Complex Graph Algorithms using Relational Database

by

Aly Ahmed

MSC. (Computer Science), Nottingham University, 2008

A Dissertation Submitted in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Complex Graph Algorithms using Relational Database

by

Aly Ahmed

MSC. (Computer Science), Nottingham University, 2008

Supervisory Committee

Dr. Alex Thomo, Supervisor

(Department of Computer Science)

Dr. Venkatesh Srinivasan, Departmental Member

(Department of Computer Science)

Dr. Issa Traore, Outside Member

(Department of Electrical and Computer Engineering)

ABSTRACT

Data processing for Big Data plays a vital role for decision-makers in organizations and government, enhances the user experience, and provides quality results in prediction analysis. However, many modern data processing solutions make a significant investment in hardware and maintenance costs, such as Hadoop and Spark, often neglecting the well established and widely used relational database management systems (RDBMS's).

In this dissertation, we study three fundamental graph problems in RDBMS. The first problem we tackle is computing shortest paths (SP) from a source to a target in large network graphs. We explore SQL based solutions and leverage the intelligent scheduling that a RDBMS performs when executing set-at-a-time expansions of graph vertices, which is in contrast to vertex-at-a-time expansions in classical SP algorithms. Our algorithms perform orders of magnitude faster than baselines and outperform counterparts in native graph databases.

Second, we studied the PageRank problem which is vital in Google Search and social network analysis to determine how to sort search results and identify important nodes in a graph. PageRank is an iterative algorithm which imposes challenges when implementing it over large graphs. We study computing PageRank using RDBMS for very large graphs using a consumer-grade machine and compare the results to a dedicated graph database. We show that our RDBMS solution is able to process graphs of more than a billion edges in few minutes, whereas native graph databases fail to handle graphs of much smaller sizes.

Last, we present a carefully engineered RDBMS solution to the problem of triangle enumeration for very large graphs. We show that RDBMS's are suitable

tools for enumerating billions of triangles in billion-scale networks on a consumer grade machine. Also, we compare our RDBMS solution's performance to a native graph database and show that our RDBMS solution outperforms by orders of magnitude.

Table of Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

"O Allah, benefit me with that which You have taught me, and teach me that which will benefit me, and increase me in knowledge"

Prophet Mohamed

First and foremost, All praise to Allah who blessed me with all kinds of favors which I cannot enumerate.

I want to express my deepest gratitude to my supervisor Dr. Alex Thomo, who has taught and helped me a great deal throughout my Ph.D. Dr. Alex Thomo is a brilliant mentor who always guides me and provides valuable insights through the entire research. I was very fortunate to work with him and benefit from his perspectives.

Also, I would like to thank my wife and kids for their patience and continues support, my mother for her prayers and encouragement, and my brothers and sisters.

DEDICATION

This work is dedicated to my parents, wife and my kids!

Thanks for believing in me.

Chapter 1

Introduction

The amount of data collected from different sources has reached unprecedented levels. According to the survey of McKinsey Global Institute in 2011 [45]: "Companies capture trillions of bytes of information about their customers, suppliers, and operations, and millions of networked sensors are being embedded in the physical world in devices such as mobile phones and automobiles, sensing, creating, and communicating data. Multimedia and individuals with smartphones and on social network sites will continue to fuel exponential growth. Big data–large pools of data that can be captured, communicated, aggregated, stored, and analyzed is now part of every sector and function of the global economy. Like other essential factors of production such as hard assets and human capital, it is increasingly the case that much of modern economic activity, innovation, and growth simply couldn't take place without data."

The data is now being measured in zettabytes, 10^{21} bytes, and petabytes, 10^{15} bytes. For example, Walmart estimates to create 2.5 petabytes of consumer data every hour [46], Facebook processes tens of billions of likes and messages every day, Google receives 1.2 trillion search requests every year, and Internet

of Things (IoT) data is expected to exceed 175 zettabytes by 2025 [42].

The graph is a natural fit to represent connected entities and their relationships as it can capture complex relationships. For example, users, posts, images, and videos are represented as nodes in social networks, and the relationships between them are modeled as edges. The same goes for world wide web where pages are considered as nodes and URL links in the pages as edges, road maps where locations are represented as nodes and roads as edges, and networks of items where the edges can represent consumer behavior similarities or social network trends or specific associations based on similar posts or similar likes and dislikes.

Graph analytics is the process of gaining new insights and patterns discovery from the data. Analyzing the topology and properties of these graphs can be extremely useful in obtaining valuable information. With the current level of data, graph analytics drive business decisions and play a vital role in shaping many commercial goals and even political goals. We have seen the impact of social network analysis in advertising campaigns and targeted political messages in recent elections.

1.1 Motivation

Based on the survey conducted by Sahu et al. [57], the major challenges faced by users of graph analytics are as follows.

- Large Graphs Are Very Common: most graphs used by organizations are huge, over one billion edge, and big graphs not limited to large companies like Google or Facebook but quite common among other companies as

well.

- Scalability: With graphs, size grows exponentially, so scalability is the most challenging issue facing companies where the need for fast processing is also increasing.

- Prevalence of RDBMS: Relational databases still are the most common system for data processing and data retrieval, and RDBMS systems are the backbone for many organization operations and applications.

As graph sizes grow to billions of nodes and edges, many graph algorithms that require loading of the whole graph into memory will not work or become extremely slow due to memory swap, hence to run these algorithms, companies might upgrade their hardware capacity or invest into new distributed systems like Hadoop. However, both alternatives include high initial cost and significant ongoing cost (operation cost). Relational Database Management Systems (RDBMSs) are widely used in most organizations and are the main component for most enterprise applications and day-to-day operations. Most companies are already invested heavily to set up RDBMS infrastructure. The market share of RDBMSs is worth billions of dollars, and it is increasing year over year. Also, the maturity and stability of RDBMS systems are well established, and businesses rely on them for storage and data processing (create, update, delete and query).

Based on the above, the logical conclusion is to run graph algorithms on RDBMS and get the benefits of the existing infrastructure and the high performance in terms of indexing, data buffering, storage, and query optimization. However, RDBMSs are often viewed by businesses as inadequate for graph

processing and therefore pushed aside in favor of alternative technologies such as vertex-centric systems like Pregel, Giraph, and graph databases like Neo4J, which we try to address in this dissertation as not entirely justified and answer the question: Can we use an RDBMS as a single platform that supports graph analysis?

In this dissertation, we propose algorithms for three fundamental graph analytics problems. First, we address *Source-to-Target Shortest Path Problem*, which is one of the core graph problems found in many applications. For example, it is used to identify how people are connected in social networks, find driving directions from location A to location B, and provide the backbone of many graph analysis tasks. Shortest path algorithms can also be adapted to compute other tasks, such as minimal spanning trees. Second, we address *Page Rank*, which is widely used by Google and other search engines to sort out search results. Third, we address *Triangle Enumeration* which is vital for many applications like performing community searches and detecting fake accounts, malicious pages, or instances of web spamming.

Our results show that graph analytics is possible with relational databases and could perform better than graph databases and process massive datasets in a consumer-grade server.

1.2 Contribution

We propose algorithms and implementations for three fundamental graph problems using RDBMS :

- Source-to-Target Shortest Paths: Chapter 2 focuses on studying the com-

putation of shortest paths (SP) from a source to a target leveraging the intelligent scheduling that an RDBMS performs when executing set-at-a-time expansions of graph vertices. Proposed algorithms perform orders of magnitude faster than baselines and even faster than main memory algorithms for large graphs. Also, we show that our algorithms on RDBMS outperform counterparts running on modern native graph databases. Our research is published in the following article.

  - Aly Ahmed and Alex Thomo. Computing source-to-target shortest paths for complex networks in RDBMS. Journal of Computer and System Sciences (JCSS), 89:114–129, 2017. ([3])

- Page Rank Problem: Chapter 3 focuses on studying the Page Rank problem, which is vital in Google Search and social networks to determine how to sort search results and how influential a person is in a social group. Page Rank is an iterative algorithm that imposes challenges when implementing it over a large graph. We study computing Page Rank using RDBMS for very large graphs using a consumer-grade server and compare the results to a dedicated graph database. Our research is published in the following article.

  - Aly Ahmed and Alex Thomo. Pagerank for billion-scale networks in RDBMS. International Conference on Intelligent Networking and Collaborative Systems (INCoS), pages 89–100. Springer, 2020. ([4])

- Triangle Enumeration: Chapter 4 focuses on studying the Triangle enumeration problem, which is considered a fundamental graph analytics problem with many applications, including detecting fake accounts, spam

detection, and community searches. We present a carefully engineered RDBMS solution to the problem of triangle enumeration for extensive graphs. We show that RDBMSs are suitable tools for enumerating billions of triangles in billion-scale networks on a consumer-grade server. Also, we compare our RDBMS solution's performance to a native graph database. Our research is published in the following article.

- Aly Ahmed, Keanelek Enns, and Alex Thomo. Triangle enumeration for billion-scale graphs in RDBMS. In Advanced Information Networking and Applications (AINA), pages 160–173. Springer, 2021. ([2])

Chapter 2

Computing Source-to-Target Shortest Paths for Complex Networks in RDBMS

2.1 Introduction

Large graphs are everywhere nowadays. They model social and web networks, knowledge networks, and product co-purchase networks, to name a few. We label these networks “complex” to distinguish them from “spatial” networks, such as road networks that have been studied extensively. Our focus in this part of the dissertation is on complex networks.

Many classical graph algorithms face challenges when the graph is large. This is because they need random access to the vertices of the graph and their adjacency lists, and random access is expensive. While this is significantly more pronounced for data residing in external storage, it is also true for data that can fit in main memory (see [34] for discussions and experiments). For complex

networks the situation is even more challenging because many optimization ideas for spatial networks are not applicable to complex networks (see [65] for a survey on shortest path approaches for different kinds of networks).

One family of graph algorithms that we identify as particularly demanding for random-access is graph search. Graph search algorithms seek subgraphs that satisfy some property, such as the shortest paths between a source and destination [18], the minimum spanning tree rooted at a vertex [53], the connected component containing a vertex [30], and so on. Most algorithms of this family have an expand-and-explore nature that exhibits an intensive random access pattern. Therefore, they are good candidates for re-engineering so that random access is reduced.

In this part, we focus on *source-to-target* (s-t) shortest path queries (or simply s-t queries), also known as point-to-point queries in literature. These queries are central in social network analysis. For instance, graph distance (often referred to as social distance) can play an important role in deriving insights on user search in Linkedin, Facebook, and other social networks (see [31, 33, 66] for examples of using social distance). S-t queries have also been used to leverage trust in social links in online marketplaces [70] and shown to be an integral part of location and social aware search [64, 74]. Furthermore, the s-t queries are a building block in answering regular path queries on semi-structured data [24, 25, 47, 63].

S-t queries have a “local search” nature that is in contrast to the source-to-all queries which have a “global search” nature. As such, s-t queries are not a good fit for Pregel-like systems (such as Graphchi in [39]) which access the whole graph in each pass.

The approach we follow for computing s-t queries is to use relational databases as pioneered by [21]. Relational databases are a mature technology representing more than 40 years of active development. What relational databases offer is a set-at-a-time mode of operation, which allows data-access scheduling for grouping requests to disk blocks and thus reducing random access.

However, the main algorithm for finding shortest paths, the Dijkstra's algorithm, follows a vertex-at-a-time approach; it seeks to expand only the best vertex (path) discovered so far. On the other hand, other search algorithms, such as breadth first search (BFS), expand a set of vertices (paths) in each iteration, thus making possible to use the set-at-a-time mode of operation that a relational database offers. One can use BFS for s-t queries, however, the discovered paths might not be the best (shortest), and we need to re-expand vertices many times until we find the shortest paths.

The authors of [21] propose Bidirectional Restrictive BFS (B-R-BFS) which is an adaptation of BFS to reduce the number of vertex re-expansions. This is achieved by partitioning the table of graph edges into multiple tables based on the weights of edges. The algorithm is also bidirectional, meaning that it runs both from the source and the target until the two searches meet. The performance improvements over the Dijkstra's algorithm and pure BFS are impressive. However, deciding termination in B-R-BFS is challenging, and the condition proposed in [21] for checking termination is unfortunately not complete.

Our contributions in this part are as follows.

First, we show the problem with termination in B-R-BFS and then propose a new termination algorithm. In general, in any bidirectional s-t algorithm that

starts two search processes, one forward from s and the other backward from t, we need to determine whether both processes have *finalized* the distance of a vertex v from s and t, respectively. In such a case, we can successfully terminate the algorithm. In B-R-BFS, deciding whether a vertex v has its distance finalized is not easy as v can be expanded multiple times using different edge tables. The solution we propose is based on determining lower bounds for vertex distances and inferring a termination condition based on these bounds.

Second, we propose another algorithm, Bidirectional Level-based Frontier BFS (B-LF-BFS), for computing s-t queries in a set-at-a-time fashion. Differently from B-R-BFS, we achieve restrictive BFS not by splitting the edge table, but by selecting only a part of the visited vertices as a frontier to be expanded. The frontier contains only those vertices that have a distance estimate less than a "level" value. We show that, if the frontier is iteratively expanded until no more expansion is possible, then the distances of the vertices expanded during the current level are final. We, then, increase the current level to the next one (by adding a step value) and repeat the process. Since we have an explicit way to determine when vertices are finalized, we obtain a much simplified termination procedure.

Third, we enhance B-LF-BFS to use a graph representation where the neighbors of each vertex and their respective edge costs are compressed in an inverted-index style. We call this enhanced algorithm B-LF-BFS-C. We borrow ideas from Information Retrieval practice to perform compression by encoding the differences in neighbor ids using variable-byte encoding. The compression achieved is such that B-LF-BFS-C is able to handle graphs of an order of magnitude bigger than what B-R-BFS and B-LF-BFS can.

Finally, we present a detailed experimental study on real and synthetic datasets. We observe that all the above three algorithms outperform the vertex-at-a-time Dijkstra's algorithm in RDBMS by orders of magnitude. This strongly affirms the benefit of the set-at-a-time mode of operation offered by RDBMSs. Furthermore, we show that B-LF-BFS-C outperforms even a memory implementation of the Dijkstra's algorithm for a relatively large graph (Live Journal). We also show that B-LF-BFS-C can easily handle very large graphs, such as UK 2005, with close to one billion edges.

The rest of the chapter is organized as follows. In Section 2.2, we discuss other systems for graph management and processing. In Section 2.3, we present preliminaries and explain the challenges of computing shortest path queries on RDBMS. In Section 2.4, we describe the problem with termination detection in B-R-BFS and the proposed solution to fix it. In sections 2.5 and 2.6, we present the B-LF-BFS and B-LF-BFS-C, respectively. In Section 2.7, we present our experimental results. In Section 2.8, we describe the related works. Finally, Section 2.9 concludes the chapter.

2.2 Other Systems for Graph Management and Processing

Systems that allow for the storage and random access of big graphs are (native) graph databases. One of the main graph databases is Neo4j[1]. To make random access feasible, Neo4j builds indexes to quickly zoom in to a vertex and its neighborhood. A good index makes random access fast. However, if there are

[1] http://neo4j.com

massive requests for random access during the execution of an algorithm, the performance will still suffer. As we show in our experiments, Neo4j is considerably slower than the proposed algorithms on RDBMS; it even fails to return results for our larger graphs.

A very different approach is followed by the systems geared towards graph analytics. As representatives of such systems, we mention Pregel [43] for a distributed setting and GraphChi [39] for a single machine. They do not offer random access to a graph. Instead, they present a vertex-centric (VC) computation paradigm [43] where each vertex independently runs the same algorithm and sends and receives messages to and from its neighbors. VC systems for a single machine, such as GraphChi, significantly reduce random access to only a negligible amount. The tradeoff is multiple sequential passes over the graph. VC computation is quite good for some problems. Global graph search can be nicely implemented as a VC computation (see [39] for a discussion). For instance, finding the shortest paths from a source vertex to *all* the other vertices of a graph or finding *all* the connected components of a graph can be efficiently done as VC computations [39]. However, a VC computation is not a good fit for more local graph search, such as finding s-t shortest paths. The latency is too high as the whole graph will be accessed.

Our goal in this part is to provide algorithms for s-t shortest paths with a latency in the order of a few seconds (on a consumer-grade machine).

2.3 Preliminaries

We denote a directed, edge-weighted graph by $G = (V, E, C)$, where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, and $C : E \rightarrow \{x \in \mathbb{R} : x > 0\}$ is the edge-weight (or cost) function.

Let $p = [(u_0, u_1), \ldots, (u_{k-1}, u_k)]$, where $(u_{i-1}, u_i) \in E$ for $i \in [1, k]$, be a path from $u_0 \in V$ to $u_k \in V$. We denote by $c_p = \sum_{i=1}^{k} C(u_{i-1}, u_i)$ the *length* (or *cost*) of p.

Given two vertices s and t, we denote by $d(s, t)$ the length of the shortest path from s to t.

In this part, we are interested in source-to-target (s-t) queries which specify a vertex pair s, t and ask for the shortest path from s to t.

2.3.1 Graphs and Shortest Paths in RDBMS

We store the edges of a graph in a RDBMS in a table TE with three columns, *fid*, *tid*, and *cost*, for the source vertex id, target vertex id, and weight (cost) of an edge, respectively. We also construct indexes on *fid* and *tid*. For the ease of exposition, we will blur the distinction between a vertex and its id.

SP algorithms start out from the source vertex s and *expand* it by *reaching its neighbors*. One or more of the neighbors are expanded in turn, and we continue like this until we reach the target vertex t.

To accommodate expansions, we need a table, called TA, which stores the set of vertices that have been visited so far. Table TA has four columns: (1) *nid* for the id of a vertex we have visited, (2) $d2s$ for the length of the best path we have discovered so far from s to nid, (3) $p2s$ for the id of of the vertex coming

before nid in this path, and (4) f for flagging nid as finalized (the best path from s to nid has been discovered) or not. When a node u is finalized it means that the shortest path from source node s to u has been determined and the discovered distance estimate will not change to a lower value in a later iteration. The main goal of the shortest path problem is to finalize target node t. The computational challenge is to finalize t as quickly as possible.

Table TA is typically much smaller than TE, usually by one or two orders of magnitude. We do not create an index for TA as it is frequently updated.

In each iteration, we select from TA a set F of vertices for expansion. Vertex expansion is computed by joining F with TE. The newly visited vertices are merged into TA. Depending on the algorithm, a vertex can be visited multiple times. Each visit can (possibly) cause an update or insert into TA. We handle both cases using the *MERGE* operator in SQL.

Initially, $(s, 0, s, 0)$ is inserted into TA. At the end of an SP algorithm, we should have $(t, d(s,t), u_k, 1)$ in TA. Upon termination of the algorithm, we output the shortest path by following backwards the chain of tuples $(t, d(s,t), u_k, 1)$, ..., $(u_1, d(s,u_1), s, 1)$ in TA, where s, u_1, ..., u_k, t are the vertices along this path.

In order to speed up the computation, we can do bidirectional search and expansions. We start simultaneously from s in the forward direction and from t in the backward direction and discover paths that eventually meet at some intermediate vertex. We need two TA tables for bidirectional search, TA^f and TA^b. Also we refer to the F sets in the forward and backward directions as F^f and F^b, respectively.

Dijkstra's Algorithm on RDBMS Dijkstra's algorithm only expands one vertex at a time; the one with the smallest $d2s$ value. The expansion joins are fast

individually as each one only involves one tuple from TA^f (or TA^b) that needs to be joined with TE. Unfortunately, these joins are too many and the overall latency is high.

Termination. For the bidirectional Dijkstra's algorithm, the termination condition is when the forward search finalizes a vertex that has also been finalized by the backward search (or vice-versa). A vertex is finalized in the forward (backward) direction when it is selected to be in F^f (F^b).

2.3.2 Set-at-a-time Evaluation

One of the strengths of an RDBMS is its set-at-a-time evaluation mode. In graph search, set-at-a-time is more efficient than vertex-at-a-time because it allows the database to perform intelligent scheduling of buffer content and disk blocks; access requests to the same block can be bundled and scheduled at the same time, thus allowing for a better query evaluation plan.

Consider the join of F with TE. In the Dijkstra's algorithm, F has only one vertex. As such, the database needs to retrieve the edges of only one vertex for each join. In the worst case there can be n such join queries. Clearly, a vertex-at-a-time mode of operation is quite inefficient in this case; there will be unnecessary I/Os for retrieving the edges of different vertices when they can happen to be in the same block. In contrast, in a set-at-a-time mode, a block can be read once and serve many vertex expansions.

On the other hand, there is significant overhead if the set-at-a-time strategy is taken to the limit, which, in our case, translates to pure breadth-first-search (BFS). In BFS, all newly visited vertices are selected to be in F, and the expansion of all these vertices is achieved with a single join operation. However, BFS

may expand the same vertices multiple times, thus incurring significant overhead in the number of expansions compared to Dijkstra's algorithm. Therefore, we need to strike a balance between pure BFS and Dijkstra's algorithm.

In [21], the strategy proposed is a restrictive BFS. Similar to BFS, multiple vertices are selected to be in F. However, in each iteration, only a subset of edges is allowed to be used for expansion. More specifically, vertices are expanded first using the lightest edges, then using more heavier edges, and so on.

However, when performing bidirectional search under a BFS-like strategy, deciding termination becomes complicated. This is because when the two searches meet at some vertex v, we do not know whether v is finalized or not. Therefore, there is no guarantee that the path discovered is the shortest. In contrast, in the Dijkstra's algorithm, we have an easy way to finalize a vertex; this happens when the vertex is selected to be in F^f (F^b). This is not true for a BFS-like strategy.

2.3.3 Bidirectional Restrictive BFS (B-R-BFS)

Bidirectional Restrictive BFS (B-R-BFS) operates in set-at-a-time mode and performs much better than the Dijkstra's algorithm, however, its termination decision is not complete. In this section, we give an overview of B-R-BFS. In the next section, we show the problem with its termination and then present a correct termination procedure.

Partitioning the edge table. B-R-BFS starts by partitioning the TE table based on the edge weights. Formally it is done as follows.

Let pts be the desired number of partitioned tables and $[w_{min}, w_{max}]$ be the

range of edge weights. We denote by $[w_0, \ldots, w_{pts}]$ the edge-weight *partitioning vector*, where $w_0 = w_{min}$, $w_{pts} = w_{max} + \epsilon$,[2] and $w_i < w_{i+1}$ for $0 \leq i < pts - 1$. We create pts partition tables, TE_0, ..., TE_{pts-1}.[3] For each edge e in the graph, if $w_i \leq C(e) < w_{i+1}$, then e is put into partition table TE_i.

High-level overview of the algorithm. Recall the TA^f and TA^b tables we use for the forward and backward search, respectively. These tables, instead of the f column, will now have a different last column, fwd for TA^f and bwd for TA^b. fwd and bwd store the number of iteration during which the tuple was inserted or updated in TA^f or TA^b, respectively.

During an iteration, a vertex will only be expanded using one partition table. Once a vertex is selected to be in F^f (or F^b), it remains there for pts iterations until it is expanded using each of the partition tables. Initially, in the forward expansion, table TA^f will have the source node s. In the first iteration, s is selected to be in F^f and subsequently expanded using TE_0. The neighbors of s reachable using TE_0 are added in TA^f. Let the set of these neighbors be N_s^0. In the second iteration, we have $F^f = \{s\} \cup N_s^0$, and try to expand s using TE_1 while the vertices in N_s^0 using TE_0. In general, consider a vertex v that enters F^f in iteration i (a vertex enters F^f in the next iteration after it is inserted or updated in TA^f). In iterations i, $i+1$, ..., $i+(pts-1)$, vertex v will be expanded using tables TE_0, TE_1, TE_{pts-1}, respectively. An analogous logic is followed for the backward direction as well. In other words, a vertex can have *delayed expansions* during the pts iterations it remains in F^f (F^b).

[2] ϵ represents a very small number.

[3] In [21], the partition tables are numbered from 1 to pts. We choose to number them from 0 to $pts - 1$ in order to simplify the exposition of results later.

2.4 Termination Procedure for B-R-BFS

In this section we present a correct termination procedure for B-R-BFS.

Consider table TA^f (or table TA^b). We call a $d2s$ ($d2t$) value in table TA^f (TA^b) a *distance estimation* (DE). This is because it can (possibly) be lowered and become a real distance later on during the execution of the algorithm.

Definition 2.4.1. Let v be a visited vertex in the forward direction and $(v, d2s_v, p2s_v, fwd_v)$ be its tuple in TA^f at the end of iteration i in the execution of B-R-BFS. DE $d2s_v$ is called *distance* if and only if it cannot change in some later iteration $i' > i$.

An analogous definition can be stated for $d2t_u$ of a visited vertex u in the backward direction.

Given a tuple $(v, d2s_v, p2s_v, fwd_v)$ in TA^f or $(u,\ d2t_u,\ p2t_u,\ bwd_u)$ in TA^b, it is not easy to determine whether $d2s_v$ or $d2t_u$ are distances.

We define by m_i^f and m_j^b the minimum DE's discovered in iterations i and j in the forward and backward directions, respectively. Formally,

$$\begin{aligned} m_i^f &= \min\{d2s_v : (v, d2s_v, p2s_v, i) \in TA^f\} \\ m_j^b &= \min\{d2t_u : (u, d2t_u, p2t_u, j) \in TA^b\}. \end{aligned}$$

These values can be easily obtained by simple MIN queries on the TA^f and TA^b tables. We have $m_i^f = d2s_v$ and $m_j^b = d2t_u$ for some vertices v and u. We might be tempted to declare m_i^f and m_j^b to be distances. This is not always true however because $d2s_v$ and $d2t_u$ can be lowered in later iterations as result of delayed expansions.

