performance gains.

5.1.4 Algorithmic Optimizations

Unlike in simpler algorithms like PageRank or shortest path, the choice of data representation in SGD for messages and node values is important, as they hold latent vectors instead of primitive data types like floats or integers. Our first attempt was to use comma-separated strings to represent the vectors. Strings could be stored easily in a column. However, processing strings for computations – which included parsing the string, extracting vector entries to floats and then computing dot products – is an expensive process. This affected the performance negatively, making the compute function time-consuming. Hence, we switched to storing each vector entry as a float column and users can define the size of vector. Although, this increases the number of columns, the size is normally not too large (around 20 or so) and Vertica is efficient in reading only the columns that are needed. Once the worker reads these float values, it uses C++ eigen library to represent the vectors, which is optimized to carry out vector computations.

5.2 Optimizations for the Disk-based Approach

Besides the optimizations discussed above, we developed several other optimizations to further improve the disk-based implementation.
5.2.1 Physical Optimizations

We implemented one physical optimization, which we call *replacing instead of updating tables*.

Most of the graph algorithms and certainly SGD result in many node and message value updates per superstep. This can slow down the performance since table updates and deletes are expensive in Vertica, which is designed for query-intensive operations instead. In order to avoid updating tables, we create a new nodes and messages table using the intermediate output table generated by the workers. The new vertex table is created by left-joining the intermediate output table, which includes both vertex and message updates, with the old vertex table.

![Figure 13 Simpler Query Plan for Disk-based Approach with Union instead of Union of Joins.](image)

The old tables are dropped at the end of each super-step and are replaced by
these new tables. Dropping and creating new tables resulted in faster super-steps than by updating the existing tables.

5.2.2 Logical Optimizations

We also developed one logical optimization, which we call simple union versus union of joins. This is an SGD-specific optimization.

While the optimization for unions of joins versus joins (Section 5.1.2.1) apply generally for all algorithms, we noticed that for SGD we can further optimize the union. Normally, we filter out the entries from the nodes and edges tables for which messages have been sent. This requires finding the distinct node IDs that exist in the messages table and join the nodes and edges table with messages table before taking the union. However, we realized that if for an algorithm almost all nodes are expected to receive messages in each superstep, which is true with zero error tolerance level for SGD, one can skip the joins and instead take a simple union of the three tables. The query plan is shown in Figure 13.

5.3 Optimizations for Main-memory Approach

There was some overlap in the optimizations used for disk-based and main-memory approaches. However, the main-memory approach differed in most of the physical and logical optimizations because of the difference in the query plan for both the approaches. This section gives an overview of the optimizations
applied to main-memory in addition to the general optimizations discussed in Section 5.1.

5.3.1 Physical Optimizations

We developed five physical optimizations for the shared memory case.

5.3.1.1 Input/Output Size

Since messages do not need to be stored externally in tables and instead reside in memory, the input and output size for the shared-memory approach is decreased. For the input, we basically eliminated the messages table, which in the case of SGD, was the size of the edges table. The reduction in input tables reduces disk I/O, thereby saving the cost of table scans as well as making the union faster. Since the number of users and items can be huge for large graphs like Amazon or Netflix, the omission of messages table reduces input size significantly and brings performance gains. Also, because the results aren’t stored in intermediate tables, this means that the output table can be made more concise. For SGD and most other algorithms, mainly the vertex values are updated as an end result and it makes sense to just output the nodes table.

5.3.1.2 Synchronization

Currently, we use a simple synchronization technique where we wait until all workers have responded to the coordinator i.e. wrote to a file after processing all
nodes in their partitions. The coordinator keeps checking the sync file until there is an entry from each worker. Since this technique requires the coordinator to wait for some duration and check the file again, it made the wait time a tuning parameter. We optimized the synchronization timing for the current implementation by trial and error: too much time added to the overhead and checking too frequently isn't efficient. We found that 1 millisecond was a good wait time. We intend to develop a more adaptive approach for choosing the wait time and also hope to extend the system to do computations asynchronously for certain algorithms.

5.3.1.3 Sequential scans/Message stores

Each message store in a worker's set of message stores contains messages in a contiguous block of memory. This avoids random I/O accesses when the worker scans through each message store to collect messages for a node's computation. Also the messages are sorted on node IDs for each message store. This means that when the worker collects messages for a node, it can sequentially scan through a message store making the process more efficient.

