MSSG: A FRAMEWORK FOR MASSIVE-SCALE SEMANTIC GRAPHS

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Timothy D. R. Hartley, B.S.

* * * * *

The Ohio State University

2006

Master's Examination Committee:

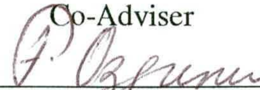
Dr. Umit Catalyurek, Co-Adviser

Dr. Füsun Özgüner, Co-Adviser

Dr. Eylem Ekici

Approved by

Co-Adviser



Co-Adviser

Graduate Program in
Electrical and Computer
Engineering

ABSTRACT

This thesis presents research into techniques for storing, accessing and analyzing massive scale semantic graphs. The end result of this research effort has been the development of a software framework, *MSSG*, to enable the analysis of scale-free semantic graphs with $O(10^{12})$ vertices and edges. Here, we present the overall architectural design of the framework, as well as a prototype implementation for cluster architectures. The sheer size of these massive-scale semantic graphs prohibits storing the entire graph in memory even on medium- to large-scale parallel architectures. We therefore propose a new graph database, *grDB*, for the efficient storage and retrieval of large scale-free semantic graphs on secondary storage. This new database supports the efficient and scalable execution of parallel out-of-core graph algorithms which are essential for analyzing semantic graphs of massive size. We have also developed a parallel out-of-core breadth-first search algorithm for performance study. Experimental evaluations on large real-world semantic graphs show that the MSSG framework scales well, and grDB outperforms widely used open-source out-of-core databases, such as BerkeleyDB and MySQL, in the storage and retrieval of scale-free graphs.

ACKNOWLEDGMENTS

I would like to take this opportunity to thank the people without whom this thesis never would have come to fruition.

I would like to thank Dr. Füsun Özgüner, who was very helpful in getting me settled into graduate school after an absence from school. I would like to thank Dr. Umit Catalyurek, whose patience and guidance have been instrumental in getting to this point. I'd also like to thank Dr. Eylem Ekici for his advice and committee membership.

I would like to thank my colleagues at Lawrence Livermore National Laboratory, Andy Yoo, Scott Kohn, and Keith Henderson.

Lastly I'd like to thank Annie Lindgren for her invaluable support and advice. Without her help, I am certain this thesis would not have seen the light of day!

VITA

January 16, 1980 .... Born - High Wycombe, UK

2002 .... B.S. Electrical and Computer Engineering

2005-present .... Graduate Research Associate, The Ohio State University.

PUBLICATIONS

Research Publications

T. D. R. Hartley, U Catalyurek, F Özgüner, Andy Yoo, Scott Kohn, Keith Henderson, "MSSG: A Framework for Massive Scale Semantic Graphs," in *Proceedings of 2006 IEEE International Conference on Cluster Computing*, 2006.

FIELDS OF STUDY

Major Field: Electrical and Computer Engineering

TABLE OF CONTENTS

Page

Abstract ii

Acknowledgments iii

Vita iv

List of Tables vii

List of Figures viii

List of Algorithms ix

List of Listings xi

Chapters:

1. Introduction 1

2. Previous and Related Work 6

3. Architectural Design 11

    3.1 DataCutter 12
    3.2 Ingestion Service 13
    3.3 Query Service 16
    3.4 GraphDB Service 16
        3.4.1 grDB: Graph Database 18

4. Prototype Implementation 22

4.1 Customization of GraphDB Service 22
4.1.1 Array 22
4.1.2 HashMap 24
4.1.3 MySQL 25
4.1.4 BerkeleyDB 26
4.1.5 StreamDB 26
4.1.6 grDB 27
4.2 Parallel Out-of-core Breadth-First Search 27

5. Experimental Results 32

6. Conclusions and Future Work 42

Bibliography 44

LIST OF TABLES

Table **Page**

5.1 Statistics for graphs used in experiments 32

LIST OF FIGURES

Figure **Page**

1.1 An example semantic graph and ontology 2

3.1 MSSG Overall System Architecture 12

3.2 An example of graph partitioning 15

3.3 An illustration of grDB file format 19

3.4 A sample graph and its storage representation in a 3-level grDB instance with $d_0 = 2$, $d_1 = 4$ and $d_2 = 8$ 21

4.1 An example of the Array GraphDB Service instance 23

4.2 An example of the `HashMap` GraphDB Service instance 25

4.3 An example of the MySQL and BerkeleyDB blocking 26

5.1 Search performance of in-memory GraphDB implementations on PubMed-S graph 33

5.2 Search performance of BerkeleyDB and grDB on PubMed-S graph with and without cache 34

5.3 Ingestion performance comparison of five GraphDB implementations on PubMed-S graph 35

5.4 Search performance comparison of five GraphDB implementations on PubMed-S graph 36

5.5 Ingestion performance comparison of five GraphDB implementations on PubMed-L graph: 8 front-end ingestion nodes, vary back-end storage nodes 37