For an example see Fig. 2.1 where we want to compute the shortest path from s to t. For simplicity, we are only considering the forward search. Suppose the partition vector is $[1, 5, 10 + \epsilon]$. We have two partition tables, $TE_0 = \{(s, u, 1), (s, v, 4), (v, t, 4)\}$ and $TE_1 = \{(u, t, 6), (t, s, 10)\}$.

Vertex s enters F^f in iteration 1, and stays there for iterations 1 and 2. Similarly, u enters F^f in iteration 2, and stays there for iterations 2 and 3. It is in iteration 3 that we find the shortest path from s to t via u. At the end of each iteration, F^f, TA^f and m_i^f are as follows.

Iteration 0: $F^f = \{\}$, $TA^f = \{(s, 0, s, 0)\}$, $m_0^f = 0$

Iteration 1: $F^f = \{(s, 0, s, 0)\}$,
$TA^f = \{(s, 0, s, 0), (u, 1, s, 1), (v, 4, s, 1)\}$, $m_1^f = 1$

Iteration 2: $F^f = \{(s, 0, s, 0), (u, 1, s, 1), (v, 4, s, 1)\}$,
$TA^f = \{(s, 0, s, 0), (u, 1, s, 1), (v, 4, s, 1), (t, 8, v, 2)\}$,
$m_2^f = 8$

Iteration 3: $F^f = \{(u, 1, s, 1), (v, 4, s, 1), (t, 8, v, 2)\}$,
$TA^f = \{(s, 0, s, 0), (u, 1, s, 1), (v, 4, s, 1), (t, 7, v, 3)\}$,
$m_3^f = 7$

As we can see, $m_2^f = 8$, corresponding to $d2s_t$, is not a distance because $d2s_t$ becomes 7 in iteration 3. Tuple $(u, 1, s, 1)$ is inserted into TA^f in iteration 1, and enters F^f in iteration 2. However, $(u, 1, s, 1)$ is not expanded using edge $(u, t, 7)$ in iteration 2. This expansion is *delayed* to iteration 3.

As in [21], let l_i^f be *the maximal distance* finalized (discovered) from s after the i-th forward iteration. Similarly, let l_j^b be *the maximal distance* finalized

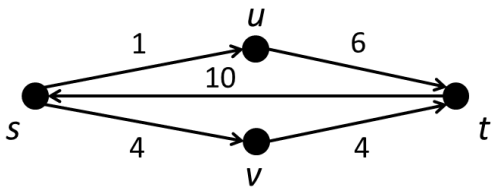


Figure 2.1: A graph illustrating the effect of delayed expansions.

(discovered) from t after the j-th backward iteration. In [21], l_i^f is proposed to be recursively computed as

$$l_i^f = \begin{cases} \min\{m_i^f, w_1\}, & \text{if } i = 1. \\ \min\{m_i^f, l_{i-1}^f + w_i - w_{i-1}\}, & i \geq 2. \end{cases}$$

and l_j^b is analogously computed replacing i by j and f by b.[4]

The above computations for l_i^f and l_j^b pose the following problem: what happens when i or j are larger than pts? In such a case, w_i or w_j are undefined and the above equalities do not work. This unfortunately is not addressed in [21].

In the following, we present another solution.It is inspired in part by an email communication we had with the authors of [21].

In fact, what we propose is computing *lower bounds* to l_i^f and l_j^b. We use ll_i^f and ll_j^b to denote the lower bounds for l_i^f and l_j^b and set them to be as follows.

First, $ll_0^f = l_0^f = m_0^f (= 0)$, $ll_0^b = l_0^b = m_0^b (= 0)$, and $ll_1^f = l_1^f = m_1^f$, $ll_1^b = l_1^b = m_1^b$.

Now, let $k = \min\{i-1, pts\}$ and $h = \min\{j-1, pts\}$. For $i, j \geq 2$, we set

$$ll_i^f = min\{m_i^f, m_{i-1}^f + w_1, \ldots, m_{i-k}^f + w_k^f\} \quad (2.1)$$

$$ll_j^b = min\{m_j^b, m_{j-1}^b + w_1, \ldots, m_{j-h}^b + w_h^b\}. \quad (2.2)$$

Suppose, for instance that $i = 2, 3, 4, 5$ and $pts = 3$. Then, $k = 1, 2, 3, 3$, and for the forward direction, we have

[4] In [21], l_i^f and l_i^b are used to terminate the algorithm if the following condition is met: minCost $\leq l_i^f + l_i^b$, where minCost is defined later in this section.

$$
\begin{aligned}
ll_2^f &= min(m_2^f, m_1^f + w_1) \\
ll_3^f &= min(m_3^f, m_2^f + w_1, m_1^f + w_2) \\
ll_4^f &= min(m_4^f, m_3^f + w_1, m_2^f + w_2, m_1^f + w_3) \\
ll_5^f &= min(m_5^f, m_4^f + w_1, m_3^f + w_2, m_2^f + w_3).
\end{aligned}
$$

We show the following theorem.

Theorem 1. $ll_i^f \leq l_i^f$ and $ll_j^b \leq l_j^b$.

Proof. Consider ll_2^f. We have that m_2^f is a distance of a vertex v (from s) unless there exists a vertex u that has remained with $fwd = 1$ after the 2nd iteration. Vertex u did not have any luck to be expanded by edges in $[w_0, w_1)$, however, it may be expanded by edges in the next interval and possibly cause the DE of v to get lower. In this case, the distance of v should be at least $m_1^f + w_1 (< m_2^f)$.

Now consider ll_i^f. We have that m_i^f is a distance of a vertex v (from s) unless there exists a vertex u that has remained with $fwd \leq i-1$ after the i-th iteration. Suppose u has $fwd = i - 1$. As such, u did not have any luck to join with edges in $[w_0, w_1)$, however again, it may be expanded by edges in the next interval and possibly cause the DE of v to get lower. In this case, the distance of v should be at least $m_{i-1}^f + w_1 (< m_i^f)$. Similarly, suppose u has $fwd = i - r$, for $r \in [1, k]$. As such, u did not have any luck to join with edges in $[w_0, w_1)$,..., $[w_{r-1}, w_r)$, however, it may be expanded by edges in the next interval and possibly cause the DE of v to get lower. In this case, the distance of v should be at least

$m^f_{i-r} + w_r (< m^f_i)$. From all the above, $ll^f_i \leq l^f_i$.

An analogous argument can be made for the backward direction as well. ■

Remark. We would like to emphasize here that m^f_i's (m^b_j's) are each time *re-computed*. For example, m^f_2 is recomputed when computing ll^f_3. This is because the vertex achieving the old m^f_2 might have been updated in the current iteration and can now have $fwd > 2$. Therefore another vertex with $fwd = 2$ will provide the new m^f_2. The new m^f_2 is greater than the old one.

For the termination decision we define

$$\mathsf{minCost} = \begin{cases} \min\{d2s_v + d2t_v\}, & \text{if } TA^f \bowtie_{nid} TA^b \neq \emptyset \\ \infty, & \text{otherwise.} \end{cases} \tag{2.3}$$

Now, we give the following condition that we use in the termination procedure.

$$\mathsf{minCost} \;\; \leq \;\; ll^f_i + ll^b_j. \tag{2.4}$$

If the above condition 2.4 is true, then by Theorem 1, the following condition is true as well.

$$\mathsf{minCost} \;\; \leq \;\; l^f_i + l^b_j. \tag{2.5}$$

If the last condition is true, then we can safely terminate B-R-BFS (see [21]). Upon such termination, minCost will be the length of the shortest path from the source to the target vertex. In other words, even though we might not have

computed yet l_i^f and l_j^b, we can infer that Condition 2.5 is satisfied based on Condition 2.4 using the lower bounds ll_i^f and ll_j^b.

With the modified termination condition provided in this section, each vertex in the shortest path p takes pts iterations to finalize, hence the algorithm is estimated to take $length(p) * pts$ iterations.

2.5 Bidirectional Level-based- Frontier BFS (B-LF-BFS)

Here we propose another set-at-a-time algorithm for computing shortest paths in RDBMS. Similarly to B-R-BFS, it works in a set-at-a-time fashion by expanding a set of vertices in each iteration. Differently from B-R-BFS, it achieves restrictive expansions not by splitting the edge table, but by selecting only a part of the visited vertices as a frontier to be expanded. The algorithm performs better in practice than B-R-BFS and it does not require partitioning the TE table into several tables as in B-R-BFS. This can be desirable as it does not need extra space for partition tables and code complexity is reduced.

The Bidirectional Level-based-Frontier BFS (B-LF-BFS) we propose uses the TE, TA^f and TA^b tables as defined before. We do not need to record the iteration number as in B-R-BFS, and we bring back the finalization flag f in TA^f and TA^b.

We define $F^f = F_i^f$, where F_i^f is the set of all the vertices in TA^f that are not finalized and their $d2s$ value is less or equal to L_i, where L_i is a *distance level*. We define F^b in an analogous way.

For the sake of explanation, let us assume we are working in the forward

direction and that initially we set $L_1 = step$, where $step$ is a small constant. Now the algorithm will (a) expand all the vertices in F^f (i.e. unfinalized vertices in TA^f that have $d2s \leq L_1$), (b) merge all the new vertices into TA^f, then (c) iterate and do the same, until no more unfinalized vertices with $d2s \leq L_1$ exist in TA^f. At this point we say that level L_1 is *cleared*, set $L_2 = L_1 + step$, and repeat the above operations for L_2.

In general, once level L_i is cleared, we set $L_{i+1} = L_i + step$, and repeat the above operations for L_{i+1}. This continues until termination is achieved. The algorithm terminates when a vertex v is finalized by both the forward and backward directions, or if there are no more unfinalized vertices in TA^f or TA^b.

Now, we illustrate the algorithm with an example. For the sake of simplicity we will explain the forward expansion; the backward expansion is similar. Consider the graph in Figure 1 and assume $step = 3$. In order to calculate the shortest distance between s and t, we initialize $L_1 = step = 3$, $\mathsf{minCost} = \infty$, and the TA^f table with the tuple $(s, s, 0, false)$. In the first iteration, the algorithm will expand the initial tuple in TA^f with $d2s = 0 \leq L_1$ and $f^f = false$. So, node s will be expanded and nodes u with $d2s$=1 and v with $d2s$=4 will be merged into TA^f and node s will be flagged as processed (its flag f^f becomes $true$). The algorithm will iterate to check if there are any tuples in TA^f with $d2s \leq L_1$ and $f^f = false$. Node u with $d2s = 1$ will be expanded and as a result node t with $d2s = 6$ will be merged into TA^f. At this point, no more nodes with $d2s \leq L_1$ are left to be processed, hence the algorithm declares that L_1 is cleared and sets $L_2 = L_1 + step = 6$. As node t is merged, the algorithm sets $\mathsf{minCost} = 6$. Next, the algorithm will expand the nodes that have not been processed yet and have $d2s \leq L_2$. As a result, node v with $d2s = 4$ will be expanded resulting in

rediscovering node t with higher cost $d2s = 8$, hence no tuple will be merged in TA^f. As no more nodes are left to process, level L_2 is cleared. At this point, node t will be flagged as finalized.

```
MERGE INTO TAf
USING (
  WITH

  --Compute frontier
  F(nid,p2s,d2s) AS (
    SELECT nid, p2s, d2s
    FROM TAf
    WHERE f_f='0' AND d2s <= Li
  ),

  --Join frontier with the edge table
  F_TE(nid,p2s,d2s) AS (
    SELECT TE.tid AS nid, TE.fid AS p2s, F.d2s+TE.cost AS d2s
    FROM F JOIN TE ON F.nid=TE.fid
  ),

  --For each nid, select the tuple with the smallest d2s value
  SELECT nid, p2s, d2s
  FROM (SELECT nid, p2s, d2s,
               row_number()  OVER
                 (PARTITION BY nid ORDER BY d2s) AS rn
          FROM F_TE)
  WHERE rn = 1

  ) source
ON (TAf.nid=source.nid)
WHEN MATCHED THEN
   UPDATE SET d2s=source.d2s, p2s=source.p2s, f_f='0'
   WHERE source.d2s<TAf.d2s
WHEN NOT MATCHED THEN
   INSERT  (nid,d2s,p2s,f_f)
   VALUES (source.nid, source.d2s, source.p2s, '0');

--Node finalization
UPDATE TAf SET f_f='1' WHERE f_f='0' AND d2s <= Li;
```

Figure 2.2: SQL statements for the B-LF-BFS algorithm.

More formally, we give the following definitions for B-LF-BFS.

Definition 2.5.1. The F^f and F^b sets in levels L_i and L_j are

$$\begin{aligned} F_i^f &= \{(nid, d2s, p2s, f^f) \in TA^f : f^f = 0 \text{ and } d2s \leq L_i\} \\ F_j^b &= \{(nid, d2t, p2t, f^b) \in TA^b : f^b = 0 \text{ and } d2t \leq L_j\}. \end{aligned}$$

Consider F_i^f, for some $i \geq 1$. The algorithm expands the vertices in F_i^f, then recomputes F_i^f. We give the following definition.

Definition 2.5.2. We say that level L_i, for $i \geq 1$, is *cleared* in the forward direction, if F_i^f is empty. Likewise, level L_j, for $j \geq 1$, is *cleared* in the backward direction, if F_j^b is empty.

Theorem 2. If level L_i is cleared in the forward direction, then the $d2s$ values in TA^f, such that $d2s \leq L_i$, are final and represent distances to source vertex s.

Proof.

Suppose not, i.e. let us assume there exists a vertex $v \in V$ processed in level L_i and assigned a $d2s$ value d, but in a later level, $L_{i'} > L_i$, v is assigned a lower $d2s$ value $d' < d$. Vertex v will be discovered in level $L_{i'}$ through a vertex, say u, not processed in iteration L_i, which implies that vertex u has a $d2s$ value greater than L_i, therefore $d' > L_i > d$, which is a contradiction. ■

Similarly, we can show that

Theorem 3. If level L_j is cleared in the backward direction, then the $d2t$ values in TA^b, such that $d2t \leq L_j$, are final and represent distances to target vertex t.

Once we clear a level L_i (L_j), we finalize all the vertices in TA^f (TA^b) with $d2s \leq L_i$ ($d2t \leq L_j$). We finalize those vertices by setting their f^f or f^b flag

to 1 ($true$). Observe that as we go to the next level, L_{i+1} (L_{j+1}), the set F^f_{i+1} (F^f_{i+1}) will contain those vertices of TA^f (TA^b) that are unfinalized and have a $d2s$ value less than L_{i+1} (L_{j+1}). Since, all the vertices of TA^f (TA^b) with $d2s \leq L_i$ ($d2t \leq L_j$) have been finalized, we have that in level L_{i+1} (L_{j+1}), we only process vertices of TA^f (TA^b) with $L_i < d2s \leq L_{i+1}$ ($L_j < d2t \leq L_{j+1}$).

In the B-LF-BFS algorithm, we have a clear way to finalize vertices, hence the termination decision becomes easier; it happens when we find a vertex that is finalized by both the forward and backward directions, or if there are no more unfinalized vertices in TA^f or TA^b. The validity of this condition for any bidirectional shortest-path algorithm (that finalizes vertices) is shown in [29]. In contrast, we did not have the ability to easily decide how to finalize vertices in B-R-BFS, hence, we had to resort to a much more complex termination procedure.

The computed F^f and F^b sets in B-LF-BFS are only a part of the TA^f or TA^b tables. Therefore the joins with the TE table are restricted in size compared to a full BFS approach.

Whereas B-R-BFS achieves the join reduction by partitioning the TE table, B-LF-BFS achieves a similar effect by performing first a selection on the TA^f and TA^b tables to generate a smaller set to join with TE.

The pseudo-code for clearing a level in B-LF-BFS in the forward direction is given in Algorithm 1. Please refer to Fig. 2.2 for the SQL statements. Once a level is cleared, we go to the next level until no more expansions are possible. The backward direction is analogous. B-LF-BFS alternates between the forward and backward direction depending on the number of tuples merged in TA^f and TA^b choosing each time the direction with the fewer merges.

Algorithm 1 B-LF-BFS expand and merge

1: **function** ExpandAndMerge(TA^f, TE, L_i)
2: **do**
3: Compute F_i^f (view F)
4: Join F_i^f with TE (view F_TE)
5: For each nid in F_TE,
6: compute the tuple with the smallest $d2s$
7: Merge all the tuples thus produced into TA^f
8: $n \leftarrow$ number of merged tuples
9: **while** $n \neq 0$
10: Finalize all vertices in TA^f with $f^f = 0$ and $d2s \leq L_i$
11: **end function**

The computations in lines 3–7 of Algorithm 1 are implemented by the SQL *MERGE* statement in Fig. 2.2. There, we first create two views, F and F_TE.[5] View F contains set F^f. View F_TE contains the join of F^f with TE.

Next, for each vertex id, nid, we compute the tuple with the smallest $d2s$ in F_TE. For this we use the **row_number**() SQL window function[6]. Then comes the merge of thus obtained tuples into table TA^f. If we obtained a tuple that is better (with respect to $d2s$) than a tuple with the same nid in TA^f, then the latter will be replaced by the former. Also, the tuples that do not have counterparts in TA^f (with respect to nid) will be simply inserted into TA^f.

Finally, we finalize all the vertices in TA^f with $f^f = 0$ (f_f = '0') and $d2s \leq L_i$. The finalization of vertices (tuples) in TA^f is done by setting the value of their f^f flag (f_f) in TA^f to 1. This is done with the last SQL statement in Fig. 2.2.

Now we analyze the number of iterations the algorithm would take. Suppose first we only do forward search. Let $mc > 0$ be the cost of the lightest edge in the graph. Let mp be the cost of the shortest path from s to t. In order to clear

[5] We are calling F and F_TE "views", however, they are more precisely called "factored subqueries" (created with the SQL keyword *WITH*).

[6] See `https://en.wikipedia.org/wiki/Select_(SQL)`

a level, we need $\lceil mp/step \rceil$ iterations in the worst case. In order to discover the shortest path from s to t, we need $\lceil mp/step \rceil$ levels in the worst case. Therefore, we need $\lceil (mp/step) \rceil * (step/mc)$ iterations in the worst case. When we do bidirectional search, we will need about half this number of iterations. This analysis shows that the algorithm terminates in a finite number of iterations. Based on the above reasoning and Theorem 2, we conclude the correctness of the algorithm.

2.6 B-LF-BFS with compressed adjacency lists (B-LF-BFS-C)

In this section, we present B-LF-BFS-C, which enhances B-LF-BFS using a compressed representation of the input graph.

A RDBMS often uses more space than necessary for storing numeric datatypes (cf. [34]). Also, an edge table, such as TE, has unnecessary redundancy. For example, if a highly connected vertex v has, say 1000 neighbors, $u_1, \ldots, u_{1000}$, then v's id will repeat 1000 times to represent the 1000 edges that connect v to its neighbors, i.e. we will have the triples (v, u_1, c_1), (v, u_2, c_2), $\ldots$, (v, u_{1000}, c_{1000}).

A better alternative is to use an adjacency list of neighbors and costs, e.g. for v, we would have a list such as $[u_1, c_1, u_2, c_2, \ldots, u_{1000}, c_{1000}]$. While this is an improvement, we can do better than just storing the numbers in their original form. In fact, we can compress an adjacency list quite efficiently.

We borrow the idea of variable-byte-encoding of postings lists from Information Retrieval practice (cf. [44, 7, 81]). A posting list for a term is a list of documents that contain the term. For example, for a term, say *dog*, we can have

a posting list like $[334, 345, 350]$, where the numbers are document ids containing *dog*. Observe that document ids are sorted in ascending order.

There is a similarity between a posting list and an adjacency list; instead of document ids we have neighbor ids.

For representing a posting list, we do not store the original document ids; rather, we store the *gaps* (or differences) between document ids. So, in the previous example, the posting list becomes $[334, 11, 5]$, where 11 is $345 - 334$, and 5 is $350 - 345$. Now, variable-byte encoding is used for the modified posting list. Specifically, we need 2 bytes for 334, and only one byte for each of the other two numbers, for a total of 4 bytes. In contrast, the original posting list, with fixed-byte-encoded integers of, say 4 bytes, needs 12 bytes.

We applied this idea for encoding the graph adjacency lists. We stored the obtained byte-encodings as *BLOB*'s (Binary Large Objects) in the TE table. More specifically, the TE table has now only two attributes, *fid* (as before) and *ncb* (which stands for *neighbor-cost bytestream*).

The pseudo-code for encoding/decoding sorted adjacency lists is given in algorithms 2, 3, and 4.

A number is encoded by a list of bytes. The rightmost 7 bits in a byte are *content* and represent a part of the number. The leftmost bit is an indicator flag. If it is 1, it means that the byte is the last one in the number encoding. If it is 0, it means there are more bytes following up in the encoding. In Algorithm 3, we encode a list of neighbor/cost numbers. We iterate over the elements of this list in pairs of neighbor and cost. We encode the difference of the current neighbor from the previous one, then encode the cost of the edge reaching the neighbor. When decoding a list of bytes (see Algorithm 4), we check for the leftmost bit of

the bytes we read in order to detect the end of a number encoding. To decode a number, we extract and put together the 7 rightmost bits of the bytes in its encoding. We also check to see if the number we decoded is a neighbor (gap) or a cost, and proceed accordingly.

Algorithm 2 Encoding of a number a

```
1: function encode(a)
2:     bytelist ← ∅
3:     do
4:         bytelist ← bytelist.prepend(a & 0x7F)
5:         a ← a >> 7
6:     while a > 0
7:     bytelist[bytelist.size] = bytelist[bytelist.size] | 0x80
8:     return bytelist
9: end function
```

Algorithm 3 Encoding of a list of neighbor/cost numbers

```
1: function encode(list)
2:     bytelist ← ∅
3:     prev_neighbor ← 0
4:     for each neighbor, cost in list do
5:         δ ← neighbor − prev_neighbor
6:         bytelist_1 ← encode(δ)
7:         bytelist_2 ← encode(cost)
8:         bytelist.concatenate(bytelist_1)
9:         bytelist.concatenate(bytelist_2)
10:        prev_neighbor ← neighbor
11:    end for
12:    return bytelist
13: end function
```

Expansion. In the following we describe the forward expansion. The backward version is analogous. The set F^f is computed using a similar query as before (see view F in Fig. 2.3). The join of F^f with TE returns now a result set of $(nid, ncb, d2s)$ tuples. The ncb value is a list of bytes encoding the neighbors

Algorithm 4 Decoding of a list of bytes $bytelist$

```
1: function decode(bytelist)
2:     list ← ∅, a ← 0
3:     prev_neighbor ← 0, is_neighbor ← true
4:     for each byte in bytelist do
5:         if byte < 0x80 then
6:             a ← a << 7 | byte
7:         else
8:             byte ← byte & 0x7F
9:             a ← (a << 7) | byte
10:            if is_neighbor = true then
11:                a ← prev_neighbor + a
12:                prev_neighbor ← a
13:                is_neighbor ← false
14:            else
15:                is_neighbor ← true
16:            end if
17:            list.append(a)
18:            a ← 0
19:        end if
20:    end for
21:    return list
22: end function
```

of nid and the costs to reach these neighbors. The $d2s$ value represents the distance estimation of nid from the source vertex.

The neighbor-cost byte lists in the join result need to be decoded first in order to obtain tuples that can be merged into TA^f. This is done by a procedure described in Algorithm 5. In this procedure, we populate a new table, $EX^f(nid, d2s, p2s)$, which will contain the tuples that will be merged into TA^f. We truncate (clean-up) this table at each expansion round.

After performing the clean-up of EX^f (first statement in Fig. 2.3), then the computation of F^f, and the join with TE, the procedure proceeds with the creation of the tuples for EX^f. Specifically, for each tuple in the join result, it decodes the ncb byte list and iterates over the produced numbers. The iteration

is done in pairs a, b to account for neighbor id and cost (a is neighbor id, b is cost). Let t be the current tuple being processed from the join result. The $d2s$ and $p2s$ values of the new tuple we create are set to be $t.d2s + b$ and $t.fid$, respectively.

Finally, once we populate the EX^f table, we merge it with TA^f using the SQL *MERGE* statement in Fig. 2.3. Differently from Fig. 2.2, the view creations in Fig. 2.3 are not part of the *MERGE* statement. Within the *MERGE* statement, for each nid, we first compute the tuple with the smallest $d2s$ in EX^f. Then, we proceed with the merge of these tuples into TA^f. The vertex finalization query is the same as in Fig. 2.2. The main algorithm for clearing a given level is shown in Algorithm 6.

Algorithm 5 Populating expansion table EX^f

```
1: procedure populate(EX^f, TA^f, TE, L_i)
2:     EX^f ← ∅ (truncate statement in Fig. 2.3)
3:     Compute F_i^f (view F in Fig. 2.3)
4:     Join F_i^f with TE (view F_TE in Fig. 2.3)
5:     for each t in F_TE do
6:         list ← decode(t.ncb)
7:         for each a, b in list do
8:             nid ← a, d2s ← t.d2s + b, p2s ← t.fid
9:             Insert (nid, d2s, p2s) into EX^f
10:        end for
11:    end for
12: end procedure
```

2.7 Experimental Results

Setup. All our experiments are conducted on a consumer-grade machine with Intel i7, 3.4Ghz CPU, and 12Gb RAM, running Windows 7 Professional. The hard disk is Seagate Barracuda ST31000524AS 1TB 7200 RPM. We used the

```
--Prepare the EXf table for a fresh expansion
TRUNCATE TABLE EXf;

WITH
--Compute frontier
F(nid,p2s,d2s) AS (
    SELECT nid, p2s, d2s
    FROM TAf
    WHERE f_f='0' AND d2s <= Li
),

--Join frontier with the edge table
F_TE(nid,ncb,d2s) AS (
    SELECT TE. d, TE.ncb, TA.d2s
    FROM F JOIN TE ON F.nid=TE.fid
),
SELECT * FROM F_TE;

--Merge into TAf
MERGE INTO TAf
USING (
  --For each nid, select the tuple with the smallest d2s value
   SELECT nid, p2s, d2s
   FROM (SELECT nid, p2s, d2s,
                row_number() OVER
                  (PARTITION BY nid ORDER BY d2s) AS rn
            FROM EXf)
   WHERE rn = 1
) source
ON (TAf.nid=source.nid)
WHEN MATCHED THEN
    UPDATE SET d2s=source.d2s, p2s=source.p2s, f_f='0'
    WHERE source.d2s<TAf.d2s
WHEN NOT MATCHED THEN
    INSERT (nid,d2s,p2s,f_f)
    VALUES(source.nid, source.d2s, source.p2s, '0')
```

Figure 2.3: SQL statements for B-LF-BFS-C.