5.3.1.4 In-place updates

During each superstep of SGD, there is a new node value that is adjusted according to the messages from the neighboring vertices. One way to apply these
changes is to delete the old nodes array and replace it with a list of new node values. However, instead of recreating the array, each worker stores a nodes array in memory during the setup phase and keeps track of the current node index in the array. During the computation, it accesses the current node in the array and directly updates the node value. Thus, the overhead of allocating new blocks in memory for the new nodes array and garbage collecting the old one is avoided.

5.3.1.5 Sorting messages
In the disk-based approach, the coordinator sorted input messages outside the UDTF, using Vertica’s query optimizers to decide the sorting mechanisms. However, in the shared memory solution, sorting is done inside the UDTF, which makes the choice of sorting methods one of the optimization parameters. Currently, the system doesn’t use a cost-estimator or query optimizer to choose the optimal query plan. Thus, the sorting method was chosen experimentally by trying different sorting mechanisms. Quick sort and radix sort work well in general, but found that quick sort was slightly better in the case of SGD.

5.3.2 Logical Optimizations
We used two optimizations for the logical query plan for main-memory approach: pipelining results and eliminating messages table in the union of the input tables.
5.3.2.1 Pipelining

Instead of storing the intermediate results externally, the updates are directly applied to the nodes and edges at the end of each superstep. Thus, the cost for input scans and writing intermediate results during each superstep is eliminated, which was one of the time-consuming steps in disk-based approach. Also, unlike the disk-based solution, in shared-memory implementation, workers can directly write the messages to the message store of worker that owns the destination node. Thus, there is faster access and updates and less communication overhead between the coordinator and worker.

5.3.2.2 Union of nodes and edges

The input to the shared-memory implementation only requires nodes and edges table. There is no need to filter for nodes that will receive messages, since the input tables are only scanned once before the iterations begin i.e. all vertices are active then. Thus, we can use a simpler query plan as shown in Figure 14 by taking a union of just the nodes and edges table.
Table 1 summarizes the optimizations used for implementing SGD on both disk-based and main-memory approaches, as discussed in this chapter. We present the results showing performance gains from these optimizations in Chapter 6.
<table>
<thead>
<tr>
<th>Physical</th>
<th>Logical</th>
<th>System-Level</th>
<th>Algorithmic</th>
</tr>
</thead>
<tbody>
<tr>
<td>General (both approaches)</td>
<td>• Load balancing using METIS</td>
<td>• Unions of joins</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Parallel Workers</td>
<td>• Sorting input</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Vertica UDTFs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• -O3 compiler flag</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Strings vs floats</td>
<td></td>
</tr>
<tr>
<td>Disk-based</td>
<td>• Replacing instead of updating tables</td>
<td>• Simple union vs unions of joins</td>
<td></td>
</tr>
<tr>
<td>Main-memory</td>
<td>• Reduced input/output table size</td>
<td>• Pipelining results</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Synchronization wait times</td>
<td>• Union of nodes and edges – no messages table.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Sequential message stores</td>
<td>• In-place updates</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Quick sort</td>
<td>• Quick sort</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6: Experiments and Results

In this chapter, we describe our evaluation of the two approaches: disk-based and shared-memory. The goal of this chapter is to portray how the optimizations as discussed in Chapter 5 were used to improve both the approaches. Before delving into the experiments and analysis, we describe the datasets and test bed on which the experiments were performed. We then start with a comparison of our optimized solutions with GraphLab and Giraph to illustrate how well our vertex-centric approaches for relational engines work in comparison with the graph processing systems. Later sections present the cost analysis of each of the implementations and explain how different optimizations affected the costs breakdown.

6.1 Benchmark Environment

In this section, we describe data and experimental testbed we use for our experiments.

6.1.1 Dataset

We performed collaborative filtering on a subset of Netflix data using stochastic gradient descent as the minimization function. The dataset contains directed edges, where each edge represents rating given by a user to an item. Every user
and item is assigned an ID and the ratings are on the scale of 1-5. The data is in the following format:

```
user_id    item_id    rating
```

The dataset contains 93,705 users and 3,561 items, thus 97,266 nodes in total and 384,3340 edges or ratings. To explore results in the scenario when communication load is maximal, we used SGD with zero error tolerance level. Thus all vertices remained active through all iterations.