5.6 Execution time search performance comparison of five GraphDB implementations on PubMed-L graph 38

5.7 Edge/s search performance comparison of five GraphDB implementations on PubMed-L graph 39

5.8 Execution time search performance for Syn-2B graph using grDB 40

5.9 Edge/s search performance for Syn-2B graph using grDB 41

LIST OF ALGORITHMS

Algorithm		Page
1	Parallel Out-of-core Breadth-First Search Algorithm	28
2	Pipelined Parallel Out-of-core Breadth-First Search Algorithm	30

LIST OF LISTINGS

Listing **Page**

3.1 GraphDB Service Interface 17

CHAPTER 1

INTRODUCTION

Graphs have been used to model many interaction networks, ranging from the biological to the computational sciences. Some examples of these interaction networks are metabolic and signaling pathways, gene regulatory networks, protein interaction networks, taxonomies of proteins and chemical compounds, and social networks [42,43,53,59,68,69]. These types of real-world graphs are known as *semantic graphs* [32]. In a semantic graph, vertices represent certain concepts or objects and edges represent the relationships between those concepts or objects. Further, the vertices and edges in the semantic graph are associated with some meaningful types. These vertex and edge types form an ontology graph, which summarizes the semantic information the corresponding semantic graph carries.

Figure 1.1 shows an example semantic graph and its ontology. By itself, an ontology is just an instance of a semantic graph. The vertex and edge interconnections only take on special meaning when the ontology is used as a blueprint for other semantic graphs. When used as a blueprint, the ontology's topology restricts the topology of the instance semantic graph. Taking the Figure 1.1 ontology as an example, note that the 'Date' vertex types are not allowed to be directly connected to the 'Person' vertex type. Any indirect association must occur through the 'Meeting' vertex type and through the allowed edge types, 'attends' and 'occurred on.' The effect of these ontological restrictions is such that

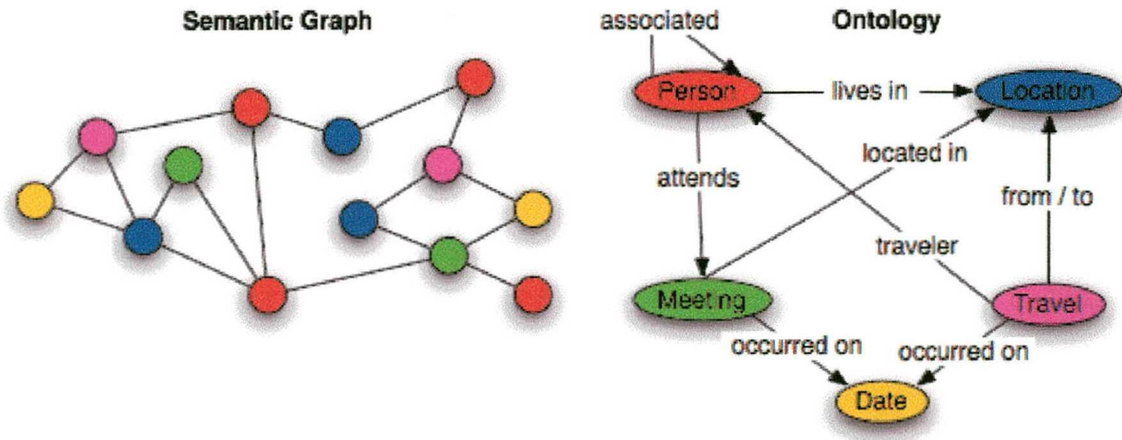


Figure 1.1: An example semantic graph and ontology

any semantic instance graph (as shown on the left of Figure 1.1) must only contain the vertex and edge interconnections as allowed in its associated ontology. That is, 'Date' vertices are only connected to 'Meeting' vertices and 'Travel' vertices.