Algorithm 6 B-LF-BFS-C expand and merge

1: **function** ExpandAndMerge(TA^f, TE, L_i)
2: **do**
3: $populate(EX^f, TA^f, TE, L_i)$
4: Merge EX^f into TA^f (*MERGE* in Fig.2.3)
5: $n \leftarrow$ number of merged tuples
6: **while** $n \neq 0$
7: Finalize all vertices in TA^f with $f^f = 0$ and $d2s \leq L_i$
8: **end function**

latest versions of two commercial databases (which we anonymously call D1 and D2) as well as PostgreSQL 9.4.4 (PG).

We performed our analysis on six real and ten synthetic graph datasets. We show the results for the real datasets in Fig.(2.6, 2.4, 2.5) and for synthetic datasets in Fig.(2.7, 2.8, 2.9).

The real datasets are Web-Google, Pokec, Live-Journal (all three from `http://snap.stanford.edu`), and UK 2002, Arabic 2005, UK 2005 (all three from `http://law.di.unimi.it/webdata`).

The characteristics of the real datasets are as follows.

Table 2.1: Real Graph Datasets

Data Set	**Nodes#**	**Edges#**	**Diameter**
Web-Google	875,713	5,105,039	21
Pokec	1,632,803	30,622,564	11
Live Journal	4,847,571	68,993,773	16
UK 2002	18,520,486	298,113,762	21
UK 2005	39,459,921	936,364,282	23
Arabic 2005	22,744,080	639,999,458	22

The last three datasets, UK 2002, Arabic 2005, and UK 2002 are significantly bigger than the first three as well as the datasets considered in [21]. UK 2005, for instance, has close to one billion edges. We give a bar chart of the edge numbers in Fig. 2.4d.

The synthetic datasets vary in size from 1 million edges to 15 million edges. We generated five random graphs with sizes of 1, 2, 5, 10, and 15 million edges (denoted by 1M, 2M, 5M, and 15M), and five graphs of the same sizes using the preferential attachment model.

In figures 2.7b, 2.7c, and 2.7d, we compare the performance of B-R-BFS, B-LF-BFS, and B-LF-BFS-C on random vs. preferential attachment graphs. We

see that their performance on the two types of graphs is more or less the same. Therefore, we show results using random graphs in the rest of the charts of Fig.(2.7, 2.8, 2.9).

For edge weights, we generated random numbers from 1 to 100 using uniform distributions for both real and synthetic datasets.

Regarding indexes we experimented with clustered and non- clustered indexes on the edge table. The results using clustered indexes are better (see figures 2.9c and 2.9d). For the TA tables, we did not create indexes as this slowed down the *MERGE* operations and the performance of all the algorithms suffered.

Each running time is given in seconds and obtained as an average over 100 random s-t queries.

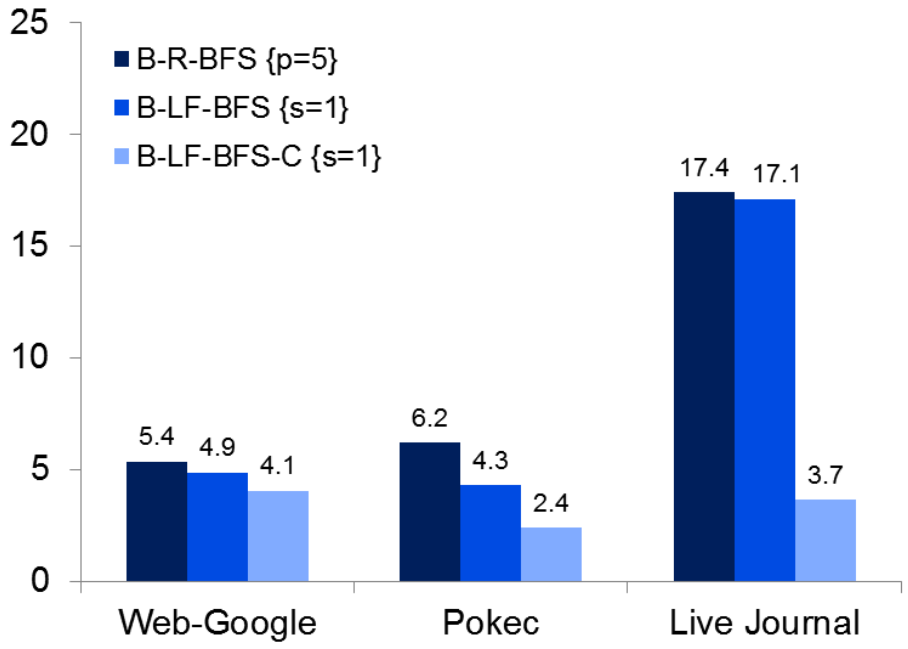
In the following, we give the questions we aim to address with our experiments.

Questions.

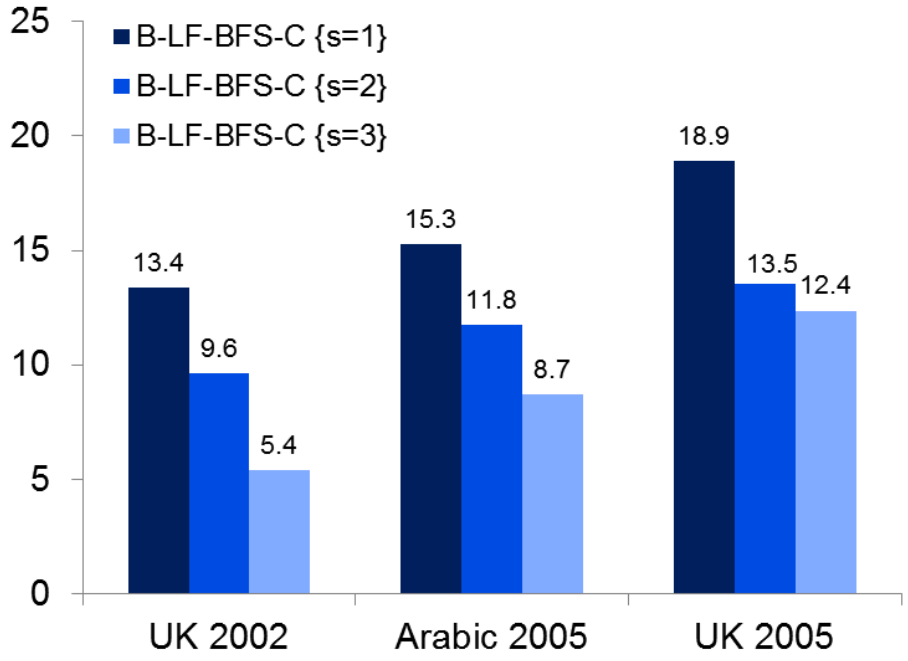
Q1 How scalable are the algorithms we consider? How do they compare to each other? In particular, can they handle large and very large graphs, e.g. Live Journal (large) and UK 2005 (very large)?

Q2 How well do the algorithms perform against baselines, such as the following variants of the bidirectional Dijkstra's algorithm: (a) in memory (when the graph fits there), (b) in RDBMS, and (c) in a modern native graph database (Neo4j)?

Q3 What are the best parameters for B-R-BFS, B-LF-BFS, and B-LF-BFS-C (number of partitions, p, for the first, and step size, s, for the second and

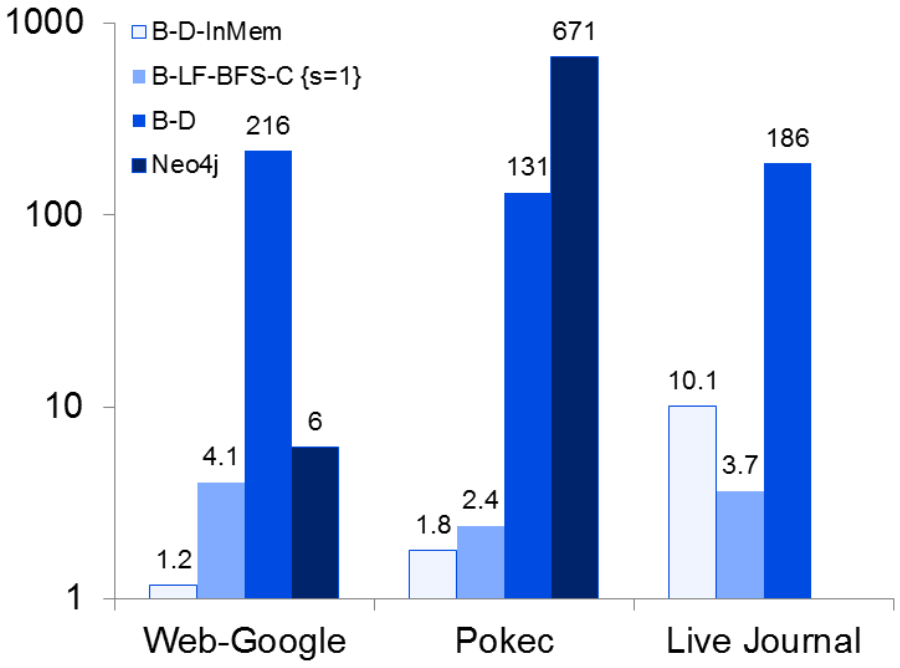


(a) Algorithms under their best parameter settings.

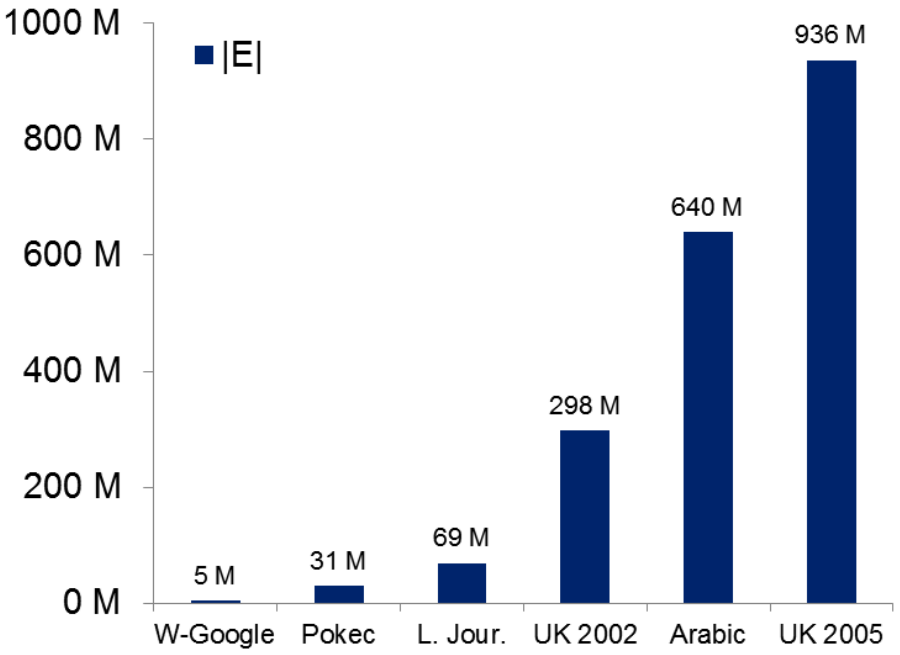


(b) B-LF-BFS-C, different step sizes, big graphs

Figure 2.4: Experimental results for real datasets. All values on the vertical axes are times in *seconds*, except for 2.4d.

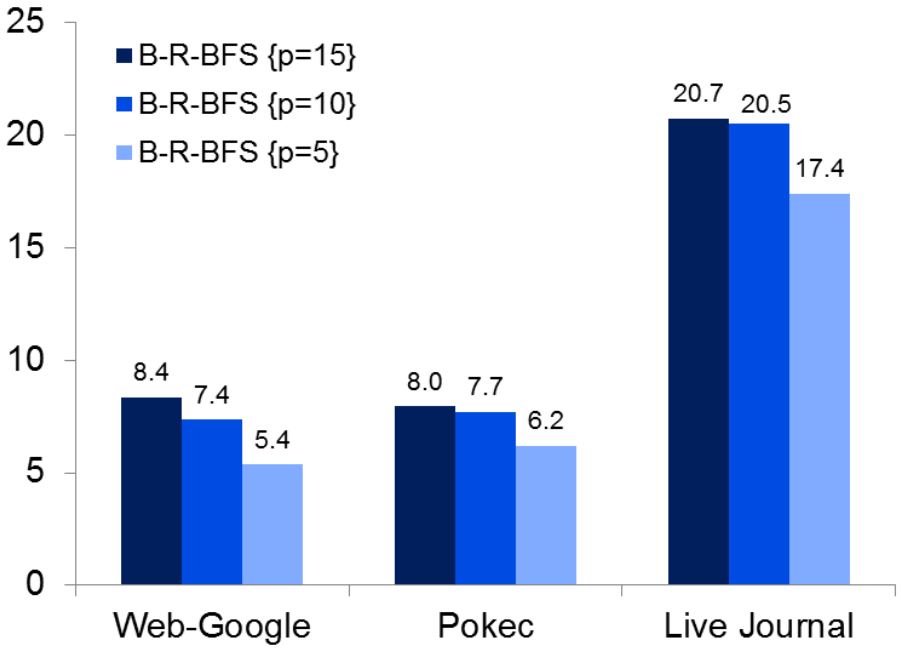


(c) B-LF-BFS-C vs. baselines.

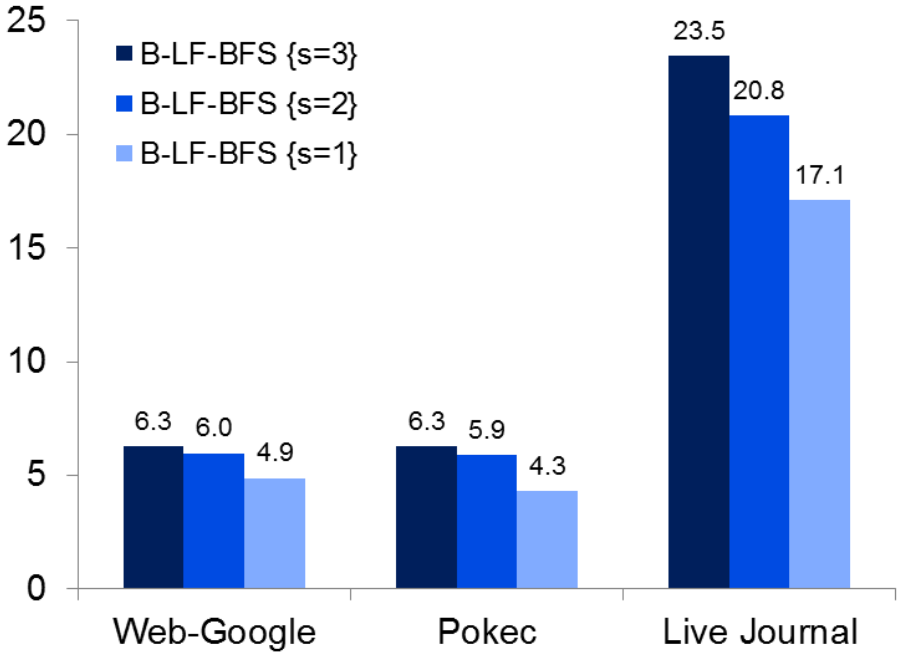


(d) Graph sizes in millions of edges (for reference).

Figure 2.4: Experimental results for real datasets. All values on the vertical axes are times in *seconds*, except for 2.4d.

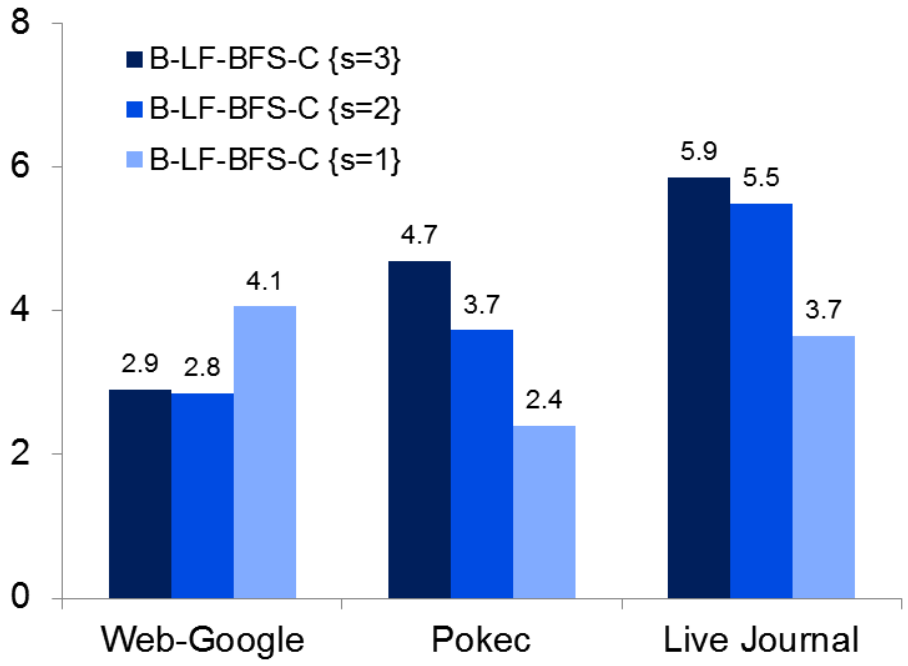


(a) B-R-BFS, different partition numbers.

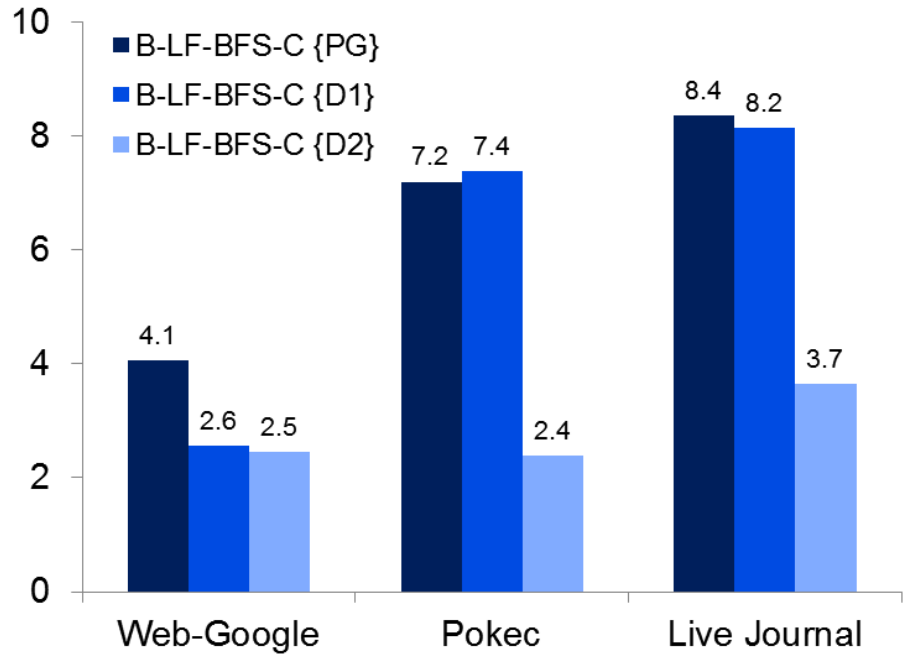


(b) B-LF-BFS, different step sizes.

Figure 2.5: Experimental results for real datasets. All values on the vertical axes are times in *seconds*.

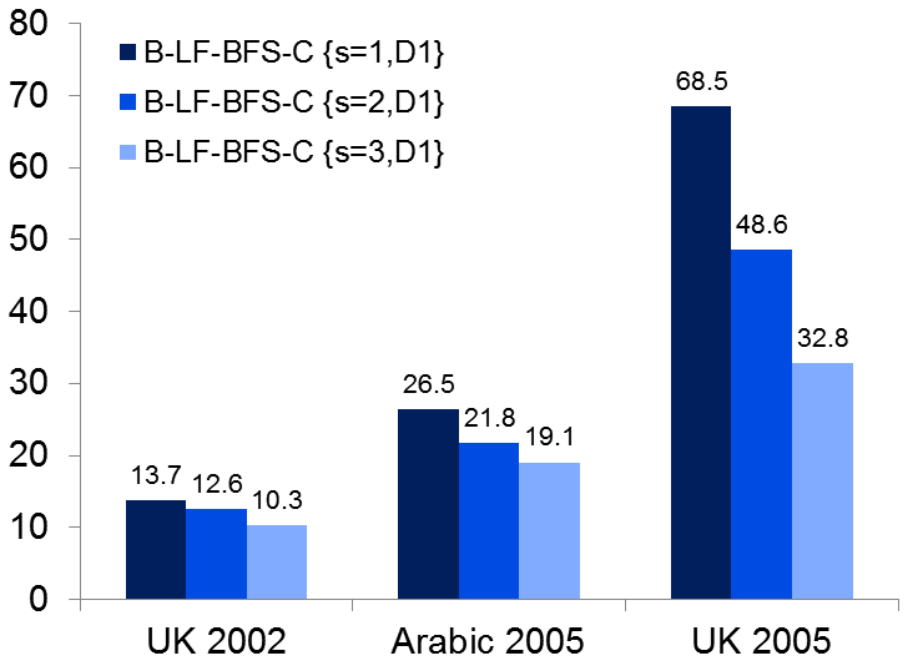


(c) B-LF-BFS-C, different step sizes.

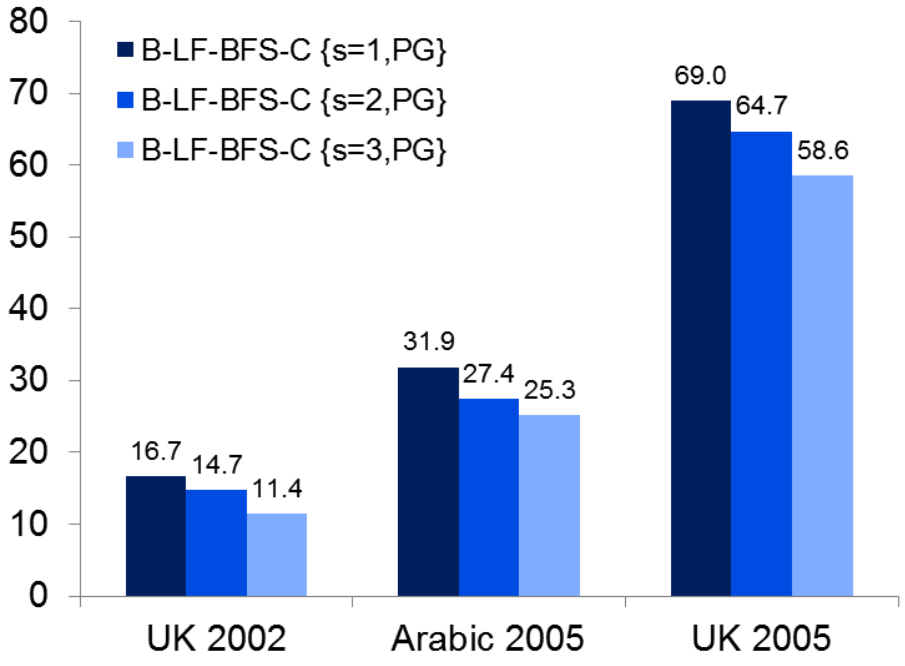


(d) B-LF-BFS-C {s=1}, different RDBMSs.

Figure 2.5: Experimental results for real datasets. All values on the vertical axes are times in *seconds*.

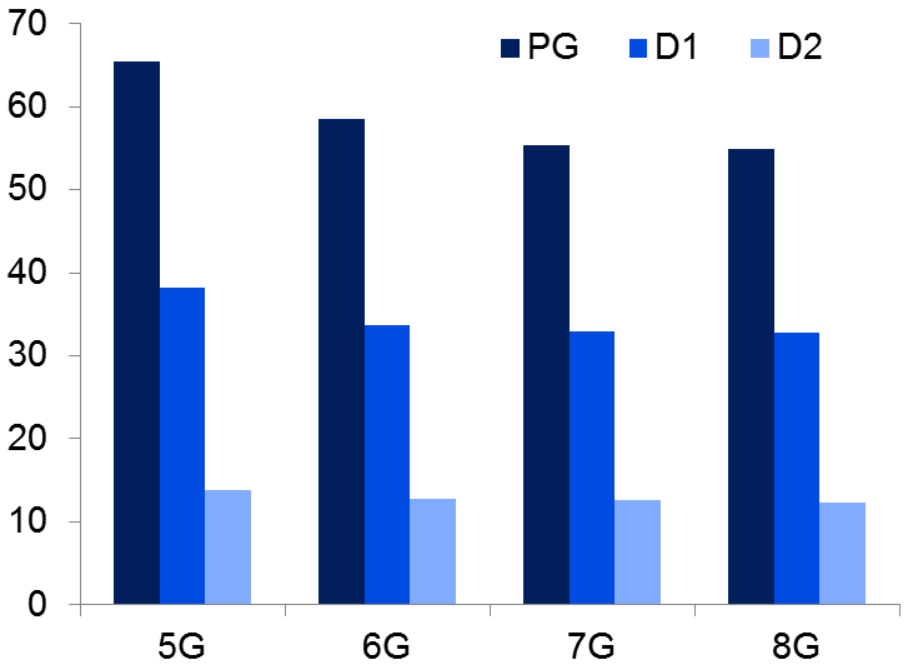


(a) B-LF-BFS-C, different step sizes, big graphs, D1.