We processed Netflix dataset to make the input tables for nodes, edges and messages. The nodes table contained an entry for each unique user and item ID, along with a vertex value and group ID. For experimental purposes, we used an all-ones column vector of size 2 as the initial latent vector for each user and item node. The group ID for each node was assigned using a partitioning function. To create the edges table, we duplicated each edge in the opposite direction such that we had an edge from item to user for each rating. The messages table contained an entry for a message via each edge along with the information for the group ID of the destination node ID.

### 6.1.2 Testbed

Our testbed consists of a single machine with 2.4 GHz Xeon, with 2-socket, 12-cores and 24-threads due to hyperthreading. The testbed has 48 GB memory, 1.4T disk and running on CentOS 6.5. We used Vertica's 6.1 to run the experiments.
6.2 Performance

This section sets the stage for the rest of the chapter. We first show how the optimized, final versions of disk-based and main-memory vertex-centric approaches on Vertica compare with the popular graph processing systems, GraphLab and Giraph. The second half of the section illustrates the overall performance gains we got from applying the optimizations as discussed in Chapter 5 to both the approaches. The later sections provide a more detailed analysis of those optimizations.

6.2.1 Comparison with GraphLab and Giraph

We compared our optimized implementations of disk-based and main-memory approach with Giraph and GraphLab 2.2. The Giraph implementation was run with 3 workers, with each worker assigned memory of 12GB. GraphLab was run with 12 threads i.e. with -ncpus option set to 12. All results were run on the same machine as described above. For both the disk-based and main-memory approach, we assume that the input tables are already loaded in the Vertica database.

Figure 15 shows the results of performing collaborative filtering with SGD on the four systems for 10 super-steps i.e. 10 updates to both user and item nodes. The times showed in the plot include time taken to load the graph in memory and to
run SGD for 10 updates. As shown in the plot, the optimized version of our disk-based approach performs 2.4 times better than Giraph, despite having disk I/O heavy implementation. Our disk-based approach is much slower in comparison with the shared-memory based system, GraphLab, which is because of the expensive disk I/Os per superstep in our disk-based implementation.

Our main-memory approach is 22 times better than Giraph and 1.3 better than GraphLab. The performance gains in comparison with GraphLab shows the advantage of using read optimized relational systems like Vertica as the storage model. The loading time for the graph in GraphLab was 5 seconds compared to 2.3 seconds for the main-memory approach because of fast input table scans in Vertica (see Figure 18 for cost breakdown of main-memory approach). Also, the shared memory approach performs 9.3 times better than the disk-based
approach since all the updates and accesses are from the main memory. Thus, if
the data fits in memory, the results show that relational databases can be utilized
for graph analytics giving comparable or even better performance as other
shared-memory graph processing systems like GraphLab.

6.2.2 Effects of Optimizations

The disk-based approach was our initial implementation to use relational engine
for graph analytics. The first version of our implementation used some logical
and architectural optimizations including parallel workers and joins of unions
instead of joins of messages, nodes and edges and 10 iterations of SGD took 224
seconds to run. The natural next step was to perform a detailed cost analysis of
our implementation, after which we applied several optimizations, bringing an
improvement by a factor of 3.3 as shown in Figure 16.

![Figure 16: Effects of optimization for disk-based and shared memory approach for running 10 supersteps. Note: The first version of main-memory approach already incorporates the general optimizations as applied to disk-based version i.e. METIS partitioning, compiler optimizations and unions of tables.](image-url)
However, one of the major factors that made the disk-based approach less efficient than GraphLab was that it performs disk I/Os for reading input and writing output to tables for each superstep. This motivated us to develop the main-memory approach that saved the graph data in memory and incorporated most of the optimizations applied to disk-based. Figure 16 shows that the first version of main-memory approach is significantly better than the disk-based approach. Since the version of main-memory approach before optimizations already incorporates the general optimizations i.e. it uses METIS partitioning, -03 flag, floats instead of strings and uses unions instead of joins, the performance gains after optimizations indicates the effects of only main-memory specific improvements. Despite beginning with an optimized first version of main-memory approach, we were able to bring further improvement by a factor of 1.3.