Since semantic graphs model the relationships between concepts or entities in the real world, their sizes often must reflect the large amount of information required for various models to be accurate. Further, while real-world semantic graphs are typically large, new graphs in some emerging fields are expected to have truly massive numbers of vertices and edges. For example, Kolda et al. [42] predict semantic graphs representing social networks of interest to the Department of Homeland Security will have 10^{15} entities. Additionally, real-world graphs which model interaction networks such as social networks or even the Internet's website interconnections have degree distributions which follow a power-law [10]. This means that if y is the number of nodes with degree x in a graph, then $y \propto x^{-\beta}$. These *power-law* or *scale-free* graphs have several unique properties. They exhibit the *small-world* phenomenon, because their diameter increases only logarithmically with the number

of vertices [66]. As a consequence of this small-world property, queries which analyze long paths often must access a significant portion of the graph data, sometimes over 80% of the total graph's edges in our experiments. Analyzing such large graphs and answering user queries within a reasonable amount of time is an important and very challenging problem.

First, a set of novel graph algorithms are needed in order to analyze these huge data sets. In the best case, a graph with one trillion edges requires 8 billion bytes of disk space to store and over 2,300 seconds at 50 MB per second just to scan through the data spread over 64 clustered compute nodes. Clearly, in order to process given user queries in a timely manner, these graph algorithms must explore large semantic graphs in parallel. More importantly, they must be out-of-core (OOC) algorithms that read and process input graphs that are stored in persistent storage, as the memory requirements of any massive semantic graph are prohibitively large. Although in the literature there are various parallel graph algorithms [9, 20,21,30,40,41,69], and OOC (also called external memory) algorithms [2,3,22,25,46,50], along with one concurrent work which provides a parallel out-of-core breadth-first search algorithm [49], there is no practical framework yet provided to allow future work in this direction.

Second, a new graph storage system and data format are needed. Although traditional relational database systems provide enough performance for business applications such as transaction processing, relational databases are not ideal platforms for storing and processing massive semantic graphs. The same flexibility which makes relational databases appropriate for a wide variety of divergent applications causes them to perform poorly when faced with such strict data and speed requirements as are imposed when dealing with large semantic graphs. Therefore, a new data management system is needed. This data manager

must be able to store streaming semantic graphs of massive size and provide an underlying infrastructure to allow the parallel graph algorithms to access and process the stored graph in a scalable and cost-effective way.

In this thesis, we present a middleware framework for storing, accessing and analyzing massive-scale semantic graphs. The framework, *Massive-Scale Semantic Graphs (MSSG)*, targets scale-free semantic graphs with $O(10^{12})$ vertices and edges. In a scale-free graph, most of the vertices are only connected to a small number of other vertices (i.e., they have low degree), while a few vertices, known as *hubs*, are connected to a large number of other vertices. Most of the real-world semantic graphs exhibit this topological property.

The sheer size of these massive-scale semantic graphs prohibits storing the entire graph in memory even on medium- to large-scale parallel architectures; hence, these graphs will need to be persistently stored in a distributed database. The framework is architected targeting large clusters with compute nodes that have direct access to fast disk storage. One of the many possible configurations is compute nodes with large local disks. Another configuration is compute nodes that are connected to fast storage arrays via Storage Area Network switches. An example of the latter is Ohio Supercomputing Center's Mass Storage system [18].

We propose a new graph database, *grDB*, for storing and processing large scale-free semantic graphs on secondary storage. The grDB database stores the vertices and edges of a semantic graph in such a way that the number of disk I/Os required to access adjacent vertices is minimized while still efficiently utilizing the storage space.

We have developed a prototype implementation of MSSG. This prototype has been built on top of DataCutter [15, 16], which uses MPI as its low-level communication protocol. Experimental evaluations with both real-world scale-free graphs and synthetic scale-free

graphs show that our prototype scales well. We have also compared grDB against other widely used open-source databases, BerkeleyDB [61] and MySQL [47], as well as in-memory graph storage implementations (for small graphs). These experiments showed the effectiveness of the proposed graph database for scale-free graphs.

The main contributions of this research are summarized as follows.

- We have architected a framework, MSSG, to store, retrieve, and process massive-scale semantic graphs.
- We have developed a novel data management system called grDB. Unlike traditional relational databases, grDB is optimized specifically for efficiently storing and accessing semantic graphs of massive size.
- We have developed a parallel OOC breadth-first search algorithm that runs on distributed parallel machines.
- We have evaluated the performance of MSSG using large real-world semantic graphs and randomly generated large scale-free graphs . The results show that the MSSG framework scales well, and that the proposed graph database, grDB, outperforms other open-source database systems, Berkeley DB and MySQL.