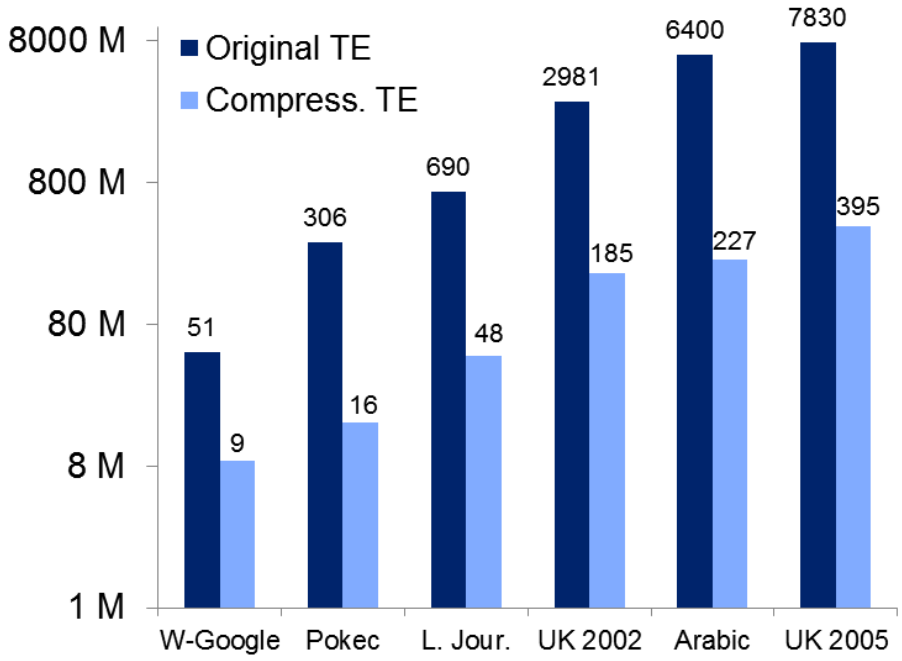


(b) B-LF-BFS-C, different step sizes, big graphs, PG.

Figure 2.6: Experimental results for real datasets. All values on the vertical axes are times in *seconds*, except for 2.6d.

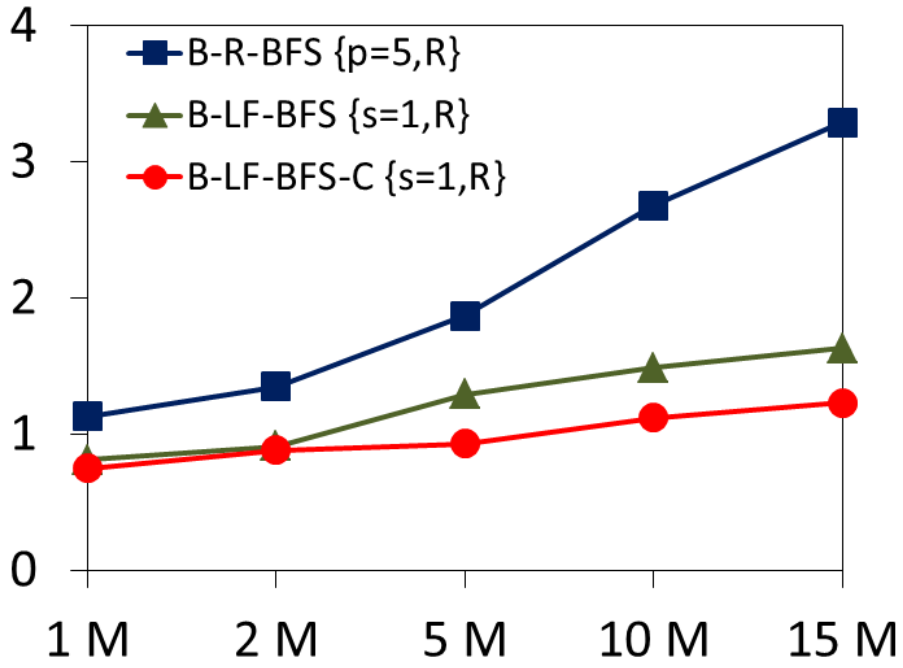


(c) Buffer size influence on run. time (B-LF-BFS-C).

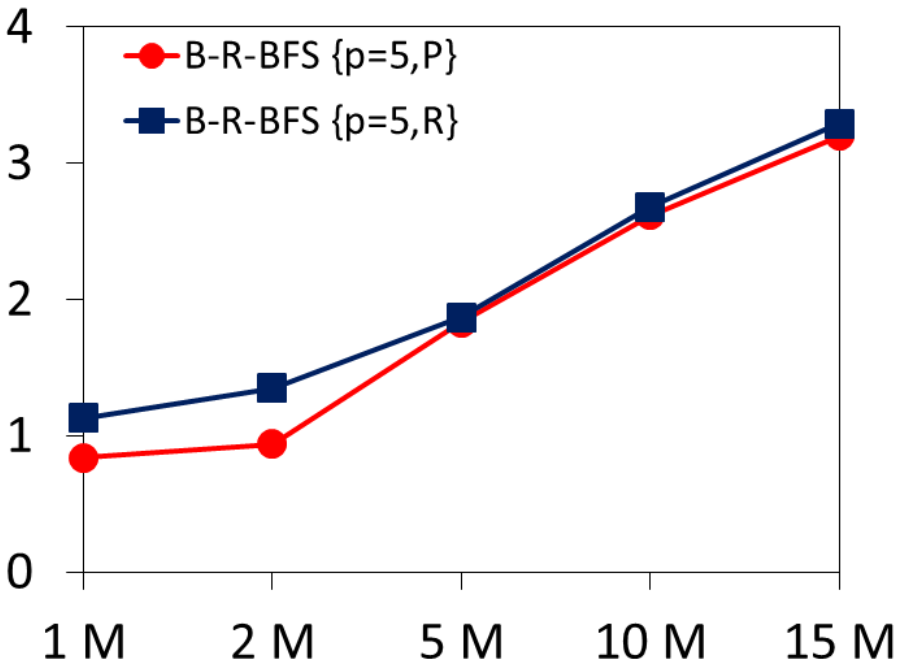


(d) Original vs. Compressed TE (megabytes).

Figure 2.6: Experimental results for real datasets. All values on the vertical axes are times in *seconds*, except for 2.6d.

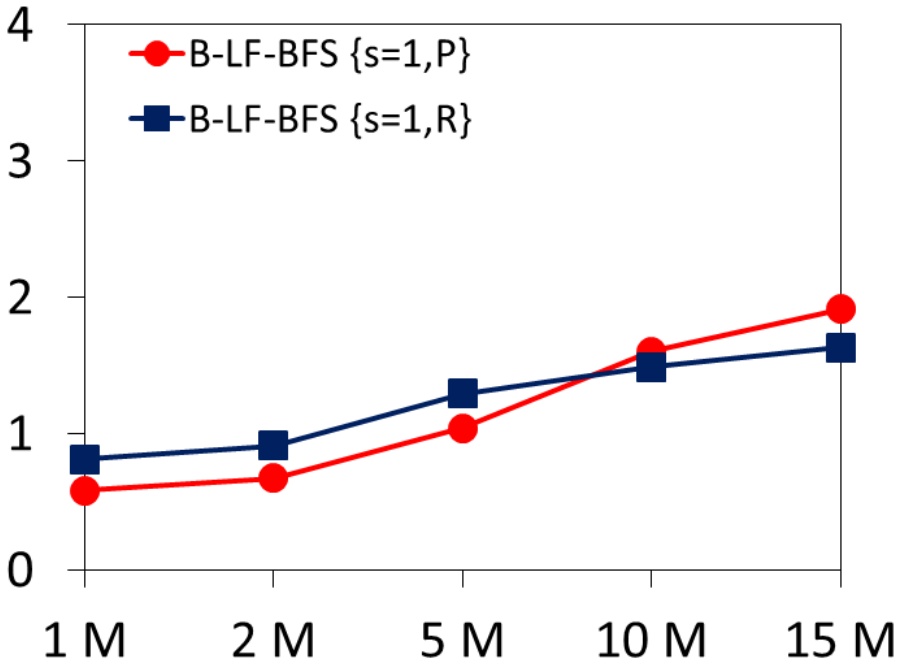


(a) Algorithms under their best parameter settings.

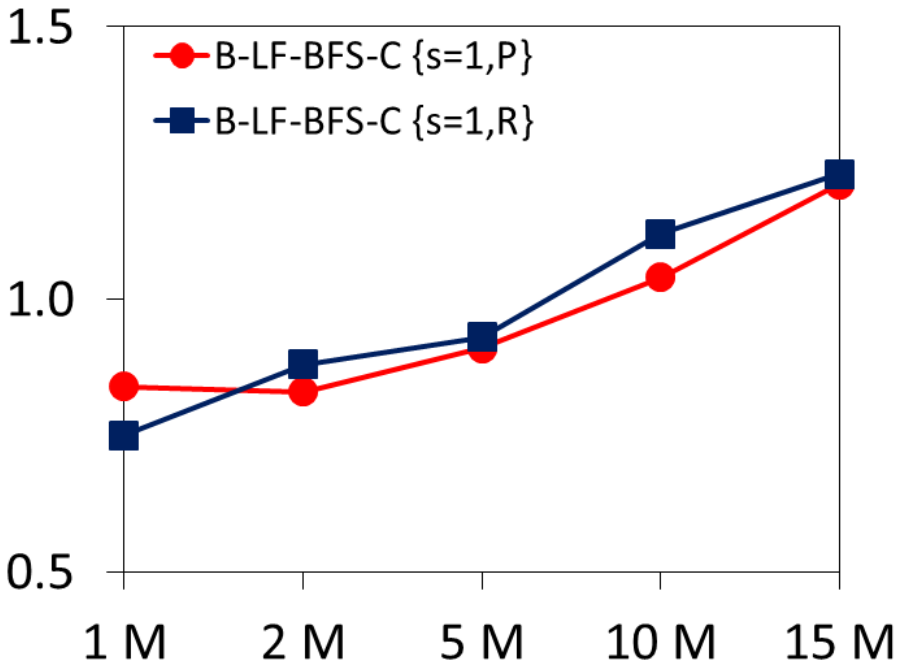


(b) B-R-BFS, pref. attachment and random graphs.

Figure 2.7: Experimental results for synthetic datasets. All values on the vertical axes are times in *seconds*.

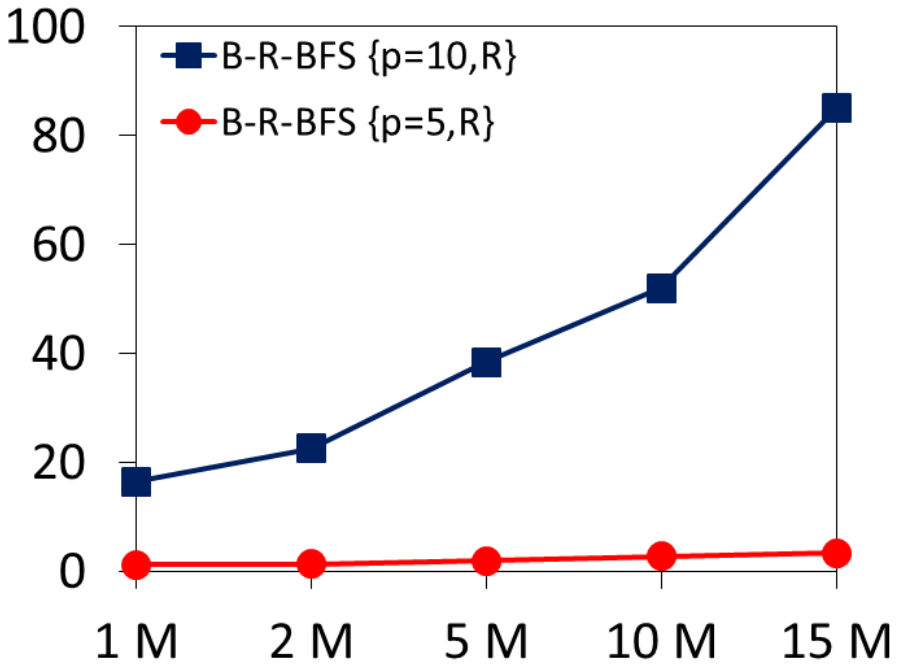


(c) B-LF-BFS, pref. attachment and random graphs.

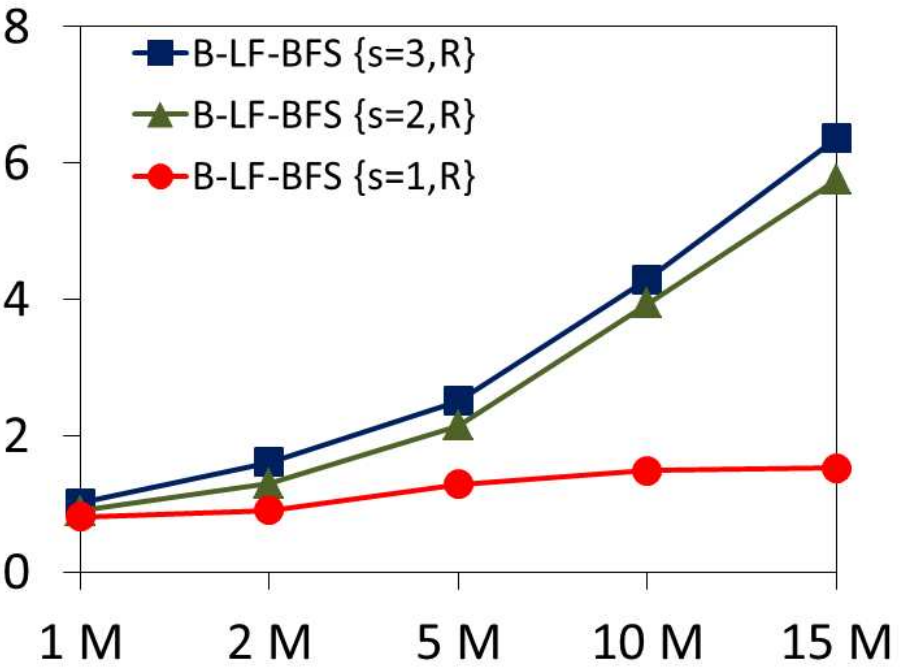


(d) B-LF-BFS-C, pref. attach. and rand. graphs.

Figure 2.7: Experimental results for synthetic datasets. All values on the vertical axes are times in *seconds*.

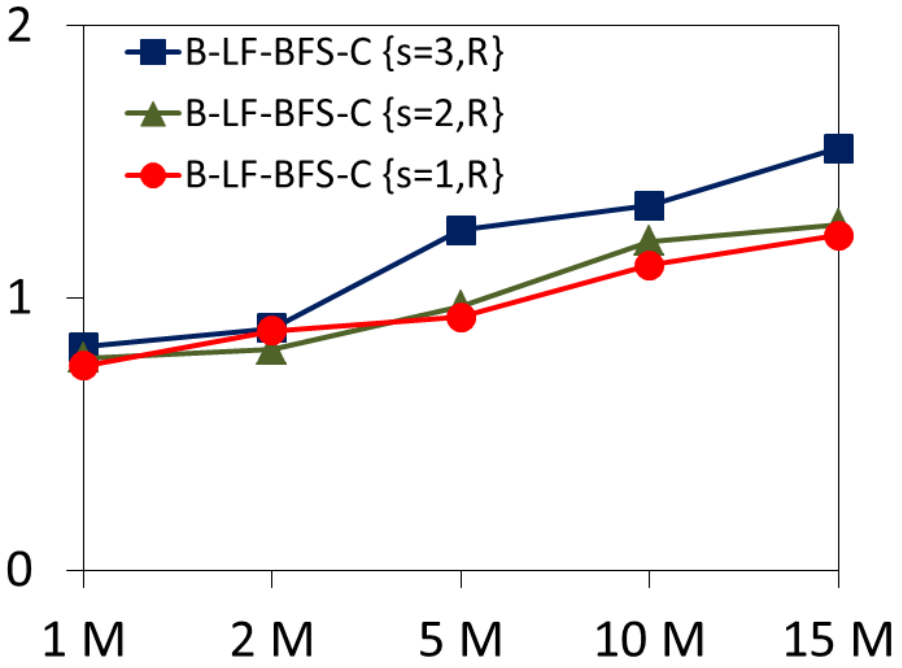


(a) B-R-BFS, different partition numbers.

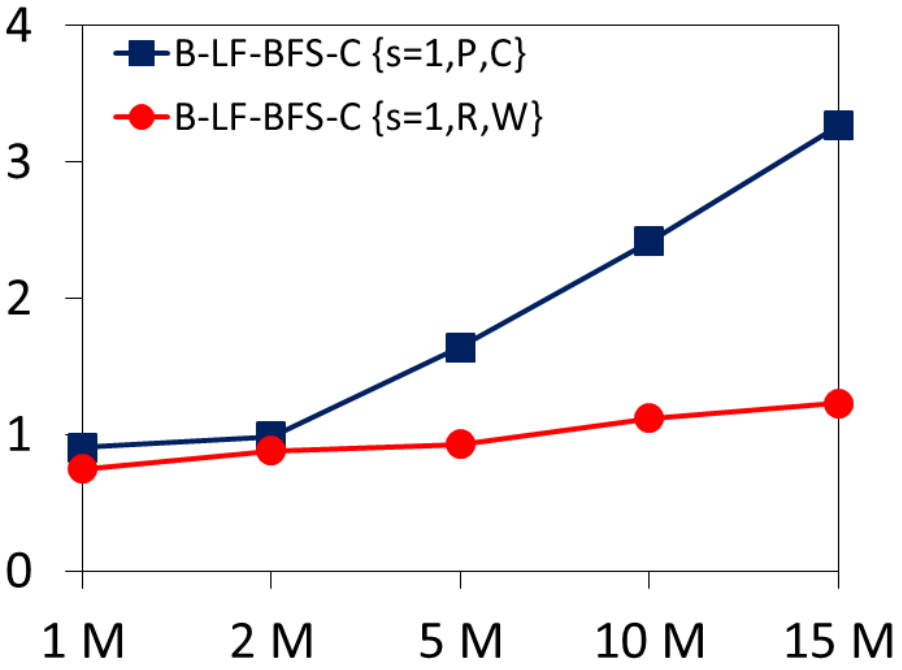


(b) B-LF-BFS, different step sizes.

Figure 2.8: Experimental results for synthetic datasets. All values on the vertical axes are times in *seconds*.

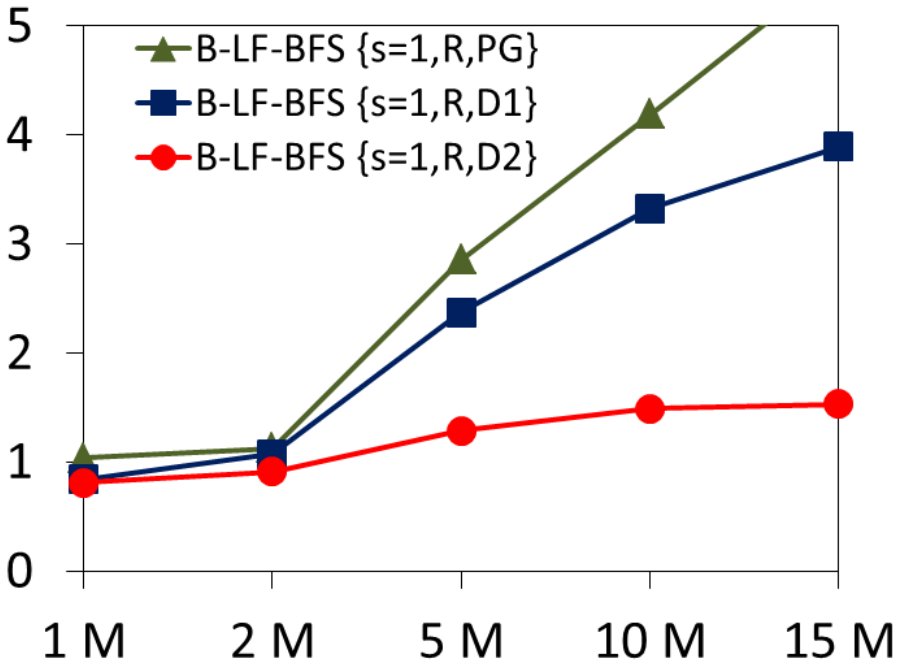


(c) B-LF-BFS-C, different step sizes.

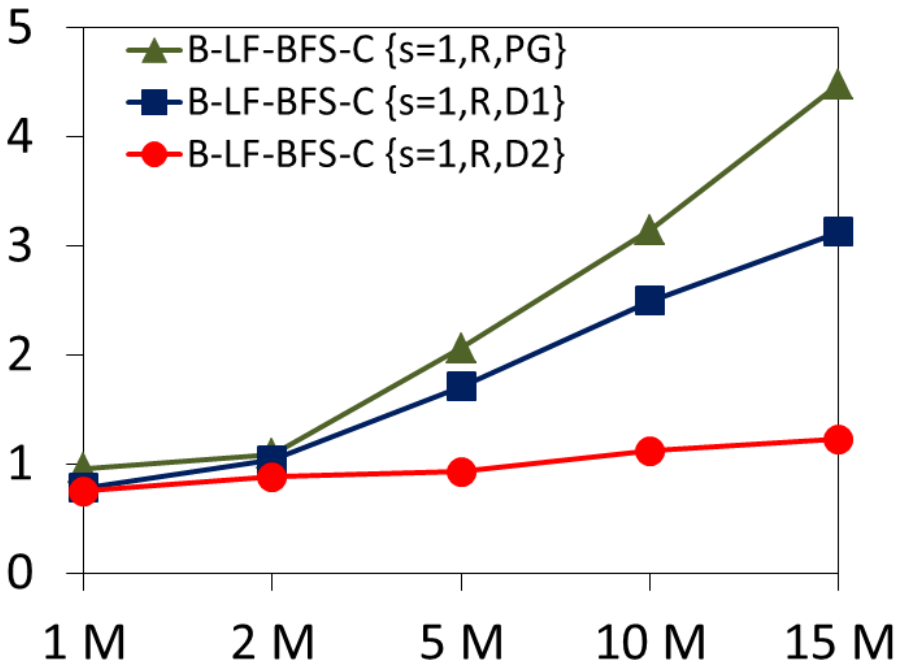


(d) B-LF-BFS-C, window functions vs. classical SQL.

Figure 2.8: Experimental results for synthetic datasets. All values on the vertical axes are times in *seconds*.

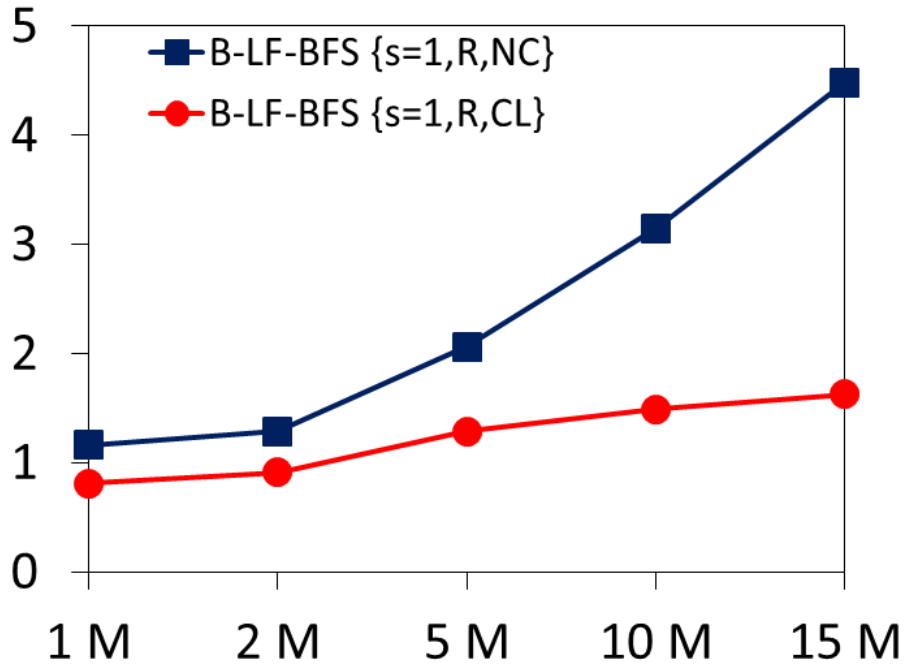


(a) B-LF-BFS, different RDBMSs.