The next few sections present details of the cost analysis and show the effects of individual optimizations on disk-based and shared memory approaches.

6.3 Cost Analysis

In this section, we first present the breakdown of a superstep for disk-based and main-memory implementation and then explain the measures that were used to estimate how much time was spent by each step in the iteration. We then show the detailed results from performing this cost breakdown for the two approaches, both before and after all optimizations were applied.
Table 2 Cost breakdown components for disk-based

<table>
<thead>
<tr>
<th>Disk-based Approach (each superstep)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union of input tables + scan</td>
</tr>
<tr>
<td>UDF Compute cost</td>
</tr>
<tr>
<td>UDF Overhead (Partitioning and sorting nodes + invoking UDTFs/workers)</td>
</tr>
<tr>
<td>Writing the intermediate output tables</td>
</tr>
<tr>
<td>Updating the vertex and messages table</td>
</tr>
</tbody>
</table>

Table 3 Cost breakdown components for main-memory

<table>
<thead>
<tr>
<th>Main-memory Approach (total, since iterations are internal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union of input tables + scan (done once)</td>
</tr>
<tr>
<td>UDF I/O (done once)</td>
</tr>
<tr>
<td>Preparing vertices before compute</td>
</tr>
<tr>
<td>Sorting messages</td>
</tr>
<tr>
<td>Compute cost</td>
</tr>
<tr>
<td>Synchronization cost</td>
</tr>
<tr>
<td>Writing output table (done once)</td>
</tr>
</tbody>
</table>

6.3.1 Methodology

Based on the architecture and query plan, as described in Chapter 3 and 4, the total time spent in each iteration can be broken down into several components as listed in Table 2 and Table 3. For both the disk-based and main-memory approach, the cost analysis was performed for running 10 iterations of SGD.

Below we describe how we estimated the time spent by each of the components for both the approaches, mainly using SQL queries and C++ timer library. For each of the experimental queries, we cleared out caches to get a more accurate measurement of time. Also, to reduce effect of outliers, we ran SGD for 10 supersteps and used the sum of costs of the different components.
**Input table scan and union:**

We used Vertica’s timing operator and ran a SQL statement that took maximum for each of the columns in the union of the three tables. Since taking the maximum requires Vertica to read each column, it gives a rough idea of the time spent in taking scanning the union of before it is provided as input to the UDTF.

**UDF I/O, Compute, Synchronization, Sorting and Preparing Vertices:**

The worker uses Vertica’s input reader and output writer to read and write to the tables. We measured the input and output time by using C++ timer library around the code that reads and writes to the I/O pipe. Similarly, for calculating how much time the core SGD compute function took, for each worker, we added the time taken by all the compute calls per instance. Since all workers operate in parallel, we took the maximum of the compute times to account for the worker that acted as the bottleneck.

For main-memory approach, we used similar technique to measure costs of preparing vertices before compute calls, sorting messages (since it’s done internally in the UDF) and synchronization for each superstep. Since all workers run in parallel, we summed up the costs for 10 iterations for each worker and then took the maximum.
Writing output tables:
We used a SQL query that copied the contents of the output table generated by the UDTF and created an identical table to get an estimate of the time to write the output to a table for disk-based approach.

Updating the nodes and messages tables for disk-based:
We used Vertica's timer to time the queries in the terminal manually.

UDF overhead:
The sum of the above costs was deducted from the total amount of time spent by each superstep as the UDF overhead. The overhead thus includes the time spent partitioning the union of the input tables, invoking the worker UDFs and sorting the input. For main-memory, it also includes the time spent by the coordinator in allocating the message stores for each of the workers during the setup phase.

The cost analysis results showed that a major portion of the time was spent in reading and writing to the tables for each superstep as well as in the UDF overhead. Also, the compute function takes a significant amount of time. In order to address all these problems, we applied several optimizations.