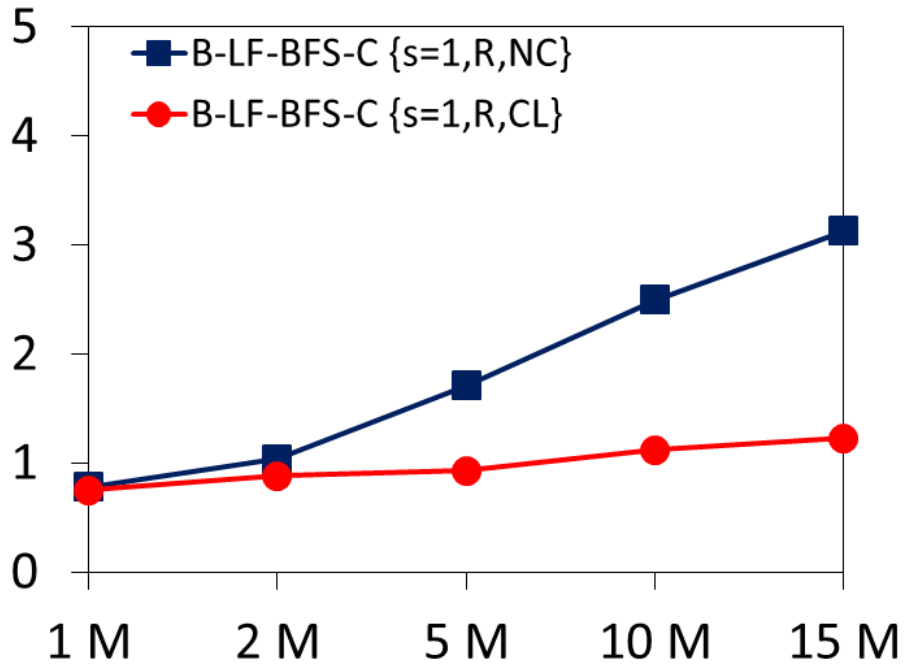


(b) B-LF-BFS-C, different RDBMSs.

Figure 2.9: Experimental results for synthetic datasets. All values on the vertical axes are times in *seconds*.



(c) B-LF-BFS, non-cluster. and cluster. ind.



(d) B-LF-BFS-C with non-cluster and cluster. ind.

Figure 2.9: Experimental results for synthetic datasets. All values on the vertical axes are times in *seconds*.

third)?

Q4 What is the processing time trend as the size of the dataset grows?

Q5 What is the relative cost of database operations?

Q6 Is there a notable difference in the particular RDBMS chosen for this problem?

Answers.

Q1. In figures 2.4a and 2.7a, we show the running times of the algorithms under their best parameter setup. For B-R-BFS, the datasets are partitioned into 5, 10, and 15 tables (p=5, p=10, p=15), respectively, and for B-LF-BFS and B-LF-BFS-C, $step$ is set to 1,2 and 3 (s=1,s=2,s=3), respectively. Fig. 2.4a shows the running times for the real datasets, whereas Fig. 2.7a for the synthetic ones (all obtained using D2).

We see that B-LF-BFS outperforms B-R-BFS on all the datasets, with the difference being more pronounced for the random graphs. Recall though that our main contribution in B-LF-BFS is simplicity over B-R-BFS both in terms of algorithmic design as well as termination detection. The fact that B-LF-BFS performs better than B-R-BFS shows that we achieved simplicity without sacrificing performance.

B-LF-BFS-C outperforms both B-LF-BFS and B-R-BFS, and the difference becomes quite significant for Live Journal. This behavior of B-LF-BFS-C is due to the fact that many fewer disk blocks are needed to store the TE table using the compression presented in Section 2.6. Therefore, there are less disk blocks to read to perform the main join. To see the compression achieved, please refer to Fig. 2.6d that shows the sizes of original and compressed TE tables

for various datasets (using D2, the best performing RDBMS). The compression is quite significant for all the datasets, and for some, such as Arabic 2005 and UK 2005, it is by a factor of more than 20. This compression ratio shows that our compression is quite efficient; it also shows that there is a blowup factor when storing data uncompressed in a database. For example the CSV edge file of UK 2005 is 38 GB, which is less than half the size of the TE table (78 GB) for the same dataset. A similar blowup has also been observed in other works, e.g. [34].

Regarding UK 2002, Arabic 2005, and UK 2005, B-LF-BFS-C is the only one to be able to handle them (see Fig. 2.4b). In fact, B-LF-BFS-C does on UK 2005 significantly better (by more than 27%) than what the other two algorithms can do for Live Journal, which is an order of magnitude smaller than UK 2005. B-R-BFS and B-LF-BFS were not able to handle the three largest datasets in our machine in a reasonable time; for instance, it took B-LF-BFS about two hours to compute a single s-t query on UK 2002.

We observe that the average time for B-LF-BFS-C is only 3.7 seconds on Live Journal, and 12.4 seconds on UK 2005. Fig. 2.6c shows the running time of B-LF-BFS-C on UK 2005 for different buffer allocations. We see there is only mild improvement as the buffer size grows. This applies to all three RDBMSs we used (D1, D2, PG). We discuss performance comparisons among RDBMSs later in this section.

Q2. Regarding the baselines, we show the results in Fig. 2.4c. We compare there the baselines against each other and B-LF-BFS-C. As expected, the in-memory implementation of bidirectional Dijkstra's algorithm (B-D-InMem) is faster than its counterparts in RDBMS (B-D) and Neo4j. Also expected is the

fact that B-LF-BFS-C does quite better than B-D and Neo4j; this is because B-LF-BFS-C benefits from a set-at-a-time approach (see Section 2.3.2 for a discussion). What is quite revealing though is that B-LF-BFS-C is a close contender to B-D-InMem and even outperforms it on Live Journal. This affirms the virtue of set-at-a-time evaluation, intelligent scheduling by the RDBMS, and graph compression in B-LF-BFS-C.

Q3. Now we focus on what the best parameters for the considered algorithms are. We see in Fig 2.5a that $p = 5$ is the best number of partitions for B-R-BFS in the real datasets. This is also confirmed by Fig. 2.8a for synthetic datasets. For the latter, we only show lines for $p = 5$ and $p = 10$ as the numbers for $p = 15$ were too large to be interesting. We explain this behavior of B-R-BFS as follows. While having more partitions (edge tables) helps to potentially achieve termination faster using the early joins (those using low-numbered edge tables), there is nevertheless an added penalty in terms of page scheduling by the RDBMS, if we are to join the same part of the F set with too many edge tables (which happens when the finalization is delayed). In other words, joins become too-small-too-many, and the performance suffers.

In Fig. 2.5b and 2.8b we see the B-LF-BFS performance for different step values on the real and synthetic datasets, respectively. We see that $s = 1$ is the best value for B-LF-BFS. Whereas the differences in running time for $s = 1, 2, 3$ are not big for the real datasets, they become quite noticeable for the synthetic datasets. For the latter, the performance for $s = 1$ is significantly better than for $s = 2, 3$. Recall that the greater the step size, the bigger the F sets become, and the more the B-LF-BFS gets closer to BFS. For B-LF-BFS-C, on the other hand, we find that $s = 3$ is the best value for the three big real datasets (Fig. 2.4b),

whereas $s = 1$ is the best value for the medium real datasets (Fig. 2.5c) and synthetic datasets (Fig. 2.8c). This suggests that some parameter tuneup is needed depending on the graph.

The tuneup of the step size can be performed by randomly selecting a set of source-target pairs as we are doing in our experiments. Then, we test different values for $step$ starting from a value equal to the cost of the lightest edge and incrementing it by this amount each time. What we look for is some value for the step size such that the set-at-the-time evaluation has an opportunity to better schedule disk accesses while not doing too much non-optimal work. As our experiments show, moderate values for the step size give a good balanced evaluation.

Q4. In Fig. 2.7a, we see that the curve for B-R-BFS, even for $p = 5$, grows faster compared to B-LF-BFS and B-LF-BFS-C. This trend for B-R-BFS is more pronounced in Fig. 2.8a for $p = 10$. The curve for B-LF-BFS grows in general mildly, unless its step value is not tuned properly (see Fig. 2.8b). Finally, B-LF-BFS-C has the mildest growing curve of the three algorithms and is somewhat "forgiving" even when s is not tuned to the best value (see Fig. 2.8c).

Q5. The experiments in Fig. 2.10 were conducted using synthetic data sets (random graphs of 1-15 million edges) with $step = 3$. Fig. 2.10a shows the relative time for finding frontier nodes versus and the time taken for executing the join queries in B-LF-BFS. The Merge time was not considered in the figure as it was insignificant. We can see that the processing time is mainly consumed by join queries. Fig. 2.10b shows the relative time for B-LF-BFS-C to decode compressed tuples in table TE, find frontier nodes, and execute join queries. We can see that decoding did not consume the significant portion of the whole

processing time. It is still the join time that dominates.

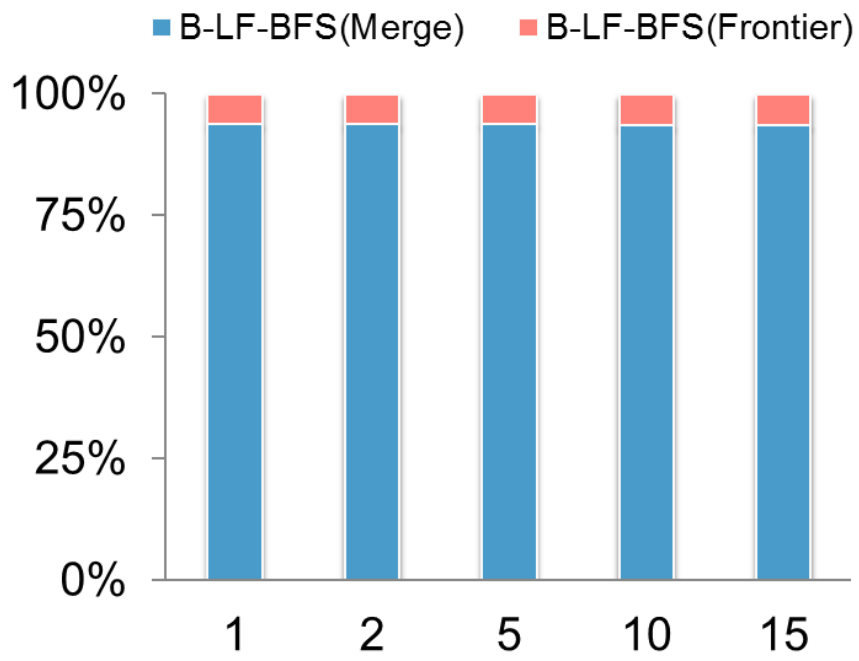
Q6. Figures 2.5d and 2.9b show the performance of different RDBMSs when running B-LF-BFS-C (best algorithm). We allocated the same amount of buffer space, 6G, and created the same index setup for all three of RDBMSs we used. We see that D1 and Postgres performed similarly, however, D2 significantly outperformed both of them. A similar behavior can also be observed when running B-LF-BFS (see Fig. 2.9a).

To better see the performance of B-LF-BFS-C on different RDBMSs, we also give figures 2.6a (for D1) and 2.6b (for Postgres), which should be compared with Fig 2.4b (for D2).

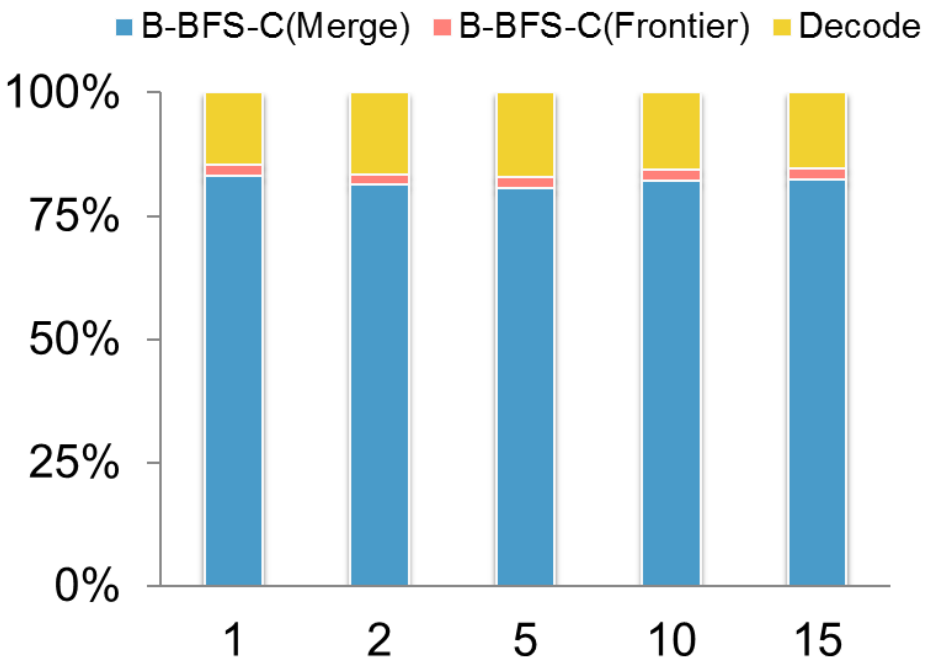
These results suggest that even though well-developed RDBMSs (such as those we consider) are close competitors in well-known benchmarks, when it comes to handling specialized workloads (such as graph operations), they show noticeable differences.

2.8 Related Work

Computing s-t queries on large complex networks has received considerable attention from the research community (cf. [1, 5, 36] and [17, 26, 52, 54, 67, 68, 71]). The first and second group of works compute exact and approximate answers, respectively. As we provide exact answers to s-t queries, our work is more related to the first group. The works in the first group achieve very good scalability, but focus on unweighted graphs, which is an easier problem to tackle. In unweighted graphs, the length of a path amounts to the number of its edges. Since social and web graphs have typically a small diameter, computing



(a) B-LF-BFS, Merge and Frontier



(b) B-BFS-C, Merge, Frontier, Decode

Figure 2.10: Relative cost of different database operations.

s-t shortest paths on unweighted graphs does not need to travel more than few hops. On the other hand, if edge weights are taken into consideration, the small diameter of the graphs is not that important anymore as shortest paths can contain an arbitrary number of edges, hence, many more expansions are needed. In this part, in contrast with the aforementioned works, we focus on weighted graphs, which are more general and challenging to handle. While we do not show experimental results for unweighted graphs, we mention that we tested extensively with such graphs in RDBMS and our performance was considerably better than for weighted graphs.

Bidirectional computation for finding shortest paths from a source to a target has been suggested in several works (cf. [23, 29]. In [29], it is shown that proper termination is when both the forward and backward search finalize a graph vertex in common. Recall, this was possible for the bidirectional Dijkstra's algorithm and B-LF-BFS, but not for B-R-BFS (for which we need to resort to a more complex termination procedure). In [23], the benefits of bidirectional search are shown by experiments for graphs that fit in memory. Another part of [23] is about A* heuristics for speeding up the s-t shortest path computation. Such heuristics, however, were in practice observed in [23] to be mainly useful for spatial networks (and not much for complex networks). In this part, we focus on complex networks (social and web networks). As such, we have not employed A* heuristics.

Relational Databases have been often used to store complex data, such as XML and RDF graphs (cf. [14, 56] and [13, 28], respectively). They have also been shown to be a good choice to support advanced applications, such as data mining [62, 80] and machine learning [22, 38].

For graph queries, as [27, 72] argue, relational technology can sometimes outperform more specialized solutions. An interesting work that uses relational technology for answering subgraph and supergraph queries is [58]. These queries are different from our source-to-target shortest path queries. Other works, such as [20, 37] use relational databases to build vertex-centric (VC) systems in the Pregel model. As we explained in Section 2.2, a VC computation is not a good fit for computing s-t shortest paths. In terms of table structure, the edge tables used in B-R-BFS and B-LF-BFS are similar to the edge table in [20]. All these works (including ours) suggest that using relational databases for graph management can be for some problems better than using specialized graph engines.

2.9 Conclusions

We showed that designing algorithms for RDBMS is a good avenue to pursue for graphs that do not fit in memory, and sometimes, even for graphs that can (e.g. Live Journal). Also, RDBMS technology is quite mature and can accommodate complex data and algorithms, sometimes, even better than special purpose systems.

We presented a correct procedure for deciding termination in B-R-BFS. We argued that it is challenging to determine whether a vertex has its distance finalized in B-R-BFS. Then we showed that we can decide termination by carefully deriving lower bounds on the forward and backward distances of vertices from the source and target.

We gave next a new algorithm, B-LF-BFS, which performs restrictive BFS by

selecting only a part of the visited vertices as a set to be expanded. This was achieved by setting distance levels that need to be cleared (in terms of expansions) before going to the next level. We showed that, once a level is cleared, the vertices that were expanded in that level have their distance finalized. This allowed us to use a much simplified termination procedure.

Then, we presented B-LF-BFS-C, an algorithm that enhances B-LF-BFS by using a compressed representation of neighbor-cost lists. The compression achieved was such that B-LF-BFS-C was able to handle graphs of an order of magnitude bigger than what B-R-BFS and B-LF-BFS could.

Using detailed experiments, we showed that all three algorithms scale well (for their best parameter setup), and B-LF-BFS-C in particular can produce results in a reasonable time even on the largest dataset we experimented with, UK 2005, with close to one billion edges, using only a consumer-grade machine.

Chapter 3

PageRank for Billion-Scale Networks in RDBMS

3.1 Introduction

With the amount of data produced daily, one of the main challenges facing big data is filtering out data and identifying the wheat from the precious. As search results tend to be in millions of pages, ranking search results becomes very crucial, however ranking pages tends to be one of the most difficult problems as the search engine is required to present a very small subset of results and order them by relevance. A variety of ranking features such as page content or hyperlink structure of the web are used by Internet search engines to come up with a good ranking. Many algorithms have been proposed to sort query results and return the most relevant pages first. Among them are PageRank [48], HillTop [10] and Hypertext Induced Topic Selection (HITS) algorithms [12].

PageRank [11, 48] developed by the Google founders is based on the hyperlink structure and on the assumption that high ranked pages usually contain

links to useful pages, therefore giving more weight to pages that have more inbound links from high weighted paged. PageRank is an iterative algorithm and executed on the whole graph, which, in Google's case, is very large. Algorithms such as PageRank for big data sets require extensive hardware setups and, in many cases, distributed computing, such as Hadoop [79] and Spark [77]. This is because we cannot fit the whole data set in one machine's memory. Building such a setup comes with significant investment and continuous running costs.

Nevertheless, some of the existing systems such as Relational Database Management Systems (RDBMS's) are still widely used and will not be deprecated in the future as the amount of investments on them is growing over the years. More specifically, RDBMS's have been around for over half-century [16] and proven to provide consistent performance, stability, and concurrency control. RDBMS's are currently the backbone of the IT industry and have been evolving over the past few decades for better performance.

On the other hand, using dedicated graph databases for graph processing is presumed to provide better performance and scalability over relational databases (c.f. [6]), however, graph databases still have a long way to reach the level of maturity of RDMBS's. From this perspective, using an RDBMS to implement graph algorithms seems logical and in fact more efficient. However computing graph algorithms using SQL queries is challenging and requires novel thinking. This chapter presents a PageRank algorithm implementation using RDBMS with table partitioning and compares it with the implementation provided by a dedicated Graph Database.

The rest of this chapter is organized as follows. A brief background review of the PageRank algorithm is given in Section 3.2. Our PageRank implementation

using RDBMS with table partitioning is given in Section 3.3. Section 3.4 shows the results of the experiments. Section 3.5 concludes the chapter.

3.2 Preliminaries

In this chapter, different from before, we consider graphs to be unweighted. Specifically, we denote by $G = (V, E)$ a graph with V as a set of vertices, and E as the set of edges. For each vertex v there will be a non-negative initial PageRank value $PR(v)$ and for each edge $e = (u, v)$ there is a weight of $1/n$ assigned to it, where n is the number of outgoing edges (links) from u.

We can use the following relational tables to store a graph. The TE table contains all the edges $e = (u, v) \in E$ along with their weights, where u is denoted by *fid*, and v by *tid* and $w(u, v)$ by *rank*. So, the schema of the table is *TE(fid,tid,rank)*. We construct a unique index on (*fid*,*tid*).

The TV table contains all vertices $u \in V$ in the graph, denoted by *id* along with its rank $PR(u)$, denoted by *pgrank*. So the schema of the table is *TV(id,pgrank)*. We construct a unique index on *id*.

The PageRank algorithm assigns a weight value to each page in the web or vertex in a graph; the higher the weight of a page or vertex, the more important it is. Web pages are represented as a directed graph where pages are vertices and links are edges. Below is an example of how we calculate PageRank for a small graph.

The graph in Figure 3.1 has four vertices representing four web pages. Page 1 has links to each of the other three pages; page 2 has links to 1 and 3 only; page 0 has a link only to 1, and page 3 has links to 2 and 0 only. Let us assume

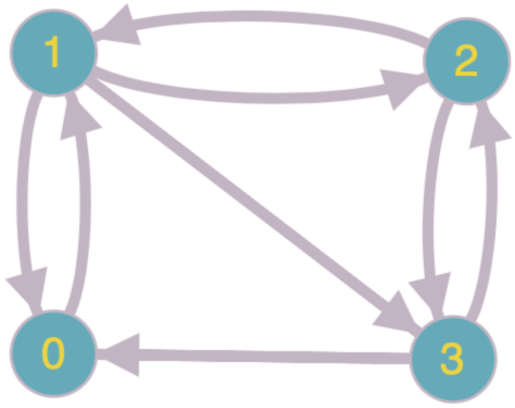


Figure 3.1: Simple Directed Graph

a random user is visiting page 1; this user will have a probability of 1/3 for each link (0,2,3) to follow and visit a next page. If the user is visiting page 0, then he will have a probability of 1 to visit page 1 as this is the only link available. If we follow the same logic, we will have a probability of 1/2 for each link on page 2 and page 3. The probability value for each link is the weight for each link, and based on this, we could build an adjacency matrix for the graph as a square matrix M with a number n of columns and rows. The PageRank algorithm proceeds in the following steps.

- Set an initial PageRank value for each page
- Repeat until convergence: compute PageRank using Equation 3.1

$$PR(A) = \sum_{i=1}^{n} \frac{PR(i)}{C(i)} \tag{3.1}$$

where $PR(A)$ is the PageRank value of vertex A, $PR(i)$ is the PageRank value of vertex i, and $C(i)$ is the number of outbound links (edges) of vertex i. Vertices i for $i \in [1, n]$ are all the vertices of the graph that contain links pointing to A. Usually, there is also a damping factor present in the computation of PageRank values but we ignore it in this research for simplicity and because all techniques

we present can be extended easily to that case.

The link probabilities ($1/C(i)$), as described above, could be represented as a matrix M. For the graph in Figure 3.1, the matrix will be as follows.

$$M = \begin{bmatrix} 0 & 1/3 & 0 & 1/2 \\ 1 & 0 & 1/2 & 0 \\ 0 & 1/3 & 0 & 1/2 \\ 0 & 1/3 & 1/2 & 0 \end{bmatrix}$$

Regarding the $PR(i)$ values, we can represent them all by a PageRank vector V. Then the computation given by Equation 3.1 can be written as $M \cdot V$, which captures the computation of PR values for all the vertices of the graph at the same time. We denote by V_t the version of V at iteration t. Then, PageRank is iteratively computed using equation 3.2, by multiplying matrix M and vector V_t and repeating until convergence.

$$V_{t+1} = M \cdot V_t \tag{3.2}$$

where V_{t+1} is the new vector holding the newly computed PageRanks for all the vertices. In each iteration, the newly computed PageRank values will get closer to the final PageRank values. We stop when PageRank values do not change much.

Observe that the PageRank value of a vertex A is dependent on the value of PageRank of vertices pointing to it. However, we do not know the Pagerank value of inbound vertices till we calculate the ones pointing to them and we will not know the Pagerank values of them till we calculate the PageRank values of

vertices pointing to them too and this keeps on. So, to overcome this starting problem, we initially set an estimated PageRank value for each vertex. This can be represented as a vector

$$V_0 = [1/n, 1/n, \cdots, 1/n]$$

where n is the number of vertices in the graph.

3.3 PageRank in RDBMS

Representing the graph in a square matrix, M requires quadratic size. Computing Pagerank in its matrix representation requires the matrix to be fully loaded in memory; however, loading the graph into memory might not be possible for large graphs like the Google web or Facebook. However, since the matrix is very sparse, all the implementations exploit sparsness and do not materialize the matrix as is. Instead only the non-zero entries are stored in the format (i, j, m_{ij}).

Using RDBMS is quite efficient in this regard. First, matrix M could be saved as tuples (i, j, m_{ij}) of only connected vertices (this is the TE table with its three columns). Second, when computing PageRank for a vertex A, the edges that need to be considered are only those pointing to vertex A. This is a tiny subset of the matrix.

Figure 3.2 shows the SQL statement used to compute Pagerank using equation 3.2, where TE stores graph edges and TV stores vertices's Pagerank estimates. If we run this SQL query, it will produce the result of multiplying the matrix with the vector V_i. The multiplication is very efficient as we only do the

calculation for existing edges in the matrix.

```
SELECT a.tid, SUM(a.rank*b.pgrank)
        From TE a, TV b
        WHERE a.fid=b.node
        GROUP BY a.tid;
```

Figure 3.2: Compute Pagerank For one Iteration

Figure 3.3 shows the full SQL statement using the new Merge SQL [19] operation, which is very efficient in saving SQL results. This way, we save the new Pagerank estimate so that it can be used in the next iteration. The query will do a full table scan or index scan based on the table setup. RDBMS will need to load parts of the table into memory to compute Pagerank. This process is acceptable when the loaded parts could be loaded into memory but cumbersome when graph size is hugely larger than available memory, which inevitably will lead to use data swap and, as a result, diminish the performance dramatically. In the following section, we solve the graph size problem by using table partitioning based on partitioning of matrix M (table TE) and vector V_i (table TV) into parts that can be loaded into memory. We would have liked for the RDBMS to do the partitioning itself, but in reality we observed that this does not happen, so we do that as part of the algorithm we implement.

```
MERGE  INTO TV as Target
        USING
         (SELECT a.tid, SUM(a.rank*b.pgrank)
                FROM TE a, TV b
                WHERE a.fid=b.node
                GROUP BY a.tid )
        AS  Source(node,pgrank)
        ON(target.node=source.node)
        WHEN MATCHED THEN
                UPDATE SET pgrank=source.pgrank
        WHEN NOT MATCHED THEN
                INSERET(node,pgrank)
```

Figure 3.3: Compute Pagerank and update vector V

3.3.1 Table Partitioning

To overcome the matrix size problem, we partitioned both the matrix and the vector into k parts and saved each part in a separate table, TV_i, and TE_i, where $i \in [1, k]$. We divide the matrix into stripes of almost equal size, and we create vectors to have only the vertices that are needed to compute Pagerank for each matrix stripe.

Figure 3.4 shows how the matrix and vector are partitioned. Each matrix stripe will have a full set of inbound edges for a set of vertices and matched with a vector containing all the *fid*'s that exist in the partitioned matrix. This way, we will be able to compute Pagerank for the set of vertices of interest. A similar matrix partitioning scheme is also described in the Map-Reduce paper of [41].

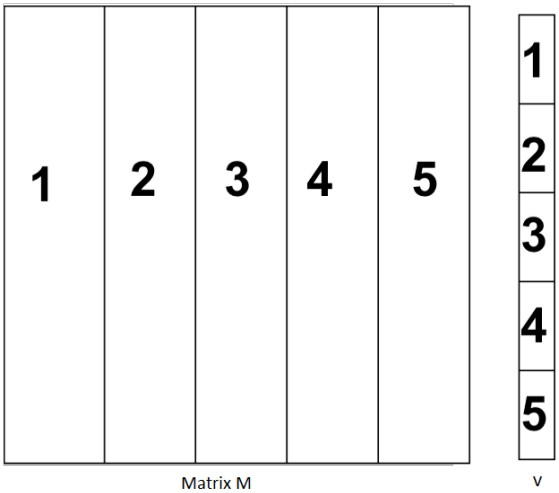


Figure 3.4: Matrix and vector partitioning into k stripes.

The main goal is to create as many stripes as needed so that the portions of the matrix in one partition can fit conveniently into memory. We used the SQL statements in Figure 3.5 to build the partitioned tables based on matrix partitions. Each TE_i table will have a subset of vertices along with all inbound edges, and each table TV_i will have all *fid*'s that exist in TE_i.

```
--Create TE_i table
INSERT into TE_i (fid,tid,rank)
SELET fid, tid, rank FROM TE
where tid >=<val> AND tid < <val2>
--Create TV_i table
INSERT INTO TV_i(node,pgrank)
SELET DISTINCT  fid node, d FROM TE_i
```

Figure 3.5: Creating partition tables TV_i and TE_i for $i \in [1, k]$.

3.4 Experimental Results

3.4.1 Setup Configurations

We executed the experiments on a consumer-grade server with Intel Core i7-2600 CPU @3.4 GHz 64 bit Processor, 12 G of RAM and running Windows 7 Home Premium, using Java JDK SE 1.8.

As RDBMS's we used the latest versions of a commercial database (which we anonymously call CD) and an open-source database (which we anonymously call OD). As graph database, we used the latest version of a graph database (which we anonymously call GD). We refrain from using the real names of these databases for obvious reasons.

We used four real datasets from Stanford's Data collection and a one-billion-edge graph from The Laboratory for Web Algorithmics. By default, we used three table partitions in the case of table partitioning experiments except stated otherwise. All the results shown are based on computing one Pagerank iteration. The real datasets are Web-Google, Pokec, Live-Journal and Orkut (from `http://snap.stanford.edu`), and UK 2005 (from `http://law.di.unimi.it/webdata`). Table 1 shows statistics about the datasets used.

Table 3.1: Page Rank Datasets

Data Set	Nodes#	Edges#
Web-Google	875,713	5,105,039
Pokec	1,632,803	30,622,564
Live Journal	4,847,571	68,993,773
Orkut	3,072,441	117,185,083
UK 2005	39,459,921	936,364,282
IT 2004	41,291,594	1,150,725,436

3.4.2 Results

We observed that in all the datasets we used, OD and CD clearly out-perform GD significantly. Figure 3.6 shows how GD performs poorly with large datasets, such as Live Journal (LJ) or Orkut. Orkut was the largest dataset that GD could manage to process without crashing out.

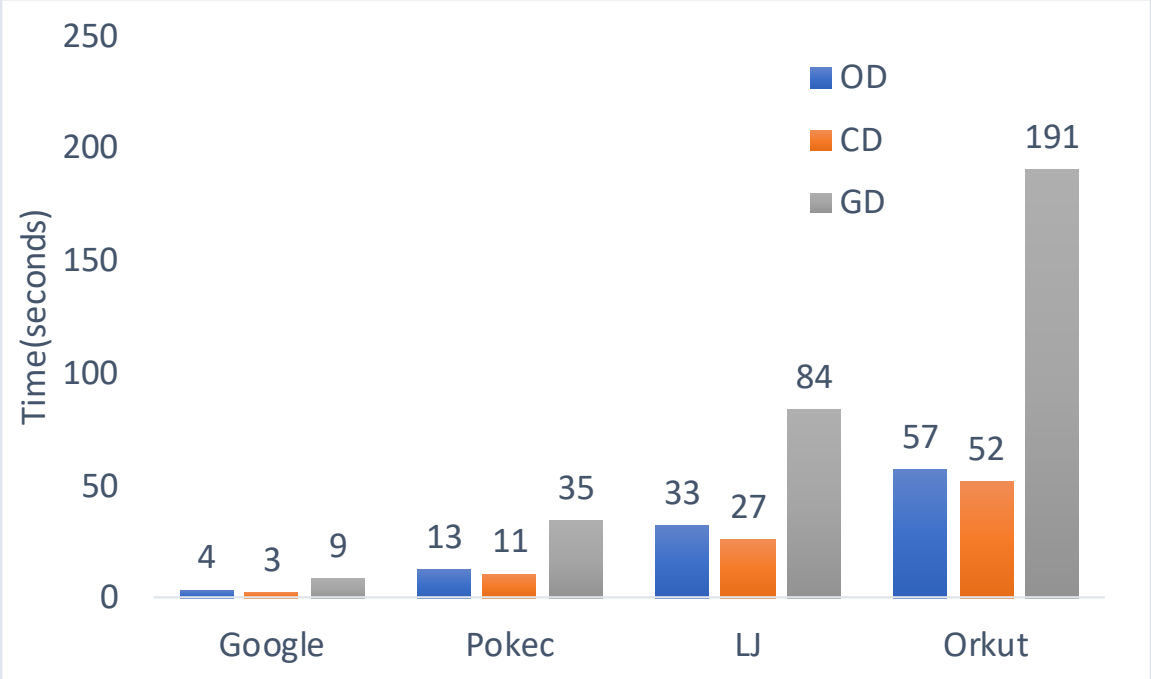


Figure 3.6: Results of running PageRank using GD, CD, and OD.

Using table partitioning gives significant enhancement in managing memory load which in turn boosts PageRank processing time especially with large

data sets such as LJ, Orkut and the large graph UK-2005. Figure 3.7 shows significant performance differences between the GD processing time and both RDBMS approaches using table scan and table partitioning. The impact of table partitioning starts to appear once the datasets become larger, as shown in the chart. Table partitioning significantly improved over the approach of table scan especially for LJ and Orkut.

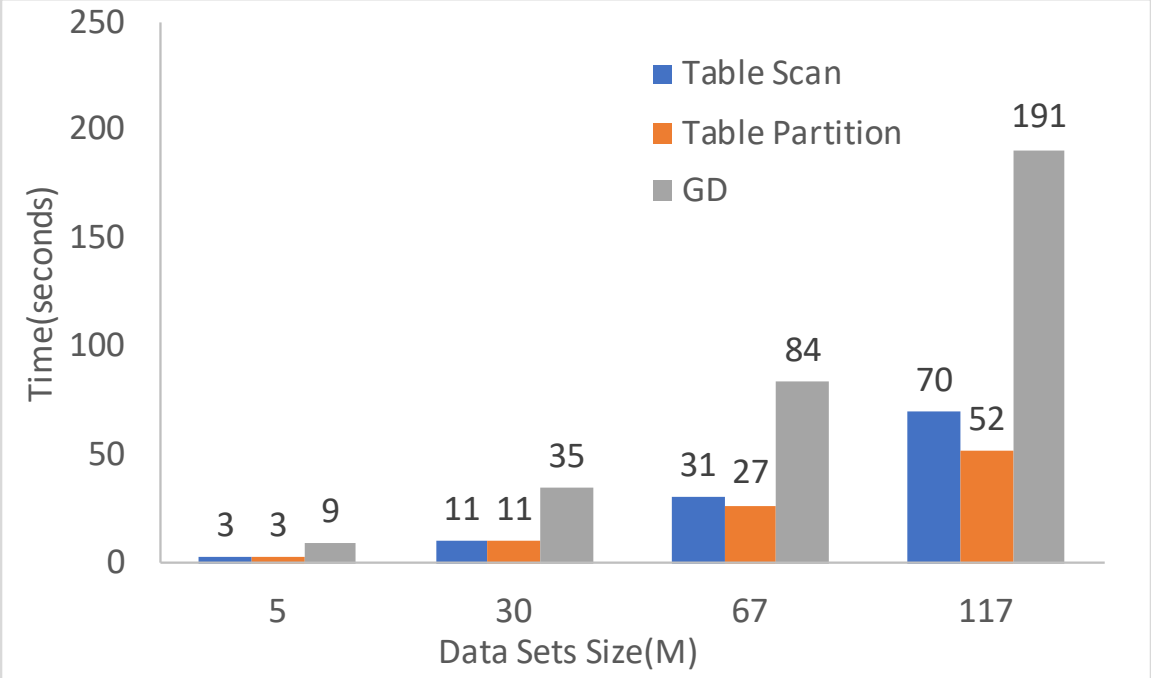


Figure 3.7: Results of PageRank in RDBMS CD using Table Scan, Table Partitioning and GD. We show here only the dataset sizes as opposed to their names. The names are as in Figure 3.6.

In our experiments we also wanted to decouple the processing time of computing PageRank from the time to save the results, hence we ran two separate experiments; one with saving the outcome and the other without saving the outcome. Figure 3.8 shows the results of these experiments. We noticed that CD did a better job than OD in both operations and the time taken for saving data was noticeably shorter for CD compared to OD. We relate this to the Merge

operation which exists in CD but does not in OD. The Merge operation showed to have superior performance over regular insert/update operations.

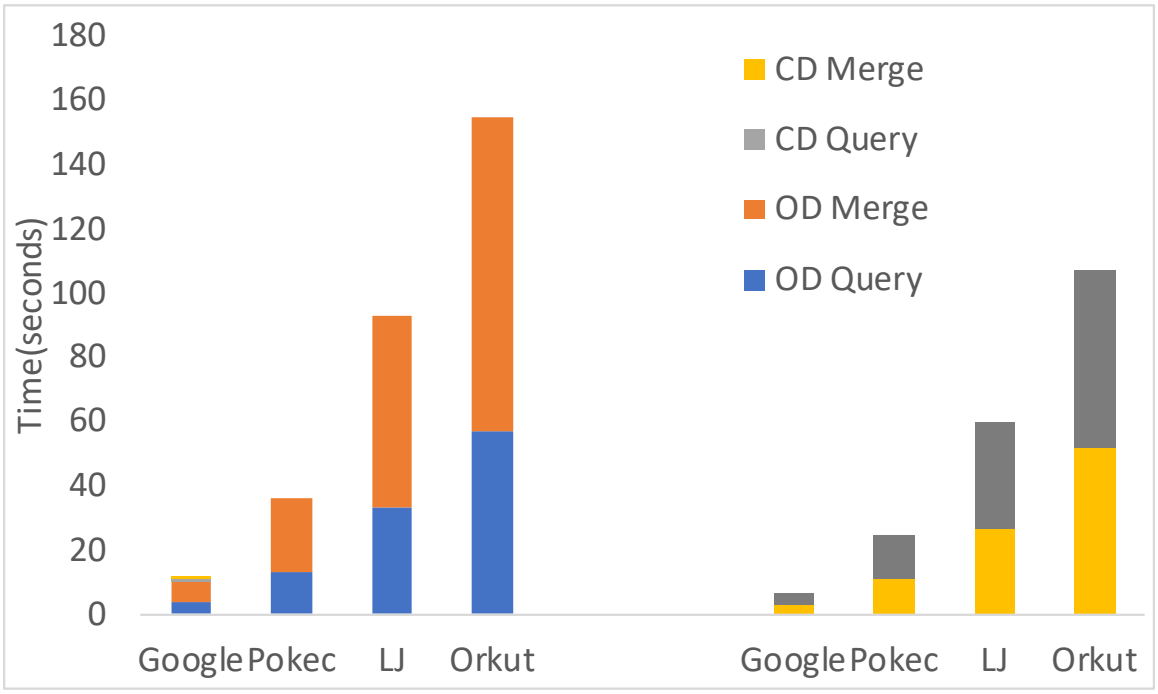


Figure 3.8: Show the difference between the time taken to only calculate PageRank without saving the results and the time taken to do the same with saving the results.

In addition to the above, OD performs poorly in computing PageRank using a non-clustered index scan. Figure 3.9 shows a big jump in time when we used non-clustered index scan in large data sets, in contrast to a clustered index scan or table scan. We relate this to the OD optimizer not being good enough in planning and executing the queries. More specifically, the query optimizer should have chosen a plan that uses table scan rather than the index. Also the I/O cost was high which indicates the data retrieval process included high random access. Such random access was reduced significantly when the table was reordered as part of building the clustered index, hence the processing time was also reduced significantly.

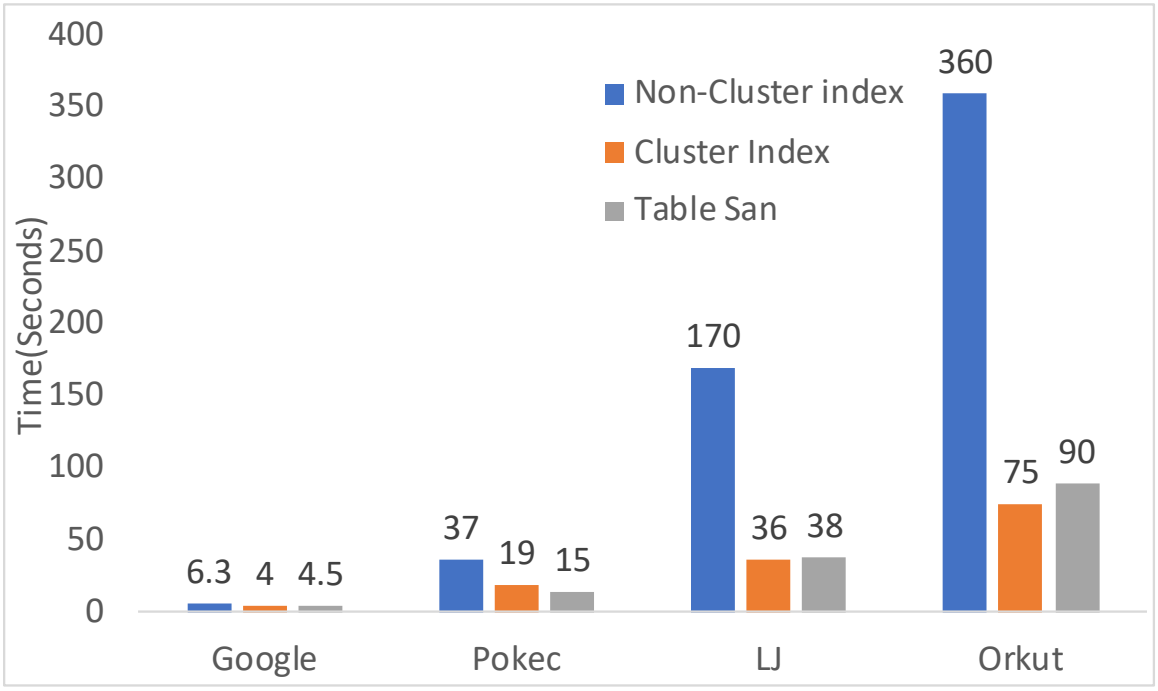


Figure 3.9: OD performs poorly in the case of non-clustered index vs table scan or clustered index.

Figure 3.10 shows the processing time in the case of using table scan and clustered index scan. Using an index did not help that much in reducing processing time and the results were very comparable to just table scan and differences were not noticeable. We relate this to the fact that the query used to calculate PageRank requires a full table retrieval hence using an index will not make that big of a difference.

3.4.3 Experiments on Billion-Scale Networks

Here we show our experiments on two very large datasets, namely UK-2005 and IT-2004, the latter with more than a billion edges. They represent the web network of UK and Italy in 2005 and 2004. The precise number of nodes and edges is given in Table 2. We ran the PageRank algorithm using table

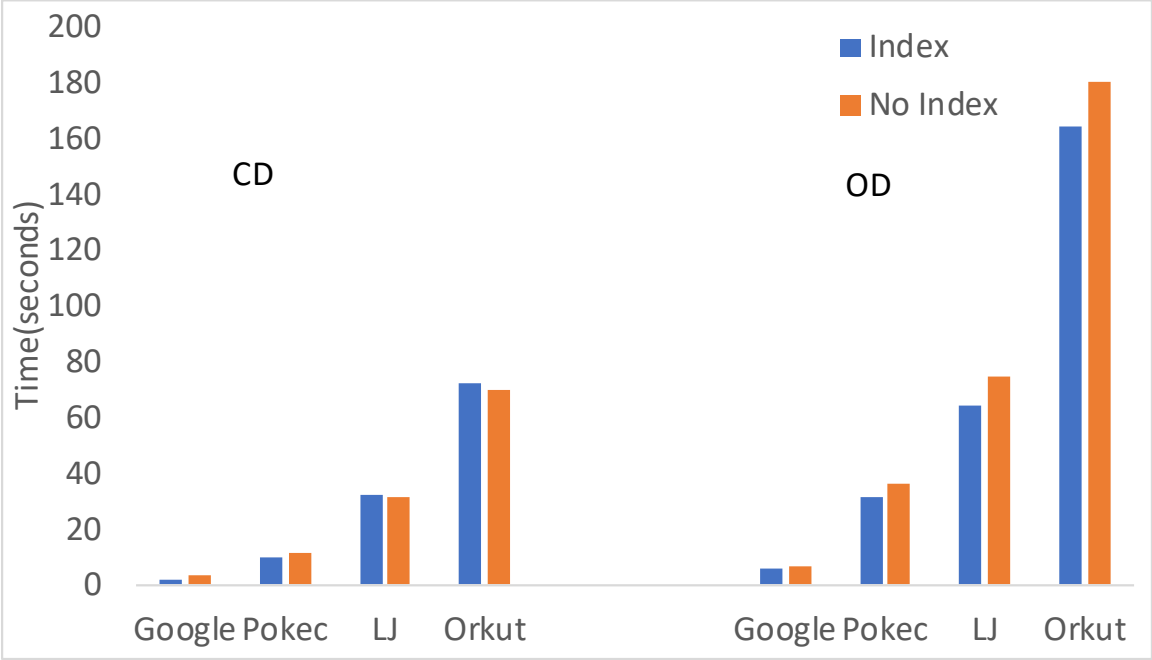


Figure 3.10: Results of using table scan vs index scan in OD and CD

partitioning. Both data sets were partitioned into 12 partitions and we sum up all the processing time to compute PageRank for each partition.

Figure 3.11 shows the runtime for each of the datasets. We used CD as it showed superiority over OD in I/O and memory management. GD could not be a part of the experiment as it failed to process any graph bigger than Orkut in our test environment setup. As Figure 3.11 shows, even with over one billion-edge graph size, we managed to get a good processing time. More specifically, for IT-2004, we were able to complete the computation of PageRank for all the partitions in about 10 min (600 sec).

Table 3.2: Page Rank Billion-size Datasets

Data Set	**Nodes#**	**Edges#**
IT 2004	41,291,594	1,150,725,436
UK 2005	39,459,921	936,364,282

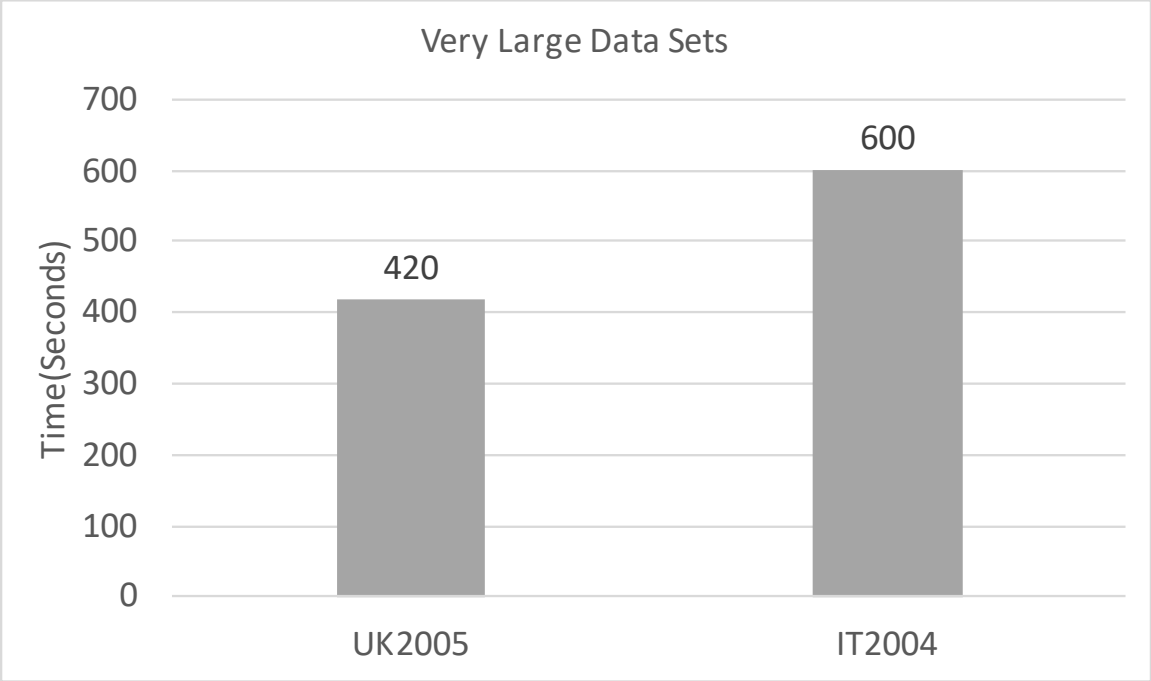


Figure 3.11: Results of calculating PageRank on very large data sets, IT 2004: 1.15 billion edges and UK 2005:0.93 billion edges.

3.5 Conclusions

We presented the implementation of the PageRank algorithm over RDBMS using different options, such as table-scan, non-clustered-index, clustered-index, and table-partitioning. We showed that RDBMSes could perform better than GD and could process very big datasets in a consumer-grade machine.

The experiments showed that the OD optimizer was not good enough for our task. For instance, it was not able to determine that using a non-clustered index is not a good choice for our queries. The OD clustered index behaved better but still there was no improvement compared to simple table-scan without any indexing at all. The CD commercial query optimizer is more intelligent than its open-source counterpart.

We observed that manually partitioning tables gives a significant improve-

ment in the execution time of our queries. This tells us that RDBMS optimizers of today, even after many decades of development, still can be improved further in order to handle heavy analytical queries such as those computing PageRank.

CD did not consume significant processing time in order to merge the data but OD in large datasets consumed significant processing time, in some cases double the query time. We clearly observed that both RDBMSes we use, without using any indexing or partitioning still dramatically outperform graph database GD. This comes as a surprise because the latter was designed for handling graphs from the ground up. Therefore we conclude that specialized graph databases still have a lot of ground to cover in order to be good competitors to RDBMS engines for large datasets when executing heavy analytical tasks.

Chapter 4

Triangle Enumeration for Billion-Scale Graphs in RDBMS

4.1 Introduction

The problem of triangle listing or triangle enumeration can be stated as follows: Given an undirected graph $G = (V, E)$ with no parallel edges or self loops, output all tuples (a, b, c) such that nodes $a, b, c \in V$ are pairwise connected in G (i.e., they form a triangle). Some algorithms only need to touch triangles or count them, but in this research, we require that algorithms store them explicitly (hence the term triangle listing).

Triangle enumeration is considered a fundamental graph analytics problem that constitutes a large portion of the computational work required for problems such as calculating a graph's global clustering coefficient (in which the number of triangles each node is involved in is required) [69], finding k-truss decomposition, and calculating the transitivity ratio of a graph. Some of the indirect applications of triangle enumeration include identifying social networks,

determining a community's age [40], performing community searches [9, 55], and detecting fake accounts, malicious pages, or instances of web spamming [50, 8]. Triangles are also used as a building block for enumerating higher order structures [59, 60, 73].

Triangle enumeration is a nontrivial problem. The optimal worst case for any algorithm's time complexity is $O(|E|^{3/2})$ [50]. Triangle enumeration does not scale very well with large datasets as any node can participate in a triangle with any other node in a graph so long as they have an edge between them, making it difficult to break the problem into smaller pieces. The sizes of datasets continue to grow, and massive datasets are becoming more common in business and research. In particular, data in real-world networks is growing to unprecedented levels [35]. Many algorithms either enumerate triangles for graphs that can fit in memory [c.f. [61, 78]], are I.O. intensive [c.f. [32, 15]] or use distributed systems such as MapReduce [c.f [51, 69]].

Relational Database Management Systems (RDBMSs) have been vital tools in storing and manipulating data for many decades [16]. They are commonplace in most businesses, and they are familiar to technical and non-technical users alike. This research aims to show that they are also useful for analyzing large graphs, specifically in the area of triangle enumeration. The research also shows how a single machine can use partitioning techniques to enumerate billions of triangles efficiently.