6.3.1 Results
This section presents the cost analysis of the disk-based and main-memory approach.
6.3.1.1 Cost Analysis of Disk-based Approach

Figure 17 shows the cost breakdown for disk-based implementation for performing 10 updates of SGD, both for the initial and the more optimized version. The first version incorporates some optimizations discussed in Chapter 5: (1) uses unions of joins instead of joins (joining the tables for messages and edges of 6.5 million entries and nodes table of size 100,000 took more than 10 minutes), (2) parallel workers working in non-fenced mode and (3) replaces nodes and messages table instead of updating. The optimized version is partitioned using METIS, uses simple unions of messages, edges and nodes and also incorporates the algorithmic and system-level optimizations. These improvements helped to decrease the input cost, compute cost and the UDF overhead, which includes time spent partitioning nodes.

Figure 17 Cost breakdown of 10 supersteps in disk-based implementation with and without optimizations.
6.3.1.2 Cost Analysis of Main-Memory Approach

The column charts in Figure 18 show the cost breakdown for the first version of the shared memory compared to the final, optimized version when run for 10 updates of SGD. The first version of the main-memory approach incorporated *all the general optimizations* as discussed in Chapter 5. In particular, the input data was partitioned using METIS, simple unions were used instead of joins and system-level and algorithmic-level optimizations were incorporated in the first version. The optimized version performs implementation-specific improvements: optimized synchronization wait timings, better sorting mechanism and reduced input size by eliminating messages table in the input tables union. The major improvement was the reduced input size that minimized table scanning costs.

![Cost breakdown of 10 updates of SGD with Main-memory Approach before and after optimizations](image)

*Figure 18 Cost breakdown of 10 updates of SGD with Main-memory Approach before and after optimizations (10 iterations without optimization and 11 iterations with optimizations).*
6.4 General Optimizations Applied to Disk-based Approach

This section walks through the order in which the major optimizations were applied to portray more clearly how each of them affected the performance of the system.

These optimizations apply generally i.e. to both disk-based and shared memory implementations. Since, disk-based approach was our first implementation, the experiments shown below use disk-based version of SGD for comparison. We incorporated these optimizations in our first version of main-memory. Since all supersteps in the disk-based approach are the same, the results below show improvements for one superstep of SGD.

6.4.1 Better Load Balancing

Hash partitioning works well in general if the workload differs in each superstep. However, in case of SGD with zero error tolerance level, all nodes send and receive messages to all its neighbors at each superstep, thus the workload remains constant. When we partitioned the nodes by taking the hash of their node IDs, all workers received the same number of nodes, but not the same number of edges. This meant that some workers had nodes with more edges than others and hence did more computations per superstep, which slowed them down. The node and edges distribution using hash partitioning is shown in Figure 19. By experimentation, we found 200 partitions as the ideal number for hash partitioning.
In order to achieve better load balancing, we used METIS partitioning technique with k-way partitioning mechanism. This technique partitioned the data based on the number of edges for each node, thus making the edge distribution even for all workers. This meant that all nodes processed approximately same number of messages (see Figure 20). For METIS, 12 partitions worked ideally. The partitioning method decreased the superstep cost by almost 28% as shown in Figure 21.

Figure 19 Workload With Hash Partitioning

Figure 20 Workload with METIS partitioning
6.4.2 String vs Floats for representing latent vectors

Using floats instead of strings decreased the cost of a superstep by slightly as shown in Figure 22. This is because string parsing is expensive and computations are faster if the latent vector values are pipelined without converting them from floats to strings. However, more than the performance gains, we chose to use strings instead of floats because of the limitation of specifying octet values to represent strings. We had to pre-specify how many characters each latent vector value can fit in, which were a hindrance for float values with many decimal places and resulted in truncation of the string representation in some cases.
6.4.3 Simplifying union of input tables

As can be noted from the cost breakdown for un-optimized version of disk-based implementation, most of the time is spent in scanning the input tables and writing out the intermediate output tables. We could not decrease the cost of intermediate output tables which was a limitation of the architecture of disk-based approach. However, for SGD zero error tolerance level, we were able to directly take the union of nodes, messages and edges tables instead of first joining nodes and edges tables with messages to filter out nodes with incoming messages. Thus, if all or most of the nodes remained active, we can avoid performing expensive joins. The optimization improved performance by a factor of 1.6 times and the cost difference is shown in Figure 23.
6.4.4 Compiler 03 optimizations

Vertica allows us to use compiler level optimizations when compiling the UDTFs. Since our machine had enough memory to adjust to any increase in code sizes, we enabled the g++ 03 optimization that performs function in-lining and tries to vectorize loops among other compiler optimizations. Using 03 made each superstep faster by almost 3.5 seconds as shown in Figure 24.
6.5 Main-memory Approach Specific Optimizations

We started off with a shared-memory implementation incorporating all general optimizations as discussed in Chapter 5. Each worker was initially assigned memory of 90 MB and we were able to increase the memory to 100MB by reducing the input size (as discussed below). All the experiments below were run in fenced mode of Vertica, as otherwise we cannot specify the amount of memory assigned to each instance of the UDTF.