In recent times, native Graph Databases (GDBs) have grown in popularity. Many businesses may be considering moving to GDBs if they often analyze large graphs. Using dedicated graph databases for graph processing is presumed to provide better performance and scalability over relational databases (c.f. [6]);

however, as we demonstrated for the previous two problems, graph databases still have a long way to go in order to reach the level of maturity of RDMBS's. Using an RDBMS to implement graph algorithms is, in many situations, more efficient. However, designing graph algorithms using SQL queries is challenging and requires novel thinking. This is especially true for triangle enumeration, which is a computationally challenging problem. Again we will show that using RDBMS's provides higher efficiency over graph databases and this is quite pronounced for the problem of triangle enumeration.

Our contributions in this part are as follows:

1. We engineer triangle enumeration algorithms in SQL using partitioning and coloring.

2. We propose a modification of the distributed algorithm PTE CD of [50], which we name Source Node Partitioning, that allows us to scale efficiently to very large graphs.

3. We compare the performance of a popular open source GDB and commonly used RDBMS's.

4. We give a comparison of the performance of each algorithm on several graphs including billion-scale graphs.

4.2 Triangle Enumeration in RDBMS

The following section introduces state of the art triangle enumeration algorithms and techniques. The basic method used for triangle enumeration, known as the Compact Forward algorithm [75], is explained first. Adaptations created

to handle larger graphs that cannot fit into a single machine's memory are then discussed. These adaptations are known as Triangle Type Partitioning [49], and Pre-partitioned Triangle Enumeration [50]. Finally, a further adaptation contributed by this research that aims to reduce the complexity and the number of queries generated, named Source Node Partitioning, is explained. The SQL implementation of each algorithm developed is illustrated so that readers may recreate these results in an RDBMS of their choice.

4.2.1 Compact Forward

The compact forward algorithm constitutes the main portion of work done to enumerate triangles in all of the algorithms that follow. Its pseudocode is shown as Algorithm 7. It consists of two main parts: an edge iterator and an orientation technique.

Algorithm 7 Compact Forward

Input: `Undirected graph G = (V, E)`

Output: `All triangles of G`

`//Orientation of G`

for Each $(u, v) \in E$ **do**

  if $deg(u) > deg(v)$ OR $(deg(u) = deg(v)$ AND $u > v)$ **then**

    Replace (u, v) with (v, u) in E

  end if

end for

`//List triangles of G`

for Each $(u, v) \in E$ **do**

  for $w \in N(u) \cap N(v)$ **do**

    Output triangle (u, v, w)

  end for

end for

Edge Iterator. For any edge $(u, v) \in E$, we can find the triangles associated with it by considering the intersection of the neighbors of u and v (denoted $N(u)$ and $N(v)$ respectively). That is, if (u, v) exists, we can check to see if both (u, w) and (v, w) exist for all such nodes $w \in V$. By performing this operation on all edges, we are guaranteed to enumerate all triangles in the graph. Unfortunately, we might count many duplicates depending on how the undirected graph G is represented.

Orientation Technique. In order to eliminate duplicates, we direct the graph G with a total ordering. Define the degree (or number of neighbouring

nodes) of a node v to be $deg(v)$ and assume nodes are represented by unique integer identifiers (i.e. $V \subset \mathbb{N}$). Define a total ordering of the nodes in V, denoted $\rightarrow$, as follows: For all nodes $u, v \in V$, we say $u \rightarrow v$ if and only if $deg(u) < deg(v)$ OR ($deg(u) = deg(v)$ AND $u < v$). We then arrange all the edges in the graph according to this total order. The resulting graph is a Directed Acyclic Graph or DAG. But how does this help?

Consider a triangle (a, b, c), such that $a \rightarrow b, b \rightarrow c$, and $a \rightarrow c$. When iterating over edge (a, b), we discover (b, c) and (a, c) in the neighbours of a and b. However, when we iterate over edge (b, c), we will not find (c, a) due to the total ordering properties (similarly for edge (a, c), we will not find (c, b)). Thus each triangle is counted exactly once. From now on, when we refer to triangle (a, b, c), we assume the total ordering applies from left to right.

Notice that a total ordering could have been defined on the node identifiers alone assuming they are unique. The reason for involving node degrees in the total ordering is to prevent any given node from having a large list of outgoing neighbours to search through. In the given total ordering, nodes of high degree will have fewer outgoing neighbours and nodes of low degree, though their neighbourhood primarily consists of outgoing neighbours, have few neighbours by definition.

The SQL queries for orienting G are lengthy, yet simple to implement, so we omit them. Algorithm 8 shows the edge iteration component written in SQL and assumes the edge list E has already been oriented.

Algorithm 8 Compact Forward Edge Iteration in SQL

```
SELECT g1.fromNode AS A, g1.toNode AS B, g2.toNode as C
FROM E g1, E g2
WHERE g1.toNode = g2.fromNode
AND EXISTS (
SELECT 1 FROM E
WHERE fromNode = g1.fromNode AND toNode = g2.toNode)
```

4.2.2 Triangle Type Partitioning

Suppose G is a large graph that does not fit in a single machine's memory, then the edge list E must be partitioned into smaller lists in order to fit. Even though an RDBMS is designed to handle data that does not fit in main memory, when it has to deal with smaller chunks of the data at a time, the RDBMS can use more efficient join algorithms, such as one-pass joins. However, a problem arises when trying to list triangles in the edge partitions. Consider triangles that have edges in more than one partition, they are certainly missed. As such, we cannot expect the RDBMS query optimizer to be able to automatically find ways to break up the data into chunks to facilitate better query evaluation algorithms. Therefore, we focus here in ways to intelligently partition the data and create independent subtasks.

The triangle type partitioning (TTP) algorithm of [50] was designed to resolve this issue. Originally it was designed as a Map Reduce algorithm to be run on distributed systems[1], but in this research, we re-purpose it be used on a single machine.

[1] See also [76] for an implementation on Amazon AWS Lambda.

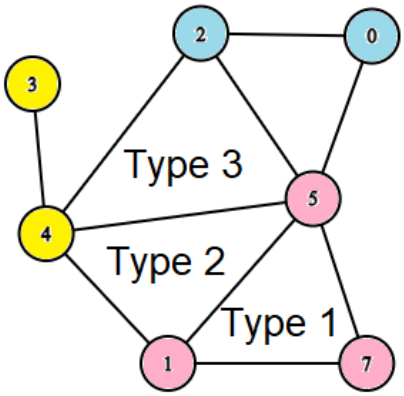


Figure 4.1: An example of each type of triangle in a node coloured graph

The first step is to colour the nodes using a function $f : V \to \{0, 1, \ldots, \rho - 1\}$ that maps each of the graph nodes to one of the ρ colors. This allows every triangle to be classified into the following three types:

- Type 1: All three nodes have the same colour.
- Type 2: Exactly two nodes have the same colour.
- Type 3: Each node has a distinct colour.

A visualization of the triangle types can be seen in Figure 4.1.

Let E_{ij} be the set of edges that have endpoints coloured i or j. There are $\binom{\rho}{2}$ such sets. Let E_{ijk} be the set of edges that have endpoints coloured i, j, or k. There are $\binom{\rho}{3}$ such sets as seen in Figure 4.2.

No edge in a type 3 triangle has endpoints with the same colour. Knowing this, we can transform the set E_{ijk} into $E'_{ijk} = E_{ijk} - \{(u, v) | f(u) = f(v), (u, v) \in E_{ijk}\}$ where we remove all such edges. This vastly improves performance when enumerating type 3 triangles by reducing the size of the edge lists. On the other hand, each set E_{ij} can contain type 1 and type 2 triangles.

In order to enumerate all triangles, Algorithm 8 is run on all $\binom{\rho}{2} + \binom{\rho}{3}$ edge sets (replace E with E_{ij} or E'_{ijk} for all i, j, and k). This can be seen in Algorithm 9, which shows the structure for partitioning and generating the SQL queries.

Algorithm 9 Triangle Type Partitioning

```
for i = 0 to ρ − 2 do
    for j = i + 1 to ρ − 1 do
        Generate E_ij from E
        Run Algorithm 8 on E_ij
    end for
end for
for i = 0 to ρ − 3 do
    for j = i + 1 to ρ − 2 do
        for k = j + 1 to ρ − 1 do
            Generate E'_ijk from E
            Run Algorithm 8 on E'_ijk
        end for
    end for
end for
Eliminate duplicates from generated triangles
```

Consider the sets E_{02} and E_{12} from Figure 4.2. The type 1 triangle of colour 2 is counted twice. In fact, this is true of all type 1 triangles.

Thus we count every type 1 triangle $\rho - 1$ times, and we count type 2 and type 3 triangles exactly once. The duplicate triangles can then be eliminated.

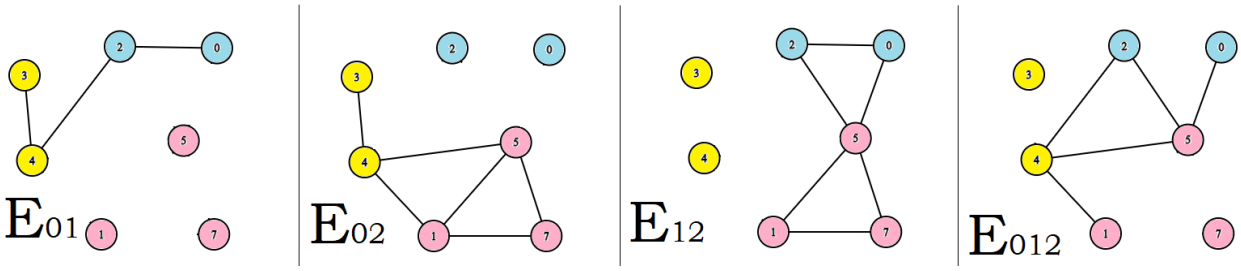


Figure 4.2: A visualization of the edge sets from the graph in figure 2 when $\rho = 3$: yellow = 0, blue = 1, pink = 2

4.2.3 Prepartitioned Triangle Enumeration - Colour Direction

Triangle type partitioning [50] is an effective method for enumerating triangles in massive graphs that do not fit into memory, yet it is also possible to ensure each triangle is counted exactly once as well as greatly reduce the size of the edge sets to be searched through in the CF algorithm. This is the goal of the Prepartitioned Triangle Enumeration - Colour Direction (PTE CD) method, which expands on the TTP algorithm. Again, this algorithm was proposed in [50] for a Map Reduce setting. Here we adapt it for an RDBMS. Consider how the TTP algorithm pays no attention to the direction of edges when partitioning the edge set, indeed Figures 4.1 and 4.2 do not display the direction of the edges because that information is irrelevant to the algorithm. PTE CD, however, takes advantage of the direction of each edge in the oriented graph.

Let T_{ijk} be the set of triangles (a, b, c) where $f(a) = i, f(b) = j$, and $f(c) = k$ (i.e. ijk is a permutation of the colours with replacement). For example, $T_{001} \neq T_{010}$ due to the total ordering on the edges in G. This is made more clear in Figure 4.3. There are ρ^3 such triangle sets.

Suppose we are trying to find a triangle $(a, b, c) \in T_{ijk}$ and we reach edge

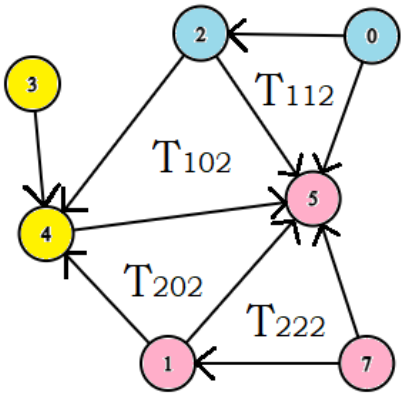


Figure 4.3: The graph of figure 4.1 with edge directions and triangles labels.

(a, b). We know edge (b, c) is in E_{jk} and edge (a, c) is in E_{ik}, which may or may not all be the same edge set from our previous definitions, but more specifically, we know (b, c) goes from colour j to k, and (a, c) goes from colour i to k.

Let E_{ij} be redefined in the following way:

$$E_{ij} = \{(u, v) | f(u) = i, f(v) = j, (u, v) \in E\}$$

There are ρ^2 such edge sets. This new definition allows the algorithm to more precisely choose which edge sets to search through and thereby reduces the total work.

The pseudo code in Algorithm 10 shows the implementation of the PTE CD algorithm. It counts triangles of all types exactly once and improves the performance greatly.

One issue is that it increases the number of edge sets from $\binom{\rho}{2}$ to ρ^2 and the number of enumeration tasks from $\binom{\rho}{2} + \binom{\rho}{3}$ to ρ^3. However, it reduces the number of colours needed to be effective compared to TTP, and since ρ is often relatively small, this does not create a major issue in performance.

Algorithm 10 PTE CD

for $i = 0$ **to** $\rho - 1$ **do**

for $j = 0$ **to** $\rho - 1$ **do**

Generate E_{ij} from E

end for

end for

for $i = 0$ **to** $\rho - 1$ **do**

for $j = 0$ **to** $\rho - 1$ **do**

for $k = 0$ **to** $\rho - 1$ **do**

Run Algorithm 8 with E_{ij} as g1, E_{jk} as g2, and E_{ik} in the EXISTS clause.

end for

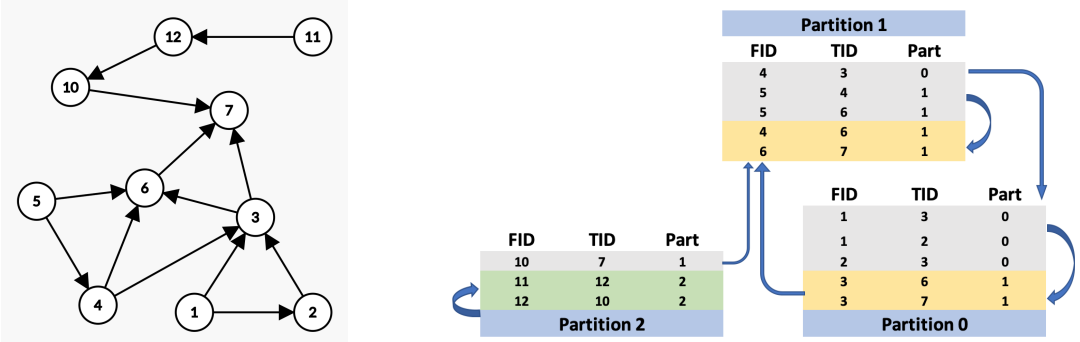
end for

end for

4.2.4 Source Node Partitioning (SNP)

In this section, we propose another algorithm, called Source Node Partitioning (SNP), that partitions the graph into ρ partitions and generates ρ^2 enumeration tasks. SNP is conceptually simpler than PTE CD. Similar to PTE CD, SNP enumerates each triangle only once and exhibits comparable performance to PTE CD for medium datasets and even better for larger datasets.

The main idea is to partition an oriented input graph G into ρ parts where all the edges (u, v) whose source nodes u are the same are placed in the same partition i based on a partitioning function $f(u)$. Partition i will contain all the edges (u, v), where $f(u) = i$. In addition to the edge endpoints, u and v, we also



(a) Input Graph (b) Partitioning into tables

Figure 4.4: SNP partitioning

store along the edge the partition number of v. So, instead of the pair (u, v) we will have a triple (u, v, j) where j is equal to the partition number for node v. Specifically, the i-th partition is as follows

$$E_i = \{(u, v, j) | f(u) = i, f(v) = j, (u, v) \in E\},\ 0 \leq i, j \leq \rho - 1.$$

Now, in order to find a triangle (u, v, k) we only need to check if there is a shared node k between the neighbours of nodes u and v. This is achieved by joining in SQL the tables for E_i and E_j (assuming $f(u) = i$ and $f(v) = j$).

For example, given the graph in Figure 4.4 (a), and $\rho = 3$, we partition the edges of the graph into the partition tables given in Figure 4.4 (b). Along with the FID and TID attributes, which give the source and target of each edge, we also have a third attribute, called Part, which stores the partition number, of the target node.

To check for instance, if edge $(3, 6)$ is part of a triangle, we need to find the intersection of the neighbors of node 3 with the neighbors of node 6. These neighbors can be extracted from the set of edges $(6, \_)$, which exist in Partition 1

(as indicated by column Part) and the set of edges $(3, \_)$, which exist in the current partition, Partition 0.

Partition oriented graph G based on the source node can be done by several ways, for instance, we could use interval partitioning where each partition i will have a unique range of node ID or use a modulo function to distribute the source nodes over partitions as shown in Algorithm 11.

In terms of pseudo code we create the partition tables using the queries given in Algorithm 11. Then, we enumerate triangles using Algorithm 12. In a nutshell, it joins two tables, for partitions E_i and E_j, respectively, and limits the scope of the join in the table for E_i to only those nodes that have $Part = j$.

Algorithm 11 SNP: Graph Partitioning

Input: Oriented Graph $G = (V, E)$, number of partitions ρ

Output: A set of ρ partition tables.

for $i = 0$ **to** $\rho - 1$ **do** INSERT INTO E_i

SELECT FID, TID, TID % ρ AS Part

FROM E

WHERE FID % $\rho = i$

end for

Algorithm 12 SNP (Enumerate Triangles)

Input: A set of ρ partition tables E_i.

Output: All Triangles (u, v, w).

for $i = 0$ **to** $\rho - 1$ **do**

for $j = 0$ **to** $\rho - 1$ **do**

SELECT G1.fid AS A, G1.tid AS B, G2.tid as C

FROM E_i AS G1, E_j AS G2

WHERE G1.tid = G2.fid AND G1.part = j AND

EXISTS

(SELECT 1 FROM E_i AS G3

WHERE G3.fid = G1.fid AND G3.tid =G2.tid)

end for

end for

Theorem 4. SNP Partitioned graph can only distribute a triangle (u, v, k) over a maximum of 2 partitions.

Proof. To prove it by contradiction let us assume there is a triangle (u, v, k) whose edges exist in three partitions which means neighbors of u and v in an edge (u, v) must exist in three partitions. However, SNP will not separate the edges with the same source node across different partitions, therefore neighbors of u and v will only exist in a maximum of 2 partitions. ■

4.3 Experimental Results

4.3.1 Setup Configuration

We executed the experiments on a cloud based virtual server running Windows Server 2019 with 4 vCores and 16 GB of RAM.

As RDBMS we used the latest version of a commercial database (which we anonymously call CD). As a native graph database, we used the latest version of a graph database (which we anonymously call GD). We refrain from using the real names of these databases for obvious reasons.

4.3.2 Datasets

We used four datasets from Stanford's Data collection and four datasets from The Laboratory for Web Algorithmics including a one-billion-edge graph.

The datasets are Web-Google, Pokec, Live-Journal and Orkut (from `http://snap.stanford.edu`), and Hollywood 2009, Hollywood 2011, UK 2005 and IT 2004 (from `http://law.di.unimi.it/webdata`). Table 1 shows statistics about the datasets used.

Table 4.1: Triangle Enumeration Datasets

Dataset	# Nodes	# Edges	# Triangles
Web-Google	875,713	5,105,039	13,391,903
Pokec	1,632,803	30,622,564	32,557,458
Live Journal	4,847,571	68,993,773	177,820,130
Orkut	3,072,441	117,185,083	627,584,181
Hollywood 2009	1,139,905	113,891,327	4,916,374,555
Hollywood 2011	2,180,759	228,985,632	7,073,951,555
UK 2005	39,459,921	936,364,282	21,779,347,099
IT 2004	41,291,594	1,150,725,436	47,249,138,589

4.3.3 Results

Our experiments begin with a comparison of the performance of the popular graph database (GD) and the well-known, commonly used RDBMS (CD).

The initial idea was to compare the time it took for each database system to compute the clustering coefficient of each graph. However, after running various experiments, it was determined that the graph database likely used approximation algorithms. This theory was then supported when attempting to use the graph database's triangle listing algorithm which took substantially longer than the clustering coefficient calculation, which does not make logical sense if no approximation algorithms are involved because triangle listing is a subset of the computations required for calculating the clustering coefficient. Moreover, the triangle listing algorithm seemed to have a bug at the time of experimentation, and only listed around a tenth of the triangles in the graph.

In order to ensure the measurements for the graph database were as fair as possible, we caused the output to be written to a file instead of writing to standard out (which can often take longer than a computation itself), and we scaled the triangle listing time linearly to match the approximate amount of time it would take to enumerate all of the triangles rather than a fraction of them.

Figure 4.5 shows a comparison between the projected runtime of the graph database and the actual runtime of the RDBMS baseline, which is the compact forward algorithm run on the entire edge set at once. Note that the time scale is logarithmic and we can see that the RDBMS greatly outperforms the graph database when the graph database is required to list all triangles explicitly.

After seeing the drastic performance difference between the two databases

Figure 4.5: Runtime comparison (in log scale) of a graph database and RDBMS when enumerating triangles on the four smallest datasets: google, pokec, live-journal, and orkut.

in just the baseline case, we did not see value in pursuing any further comparisons with other methods or larger datasets.

Figure 4.6 compares the performance of all the algorithms discussed above on the Orkut dataset. Smaller datasets did not see much improvement when partitioned, as their edge sets were already quite small. No entry has been given for the PTE BASE method (which is another name for TTP) when $\rho = 2$. This is because the algorithm no longer benefits from the partitioning in this case, and the results would be the same as the baseline CF results.

Note that the yellow bar is the same datapoint that was used in Figure 4.5 for the RDBMS baseline on the Orkut dataset. Clearly the partitioning algorithms made great improvements over the single table compact forward algorithm. From this we conclude that partitioning can be helpful even when the dataset is

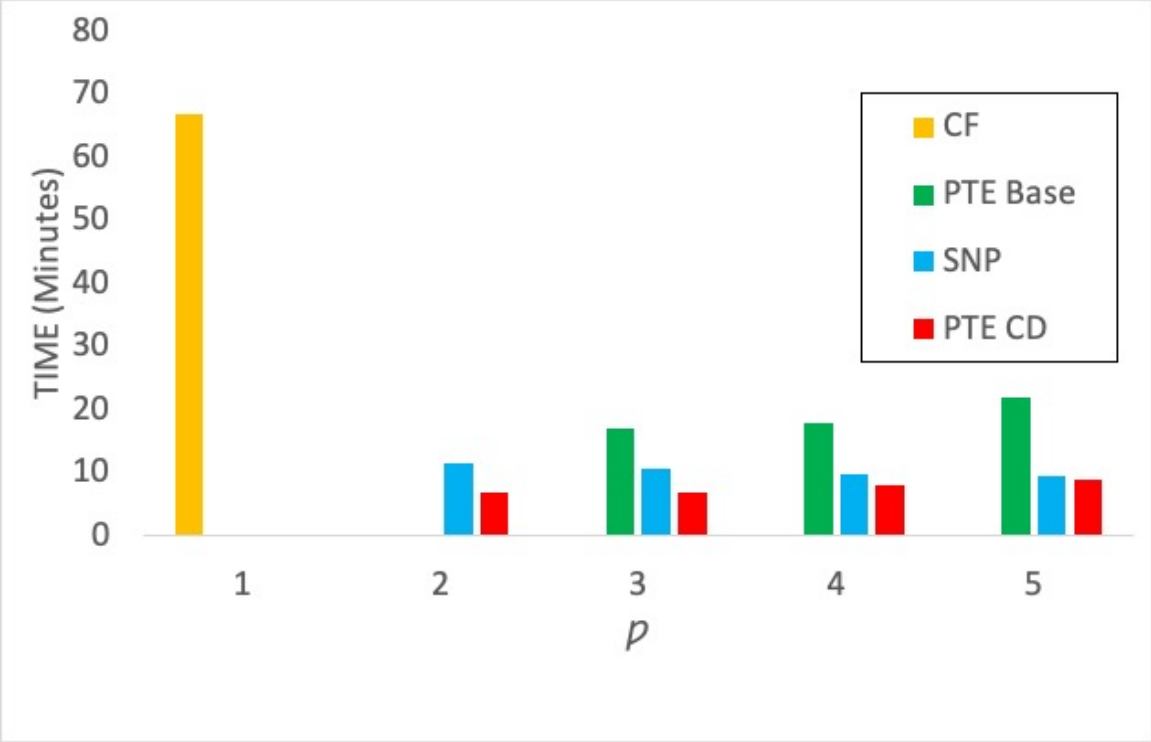


Figure 4.6: A comparison of runtimes of the four algorithms discussed above on the Orkut dataset. ρ is the number of node partitions of the graph, not the number of edge sets created by the algorithms.

able to fit into a machine's memory (as is the case with the orkut dataset). However, as noted before, the benefits become less notable the smaller the dataset becomes and in some cases the performance decreases slightly as the cost of partitioning tables and query planning overcome the savings. As we moved to graphs with billions of triangles, PTE Base reached its performance bottleneck; eliminating duplicate triangles takes a considerable amount of time and resources to perform, therefore, we decided not to test it with other datasets.

We turn our attention to the hollywood datasets for 2009 and 2011 which have over 8 and 11 times more triangles than orkut respectively. Figure 4.7 shows that PTE CD slightly outperforms the SNP method. Notice, however, that the performance gap begins to close with more node partitions or colours.

At first inspection, this seems to be due to the growth rates in the number of queries generated by each method. SNP generates ρ^2 queries compared to PTE CD's ρ^3 queries, which conceivably increases compiling and execution time as well as clutters the SQL script files.

However, the difference is not as clear cut as it may seem. SNP only creates ρ edge sets (or edge partitions), whereas PTE CD creates ρ^2 edge sets. If the desired edge set size is the same relative to the size of a machine's memory (e.g. we want edge sets to be a third the size of a machine's available memory), then PTE CD can use a smaller value for ρ than SNP can. Therefore PTE CD may actually use less queries in practice.