Unlike the disk-based approach, where all super-steps are essentially the same, the first and last superstep of main-memory approach are more I/O intensive than the rest of the supersteps. The external UDF costs, i.e. the costs of scanning input tables, partitioning them across the workers and writing the final output to a relational table, are incurred only once. To capture the performance gains that
occur once the data is read into memory, for each of the experiments below, we computed the costs for 10 updates of SGD.

### 6.5.1 Synchronization wait time

The coordinator synchronizes each super-step by waiting until all workers finish writing their outgoing messages. We experimented by varying the time coordinator waits for until it pings the worker again. Initially, we started with 50 milliseconds, but noticed that the workers were ready in the earlier half of the duration. We found that 1 millisecond of wait time was neither too small nor too large a duration for the coordinator to check with the worker. This helped us reduce the synchronization cost by 34%, as shown in Figure 25 from approximately 1760 ms to 1160 ms.

![Varying synchronization wait timings](image)

**Figure 25** Change in synchronization cost with reduced wait time for 10 iterations
6.5.2 Radix vs Quick sort

We tried various sorting mechanisms for sorting the messages before they are sent to the workers. For most algorithms, including PageRank and SSSP, radix sort performed significantly better than quick sort. However, in case of SGD, we found that the performance was comparable, with quick sort performing slightly better – a difference of around 800 ms for 10 computations of SGD (11 iterations) as shown in Figure 26.

![Effect of sorting mechanisms for sorting messages](image)

**Figure 26** Change in sorting costs by using quick sort instead of radix sort for 10 iterations.

6.5.3 Reducing input size

We eliminated the messages table and instead used the first iteration for the to propagate node values for all vertices across their neighbors. The elimination of messages table decreased the input size of tables to scan, take union of and sort by 6.5 million entries with the tradeoff of an extra iteration. Since the extra
iteration cost on average is 70 ms compared to the decrease in union and scan cost of about 600ms, the overall result was a net gain in performance by almost 16% for 10 updates. Also, reduced input size allowed us to allocate 10 MB more memory per instance/worker.

Figure 27 summarizes the change in superstep timings for 10 updates to each user and item node in the dataset after applying each of the above optimizations.

![Effect of optimizations on shared-memory](image)

**Figure 27** The plot shows the gain in performance as the implementation-specific optimizations were applied incrementally to the main-memory approach. The times shown are for 10 iterations of SGD.
Chapter 7: Ongoing and Future Work

We were successful in developing two different architectures that provide users with a powerful and easy-to-use vertex-centric interface to run iterative graph algorithms on Vertica. There are many interesting opportunities to extend the idea behind the two implementations to make graph analytics further efficient and scalable on relational databases. This chapter discusses the next steps in building a more robust and flexible system for graph processing on a relational engine.

7.1 Extending shared memory

As the results show, the shared memory approach works almost 9 times better than the disk-based approach, but doesn't provide fault tolerance and scalability. It will be useful to introduce check-pointing after every few super-steps to provide better recovery for the system. Currently, the approach is designed to work on a single machine and utilizes threads and cores for workers. In the short term, we hope to extend the shared-memory idea to use a cluster of machines, which will allow us to scale to larger graphs that cannot fit in memory, which is one of the major negatives of the current implementation.
7.2 In-database graph analytics

The two approaches for implementing vertex-centric query interface on Vertica and various possible optimization techniques for each of them imply that there can be multiple ways for solving graph algorithms on relational databases. However, the above implementations are based on a mostly static plan. Rather than experimentally deciding between which techniques to use for a workload, it will be more useful if the system is dynamic and can automatically adapt query plans according to the datasets and algorithms.