For example, suppose a graph has an edge list with 1.2 million entries, and we desire to use a machine that can fit 300,000 edges, then our edge sets should have about 100,000 edges in them (if we want the database to perform an all-in-memory join). SNP will use $\rho = 12$ to meet this requirement, whereas PTE CD will use $\rho = 4$. Notice that $4^3 = 64 < 12^2 = 144$ in this instance, and PTE CD actually has fewer queries for similar-sized edge sets. Nevertheless, in a very large graph, fewer queries might not be as desired. SQL might need in worst-case scenario memory size of $|E_i| * |E_j| * |E_k|$ to find triangles; hence more scoped queries would be more efficient. Also, the partitioning function used in SNP impacts the performance; Figure 4.9 shows a noticeable difference in performance when we used the modulo function, and this could be explained as it is most likely to distribute high degree nodes across all partitions relatively than distribute them using interval partitioning.

It is also worth mentioning that the distance between squares increases quickly. In the previous example, PTE CD creates 16 edge sets rather than

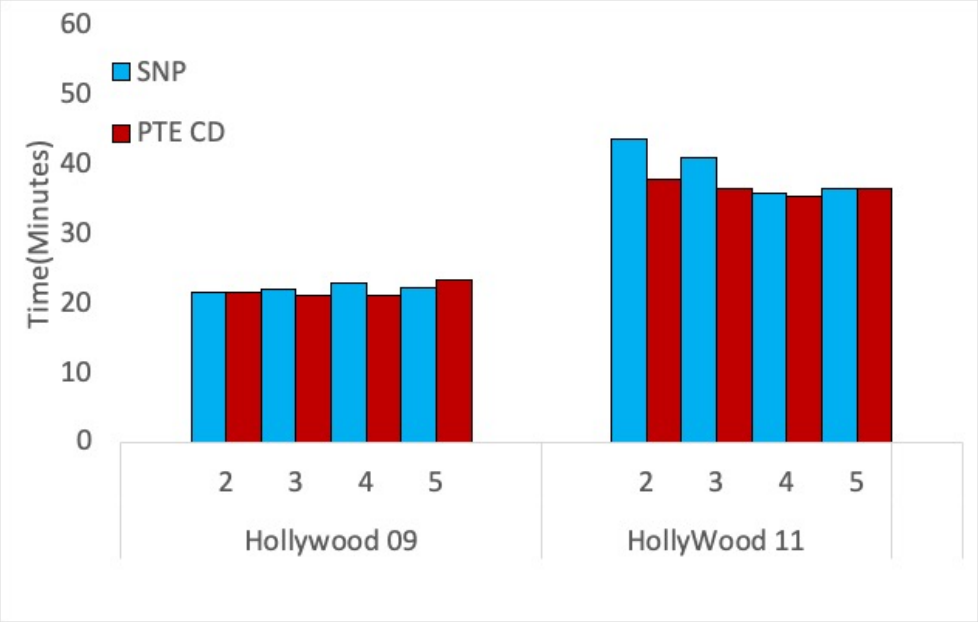


Figure 4.7: Runtimes of SNP (blue) and PTE CD (red) algorithms on the hollywood-2009 dataset ($\sim$ 5 billion triangles) and hollywood-2011 dataset ($\sim$ 7 billion triangles). The x axis shows the value of ρ or the number of node partitions.

the ideal 12. Suppose we need to divide the edge list of the graph into 37 edge sets. Then PTE CD must use $\rho = 7$ and create 49 edge sets, which may cause performance issues because the edge sets are too small compared to the optimal value and the time to shuffle the data will increase. Figure 4.8 shows a decrease in the performance of PTE CD as the number of triangles increase.

It would seem that using the same value of ρ for both methods is in inappropriate comparison, and this would explain the reduction in the performance gap as ρ increases, since SNP performs better for higher values of ρ.

In each case, it may take some testing and experimentation to determining the ideal value for ρ for a given dataset.

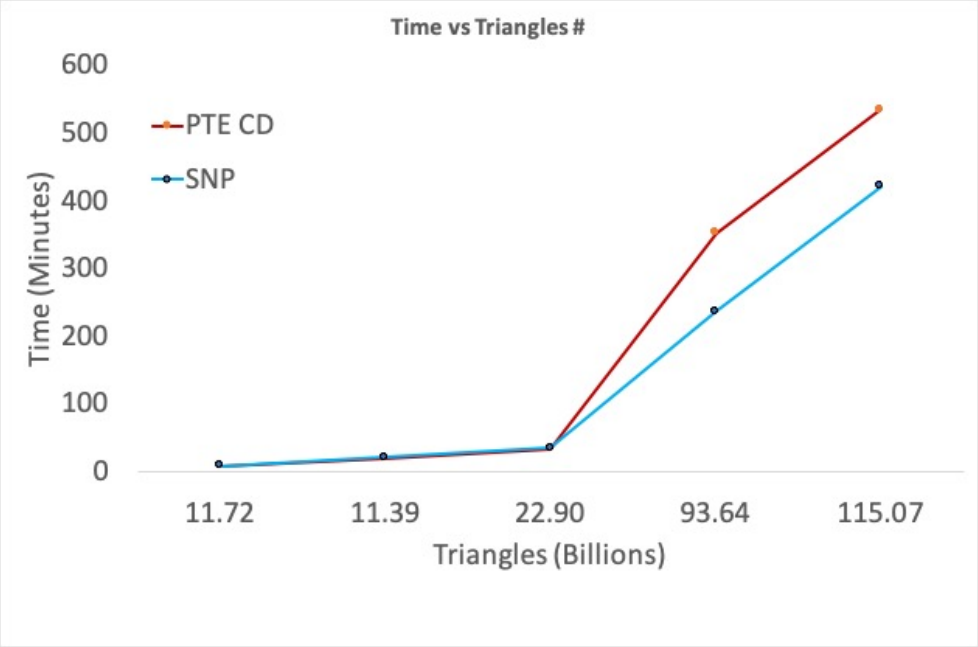


Figure 4.8: A comparison of running time between SNP and PTE CD as the number of triangles increased. PTE CD takes more time as partitioning takes longer than SNP.

4.3.4 Experiments on Billion-Scale Networks

We show our experiments on two vast datasets, namely UK-2005 and IT-2004, the latter with more than a billion edges. They represent the web network of UK and Italy in 2005 and 2004, respectively. The precise number of nodes and edges is given in Table 1. We ran both the SNP algorithm with $\rho = 20$ and PTE CD algorithm with $\rho = 8$. Both algorithms ran in a single machine and enumerated all triangles in a very reasonable time regarding the graph's size and the number of triangles found in each data set; IT 2004 and UK 2005 contain 47.2 and 21.7 billion triangles, respectively.

We compare the running time of the two algorithms in Figure 4.10. The result indicates that the SNP algorithm shows better performance than PTE CD. The reason being that SNP scales linearly with ρ in partitioning the tables and

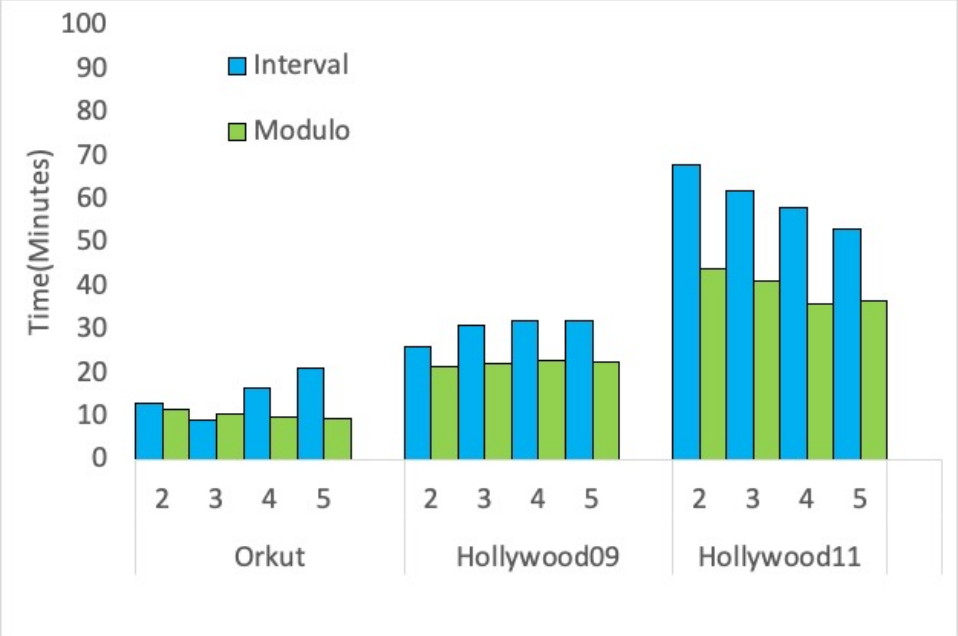


Figure 4.9: SNP: A noticeable difference in running time when partitioning the graph using Interval Partitioning vs using Modulo function.

quadratically in the number of queries when enumerating triangles, however PTE CD scales quadratically to ρ for partitioning tables and cubic to the number of queries performed.

Figure 4.11 shows the running time differences to partition the datasets and to enumerate triangles between SNP and PTE CD algorithms. PTE CD takes significantly more time to create ρ^2 tables and execute ρ^3 queries.

4.4 Conclusions

We implemented triangle enumeration algorithms, such as CF, PTE Base, and PTE CD, using RDBMSs. We proposed the SNP algorithm that partitions the graph into ρ partitions and generates ρ^2 enumeration tasks. SNP exhibits comparable performance to PTE CD for medium datasets and even better for larger

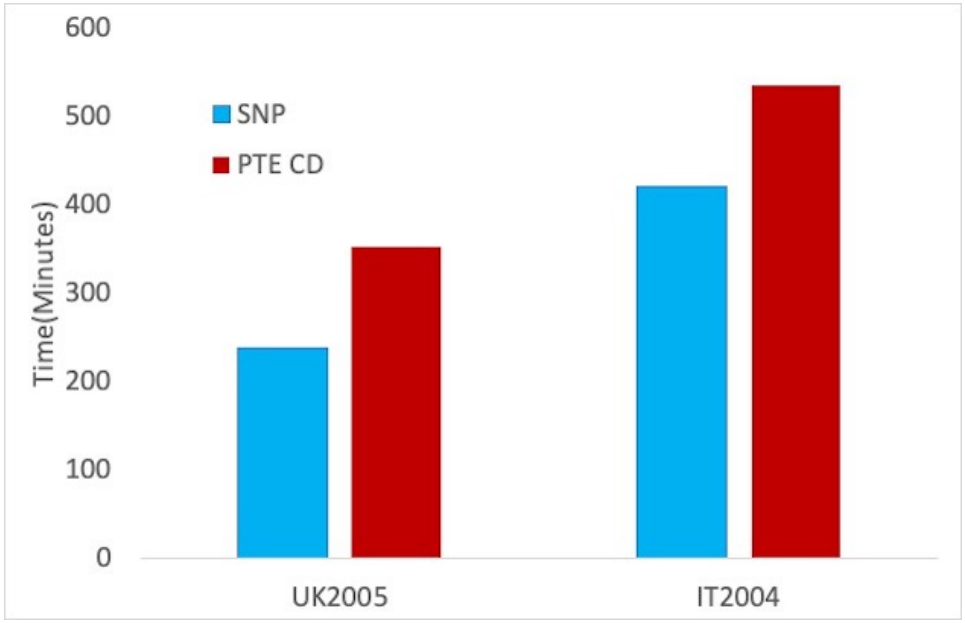


Figure 4.10: Results of triangle enumeration on very large data sets, IT 2004: 1.15 billion edge graph and UK 2005:0.93 billion edge using SNP and PTE CD.

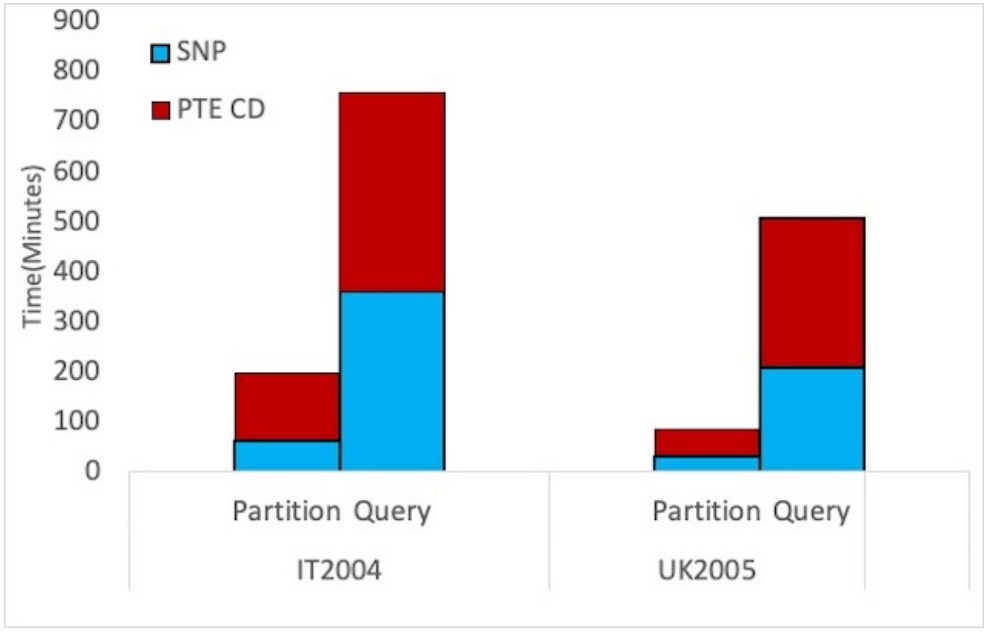


Figure 4.11: The time for SNP to build the partitions and run queries is significantly less when datasets get significantly larger.

datasets. We experimented with billion scale graphs and enumerated tens of billions of triangles, showing that RDBMS's can perform better than GBDs by multiple orders of magnitude and process massive datasets in a consumer-grade server. One possible direction for future research is to improve performance on RDBMS's by compressing the edge list using variable byte encoding (VBE).

Bibliography

[1] Rachit Agarwal, Matthew Caesar, Philip Brighten Godfrey, and Ben Y Zhao. Shortest paths in less than a millisecond. In *ACM Workshop on Online Social Networks*. ACM, 2012.

[2] Aly Ahmed, Keanelek Enns, and Alex Thomo. Triangle enumeration for billion-scale graphs in rdbms. In *AINA (2)*, pages 160–173, 2021.

[3] Aly Ahmed and Alex Thomo. Computing source-to-target shortest paths for complex networks in rdbms. *Journal of Computer and System Sciences*, 89:114–129, 2017.

[4] Aly Ahmed and Alex Thomo. Pagerank for billion-scale networks in rdbms. In *International Conference on Intelligent Networking and Collaborative Systems*, pages 89–100. Springer International Publishing, 2020.

[5] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*. ACM, 2013.

[6] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[7] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[8] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *CIKM*, pages 529–538, 2013.

[9] Jonathan W Berry, Bruce Hendrickson, Randall A LaViolette, and Cynthia A Phillips. Tolerating the community detection resolution limit with edge weighting. *Physical Review E*, 83(5):056119, 2011.

[10] Krishna Bharat and George A Mihaila. When experts agree: using non-affiliated experts to rank popular topics. In *Proceedings of the 10th international conference on World Wide Web*, pages 597–602, 2001.

[11] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. 1998.

[12] Soumen Chakrabarti, Byron E Dom, S Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, Andrew Tomkins, David Gibson, and Jon Kleinberg. Mining the web's link structure. *Computer*, 32(8):60–67, 1999.

[13] Huajun Chen, Zhaohui Wu, Heng Wang, and Yuxin Mao. Rdf/rdfs-based relational database integration. In *ICDE*, pages 94–94. IEEE, 2006.

[14] Liang Jerry Chen, Philip Bernstein, Peter Carlin, Dimitrije Filipovic, Michael Rys, Nikita Shamgunov, James F Terwilliger, Milos Todic, Sasa Tomasevic, Dragan Tomic, et al. Mapping xml to a wide sparse table. *Knowledge and Data Engineering, IEEE Transactions on*, 26(6):1400–1414, 2014.

[15] Shumo Chu and James Cheng. Triangle listing in massive networks. *ACM TKDD*, 6(4):1–32, 2012.

[16] Edgar F Codd. A relational model of data for large shared data banks. In *Software pioneers*, pages 263–294. Springer, 2002.

[17] Atish Das Sarma, Sreenivas Gollapudi, Marc Najork, and Rina Panigrahy. A sketch-based distance oracle for web-scale graphs. In *WSDM*. ACM, 2010.

[18] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[19] Andrew Eisenberg, Jim Melton, Krishna Kulkarni, Jan-Eike Michels, and Fred Zemke. Sql: 2003 has been published. *ACM SIGMoD Record*, 33(1):119–126, 2004.

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

[21] Jun Gao, Jiashuai Zhou, Jeffrey Xu Yu, and Tengjiao Wang. Shortest path computing in relational dbmss. *IEEE Trans. Knowl. Data Eng.*, 26(4):997–1011, 2014.

[22] Wolfgang Gatterbauer, Stephan Günnemann, Danai Koutra, and Christos Faloutsos. Linearized and single-pass belief propagation. *PVLDB*, 8(5):581–592, 2015.

[23] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, pages 156–165, 2005.

[24] Gösta Grahne, Alex Thomo, and William Wadge. Preferentially annotated regular path queries. In *International Conference on Database Theory*, pages 314–328. Springer, 2007.

[25] Gösta Grahne, Alex Thomo, and William W Wadge. Preferential regular path queries. *Fundamenta Informaticae*, 89(2-3):259–288, 2008.

[26] Andrey Gubichev, Srikanta Bedathur, Stephan Seufert, and Gerhard Weikum. Fast and accurate estimation of shortest paths in large graphs. In *CIKM*, pages 499–508. ACM, 2010.

[27] Andrey Gubichev and Manuel Then. Graph pattern matching: Do we have to reinvent the wheel? In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–7. ACM, 2014.

[28] Stephen Harris and Nigel Shadbolt. Sparql query processing with conventional relational database systems. In *Web Information Systems Engineering–WISE 2005 Workshops*, pages 235–244. Springer, 2005.

[29] Martin Holzer, Frank Schulz, Dorothea Wagner, and Thomas Willhalm. Combining speed-up techniques for shortest-path computations. *ACM Journal of Experimental Algorithmics*, 10, 2005.

[30] John E Hopcroft and Robert E Tarjan. Efficient algorithms for graph manipulation. 1971.

[31] Hsun-Ping Hsieh, Cheng-Te Li, and Rui Yan. I see you: Person-of-interest search in social networks. In *SIGIR*, pages 839–842. ACM, 2015.

[32] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. Massive graph triangulation. In *SIGMOD*, pages 325–336, 2013.

[33] Shih-Wen Huang, Daniel Tunkelang, and Karrie Karahalios. The role of network distance in linkedin people search. In *SIGIR*, pages 867–870. ACM, 2014.

[34] Adam Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.

[35] HV Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M Patel, Raghu Ramakrishnan, and Cyrus Shahabi. Big data and its technical challenges. *Communications of the ACM*, 57(7):86–94, 2014.

[36] Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *SIGMOD*, pages 445–456. ACM, 2012.

[37] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. Vertexica: your relational friend for graph analytics! *PVLDB*, 2014.

[38] M Levent Koc and Christopher Ré. Incrementally maintaining classification using an rdbms. *PVLDB*, 4(5):302–313, 2011.

[39] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, volume 12, pages 31–46, 2012.

[40] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining Social-Network Graphs*, page 325–383. Cambridge University Press, 2 edition, 2014.

[41] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive data sets*. Cambridge university press, 2020.

[42] Sandeep Mahanthappa and BR Chandavarkar. Data formats and its research challenges in iot: A survey. In *Evolutionary Computing and Mobile Sustainable Networks*, pages 503–515. Springer, 2020.

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

[44] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge university press, 2008.

[45] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, Angela Hung Byers, et al. *Big data: The next frontier for innovation, competition, and productivity*. McKinsey Global Institute, 2011.

[46] Andrew McAfee, Erik Brynjolfsson, Thomas H Davenport, DJ Patil, and Dominic Barton. Big data: the management revolution. *Harvard business review*, 90(10):60–68, 2012.

[47] Zhuo Miao, Dan Stefanescu, and Alex Thomo. Grid-aware evaluation of regular path queries on spatial networks. In *21st International Conference on Advanced Information Networking and Applications (AINA'07)*, pages 158–165. IEEE, 2007.

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

[49] Ha-Myung Park and Chin-Wan Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *CIKM*, 2013.

[50] Ha-Myung Park, Sung-Hyon Myaeng, and U. Kang. Pte: Enumerating trillion triangles on distributed systems. In *KDD*, 2016.

[51] Ha-Myung Park, Francesco Silvestri, U Kang, and Rasmus Pagh. Mapreduce triangle enumeration with guarantees. In *CIKM*, pages 1739–1748, 2014.

[52] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, pages 867–876. ACM, 2009.

[53] Robert Clay Prim. Shortest connection networks and some generalizations. *Bell system technical journal*, 36(6):1389–1401, 1957.

[54] Miao Qiao, Hong Cheng, Lijun Chang, and Jeffrey Xu Yu. Approximate shortest distance computing: A query-dependent local landmark scheme. *IEEE Trans. Knowl. Data Eng.*, 26(1):55–68, 2014.

[55] Filippo Radicchi, Claudio Castellano, Federico Cecconi, Vittorio Loreto, and Domenico Parisi. Defining and identifying communities in networks. *Proceedings of the National Acad Sciences*, 101(9):2658–2663, 2004.

[56] Michael Rys, Don Chamberlin, and Daniela Florescu. Xml and relational database management systems: the inside story. In *SIGMOD*, pages 945–947. ACM, 2005.

[57] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB Journal*, 29(2):595–618, 2020.

[58] Sherif Sakr and Ghazi Al-Naymat. Efficient relational techniques for processing graph queries. *Journal of Computer Science and Technology*, 25(6):1237–1255, 2010.

[59] Yudi Santoso, Venkatesh Srinivasan, and Alex Thomo. Efficient enumeration of four node graphlets at trillion-scale. In *EDBT*, pages 439–442, 2020.

[60] Yudi Santoso, Alex Thomo, Venkatesh Srinivasan, and Sean Chester. Triad enumeration at trillion-scale using a single commodity machine. In *Advances in Database Technology-EDBT 2019, 22nd International Conference on Extending Database Technology, Lisboa, Portugal, March 26-29, Proceedings*. OpenProceedings. org, 2019.

[61] Thomas Schank. Algorithmic aspects of triangle-based network analysis. 2007.

[62] Xuequn Shang, Kai Uwe Sattler, and Ingolf Geist. Sql based frequent pattern mining without candidate generation. In *SAC*, pages 618–619. ACM, 2004.

[63] Maryam Shoaran and Alex Thomo. Fault-tolerant computation of distributed regular path queries. *Theoretical Computer Science*, 410(1):62–77, 2009.

[64] Parag Singla and Matthew Richardson. Yes, there is a correlation:-from social networks to personal behavior on the web. In *WWW*. ACM, 2008.

[65] Christian Sommer. Shortest-path queries in static networks. *ACM Computing Surveys (CSUR)*, 46(4):45, 2014.

[66] Nikita V Spirin, Junfeng He, Mike Develin, Karrie G Karahalios, and Maxime Boucher. People search within an online social network: Large scale analysis of facebook graph search query logs. In *CIKM*. ACM, 2014.

[67] Dan Stefanescu and Alex Thomo. Enhanced regular path queries on semistructured databases. In *International Conference on Extending Database Technology*, pages 700–711. Springer, 2006.

[68] Dan C Stefanescu, Alex Thomo, and Lida Thomo. Distributed evaluation of generalized path queries. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 610–616, 2005.

[69] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, 2011.

[70] Gayatri Swamynathan, Christo Wilson, Bryce Boe, Kevin Almeroth, and Ben Y Zhao. Do social networks improve e-commerce? a study on social marketplaces. In *First Workshop on Online Social Net.* ACM, 2008.

[71] Konstantin Tretyakov, Abel Armas-Cervantes, Luciano García-Bañuelos, Jaak Vilo, and Marlon Dumas. Fast fully dynamic landmark-based esti-

mation of shortest path distances in very large graphs. In *CIKM*, pages 1785–1794. ACM, 2011.

[72] Adam Welc, Raghavan Raman, Zhe Wu, Sungpack Hong, Hassan Chafi, and Jay Banerjee. Graph analysis: do we have to reinvent the wheel? In *First International Workshop on Graph Data Manag. Experiences and Systems*. ACM, 2013.

[73] Jian Wu, Venkatesh Srinivasan, and Alex Thomo. Graph-xll: a graph library for extra large graph analytics on a single machine. In *2019 10th International Conference on Information, Intelligence, Systems and Applications (IISA)*, pages 1–7. IEEE, 2019.

[74] Sihem Amer Yahia, Michael Benedikt, Laks VS Lakshmanan, and Julia Stoyanovich. Efficient network aware search in collaborative tagging sites. *PVLDB*, 1(1), 2008.

[75] Michael Yu, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. Aot: Pushing the efficiency boundary of main-memory triangle listing, 2020.

[76] Tengkai Yu, Venkatesh Srinivasan, and Alex Thomo. Triangle enumeration on massive graphs using aws lambda functions. In *International Conference on Intelligent Networking and Collaborative Systems*, pages 226–237. Springer, 2020.

[77] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[78] Yang Zhang and Srinivasan Parthasarathy. Extracting analyzing and visualizing triangle k-core motifs within networks. In *ICDE*, pages 1049–1060, 2012.

[79] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[80] Beibei Zou, Xuesong Ma, Bettina Kemme, Glen Newton, and Doina Precup. Data mining using relational database management systems. In *Advances in Knowledge Discovery and Data Mining*, pages 657–667. Springer, 2006.

[81] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, pages 59–59. IEEE, 2006.