Xie et al. [27] proposes a similar approach for automating decisions for queries but they only focus on choosing the optimal mode for synchronization. Their work shows that we can use the properties of algorithms, like convergence, resource utilization and communication between nodes, to generalize if synchronous or asynchronous mode will be optimal. They propose MSync, an execution mode for PowerGraph that allows the system to decide between running an algorithm in asynchronous or synchronous mode or alternate between the two during super-steps. They use statistical data about memory accesses and usage from previous iterations to predict which mode will give better performance benefits. Similarly, Grace [15] exposes API that allows users to choose how vertices will be scheduled and which messages will be prioritized, thus giving more flexibility. However, it does not make these decisions dynamically and instead leaves it to the user to define the policies.
Currently, we are working towards making UDF-based graph analytics a part of a relational database, which can allow us to express the graph analysis as a query plan. A lot of query optimizations and execution decisions depend on the properties of the algorithm and the workload. We intend to incorporate new graph-specific operators to a relational database and use cost estimators that can choose the optimal query plan based on the dataset and algorithm. The operators can serve as an abstraction to the user, while the query optimizer will choose the best execution plan.

There are a number of optimization knobs that can be tuned for the vertex-centric approach as we discussed in the earlier chapters. The operators can take care of performing those optimizations automatically and can thus be categorized into two types: data related or algorithm specific. Data related operators can include choosing different techniques for partitioning data, denormalization of tables, storing intermediate results and choosing caching strategies e.g. pinning certain pages and buffer pool management. Algorithm specific operators, on the other hand, can make optimization decisions for the design of an algorithm e.g. for scheduling vertices synchronously or asynchronously, running the algorithm on either node-level, edge-level or on blocks of data and for converging the algorithm locally or globally.

By allowing for in-database graph analytics, we will be able to better combine graph and relational analysis in the same query plan and make a more complete end-to-end data analytics system.
Chapter 8: Conclusion

Specialized graph analytical systems, like GraphLab and Giraph, are often favored in comparison to relational databases for performing graph queries, mainly because of the complexity associated with using SQL to write iterative queries. In this thesis, we explored the possibility of using relational databases for performing graph analytics. We showed that highly optimized column-oriented databases like Vertica can be used to provide a popular, easy-to-use vertex-centric interface for running graph algorithms. As a proof-of-concept, we developed two solutions: disk-based implementation and main-memory approach. To test the implementations, stochastic gradient descent, a widely used loss-minimization algorithm for machine learning techniques, was used as the representative iterative algorithm.

Both disk-based and shared memory approaches followed a similar logical query plan as Pregel and provided a Pregel-like API on top of Vertica's UDTFs. The disk-based approach used relational tables for input and output of each iteration or super-step. While doing so made the system more fault-tolerant, the table scans were costly and affected performance. For graphs that fit in memory, we proposed the second approach that uses main memory to store and access data. When developing the two solutions, we explored several methods in which the two approaches could be optimized including parallelizing UDF computations, reducing random I/O accesses and pipelining results.
Our experiments and analysis show that the two approaches provide with reasonable performance benefits. The disk-based approach works 2x better than Giraph and the main-memory approach gives comparable performance benefits as GraphLab, which is 20x better than Giraph and 9x better than the disk-based approach. We also analyzed the effect of major optimizations on performance of the two implementations and showed a detailed cost analysis before and after those optimizations were applied.

In the future, we hope to extend our work to make the optimization decisions dynamic, based on the workload and general properties of algorithms e.g. convergence and granularity at which computations will be performed. We intend to implement these optimization knobs as operators in a relational database to support in-database graph analytics and provide an end-to-end system for data analytics.
Appendix A

This code snippet below shows the implementation for the coordinator for the disk-based approach.

while true; do

# run the super step

$VERTICA -d db_usr -w db_pwd -U dbadmin -c
"CREATE TABLE M2 AS
SELECT pregel(t.src,t.dst,...) USING PARAMETERS
iteration=$iteration,grpSize=$grpSize,maxSuperStep=$maxSuperStep,
lambda=$lambda, gamma=$gamma)
OVER (PARTITION BY t.dstgrp ORDER BY t.dst)
FROM (SELECT ... FROM E
UNION ALL
SELECT ... FROM V
UNION ALL
SELECT ... FROM M)
) AS t;"

# any more messages to pass?
ROW_COUNT='"$VERTICA -A -t -d db_usr -w db_pwd -U dbadmin -c "SELECT
count(*) FROM M2;"'

# if no more messages to process, stop iterations
if [ "$ROW_COUNT" == "0" ]; then
  $VERTICA -q -d db_usr -w db_pwd -U dbadmin -c "DROP TABLE M2;"
  break
fi

# update vertex table V by "replace instead of update" technique.

$VERTICA -d db_usr -w db_pwd -U dbadmin -c
"CREATE TABLE V2 AS SELECT
V.dst,
isnull(tmp.dst_state,V.dst_state) AS dst_state,
V.dst_grp,
isnull(tmp.dstvalue_vl, V.dstvalue_vl) AS dstvalue_vl,
isnull(tmp.dstvaluev2, V.dstvaluev2) AS dstvaluev2,
FROM V LEFT JOIN
(SELECT distinct(src),dst_state,dstvalue_vl, dstvaluev2 FROM M2)
AS tmp
ON V.dst=tmp.src;
DROP TABLE M;
ALTER TABLE M2 RENAME TO M;
DROP TABLE V CASCADE;
ALTER TABLE V2 RENAME TO V;"

# increment iteration count
iteration=$((iteration+i))
done
Appendix B

Below is an example code snippet for `processPartition` which implements the logic for the workers.

```cpp
virtual void processPartition(ServerInterface &srvInterface,
PartitionReader &inputReader, PartitionWriter &outputWriter){
try {
    // make sure that the number of input columns match the transform
    // function's input
    if (inputReader.getNumCols() != 11)
        vt_report_error(0, "Function only accepts 11 argument, but %zu provided", inputReader.getNumCols());

    // read input parameters
    ParamReader paramReader = srvInterface.getParamReader();
    const vint iteration = paramReader.getIntRef("iteration");
    const vint grpSize = paramReader.getIntRef("grpSize");
    const vint maxSuperSteps = paramReader.getIntRef("maxSuperSteps");
    const vfloat lambda = paramReader.getFloatRef("lambda");
    const vfloat gamma = paramReader.getFloatRef("gamma");
    ITERATION = iteration;
    GRP_SIZE = grpSize;
    MAX_SUPERSTEPS = maxSuperSteps;
    LAMBDA = (float)lambda;
    GAMMA = (float)gamma;
    vector_size = 2;
    value.resize(vector_size);
    new_value.resize(vector_size);

do{
    // read input from the input tables
    const vint src = inputReader.getIntRef(0);
    const vint dst = inputReader.getIntRef(1);
    const vint dst_edge = inputReader.getIntRef(2);
    const vint dst_state = inputReader.getIntRef(3);
    const vint dst_grp = inputReader.getIntRef(4);
    const vint dst_edge_grp = inputReader.getIntRef(5);
    const vfloat dst_edge_value = inputReader.getFloatRef(6);
    const vfloat vl = inputReader.getFloatRef(7);
    const vfloat v2 = inputReader.getFloatRef(8);
    const vfloat msg1 = inputReader.getFloatRef(9);
    const vfloat msg2 = inputReader.getFloatRef(10);
    if(id!=-1 && id!=dst){
        doCompute(outputWriter); // need to compute the previous vertex
        id = dst;
    }

    // set vertex value
    if(src==vint_null && dst_edge==vint_null){
        num_vertices++;=1;
        value(0) = vl;
```

value(1) = v2;

// collect 'message'
if(dst_edge==vint_null && dst_state==vint_null) {
    VectorXd msg(vector_size);
    msg(0) = msg1;
    msg(1) = msg2;
    messages.push_back(msg);
}

// collect all 'edge' for any one 'from' node
if(src==vint_null && dst_state==vint_null) {
    num_outgoing_msgs += 1;
    edges.push_back(dst_edge);
    edge_values.push_back(dst_edge_value);
    edge_grps.push_back(dst_edge_grp);
}

} while(inputReader.next());

doCompute(outputWriter); // need to compute the last vertex

} catch(exception& e) {
    // Standard exception. Quit.
    vt_report_error(0, "Exception while processing partition: [%s]", e.what());
}

}
References