The remainder of this thesis is organized as follows. Some related work is presented in chapter 2. The architecture of the proposed MSSG framework and its implementation are described in detail in chapters 3 and 4. Experimental results are presented in chapter 5, followed by concluding remarks in chapter 6.

CHAPTER 2

PREVIOUS AND RELATED WORK

Since the focus of this work is to create a framework for designing parallel, out-of-core algorithms to analyze massive-scale semantic graphs, there are several fields which can be looked to for inspiration and competition. Clearly, the field of graph models will have a large part to play, as well as the newer specialization within that field dealing with scale-free graphs. Since no analysis can occur in a computer system without an algorithm, the fields of external-memory algorithms and graph algorithms will both come into the equation. Also, the field of parallel algorithms must be given a voice, since the volume of data to be processed will necessitate a parallel solution to our problem, as stated in the introduction. Lastly, the field of runtime middleware will likely have some part to play, since our system intends to be an easy-to-use framework on top of which parallel, out-of-core graph algorithms can be designed.

The early graph models [19] relied on the assumption that any sufficiently large graph would exhibit two properties which would completely describe the graph, number of vertices and average degree of connectedness of the vertices. This model was the most widely used for randomly constructed graphs for the first several decades of graph modeling. With the advent of the Internet and the increased ability to automatically traverse and store graphs, [10] found that the ER model for random graphs failed to explain the topology

present in the Internet website graph. It was also found that other types of self-organizing, real-world graphs have a degree distribution which is not predicted by the ER model. Therefore, a new theory for random graphs has become popular. The scale-free theory of random graphs states that in these graphs, the degree distribution follows a power-law. That is, the probability of a node to have a certain degree k is proportional to k raised to a some negative constant. The term 'scale-free' denotes that these power-law graphs will retain the power-law degree distribution no matter how you scale the time over which the vertices are added and connected, provided that the method used to add vertices and edges is not altered.

The field of out-of-core algorithms extends a computer's ability to process data sets to those which do not fit into main memory. Since many algorithms use a simple record sort or permutation as part of their computational kernel, it stands to reason that the field of external-memory algorithms began with techniques for sorting and permuting records which do not fit into the main memory of a single machine [33, 39, 51]. That is, some form of persistent secondary memory is the main location where the records are resident, and only some small proportion of the records are brought into the main memory at any one time. Algorithms in fields such as computational geometry [27], computer graphics [17], linear algebra [62], and graph theory [26] make use of out-of-core techniques when the input data sets cross certain thresholds. Out-of-core graph algorithms, which are directly pertinent to our research, exist for most of the in-memory algorithms for graph problems: directed and undirected search, connected components, minimum spanning trees, etc.

Due to the large size of the data this research targets, the field of parallel algorithms must also have a part to play in our solution. Parallel algorithms have been developed for an enormous array of different problems, ranging from simple sorting and permuting [29, 58]

to graph algorithms of all kinds [57], from linear algebra [35] and matrix multiplication to finite element methods (FEM) for solving systems of partial differential equations [31], from parallel molecular dynamics simulations [28,36,56] to algorithms for visualizing and analyzing data sets too large to fit into a single computer [17]. It is a mature field with a large number of publications, and it has much to offer this research in the way of inspiration and background.

Since the goal of this research is to design and implement a runtime system to enable graph algorithms to process extremely large data sets, the runtime engine or middleware fields [1,5,6,23,38,52,55] may also have some valuable insight. Additionally, any research into the handling of massive data is also worth reviewing. A common thread in practical methods for processing massive data is to consider the data to be streaming through some computation. This idea is found in many of the data-intensive runtime systems presented in the literature, and should therefore be considered carefully.

TPIE [7] is a runtime engine where the algorithm developer does not specify any of the details about the movement of data to and from the secondary memory. Rather, the developer need only specify the operations to perform the computation, and the runtime system itself will manage the movement of data in an I/O-efficient manner. DataCutter [16] is a component-based middleware framework designed to provide support for user-defined processing of large multi-dimensional datasets across a wide-area network. In DataCutter, application processing structure is implemented as a set of components, referred to as filters, that exchange data through a stream abstraction. River [8] takes the stream concept a stage further and allows its processing filters to take work from a distributed queue, thereby adaptively allocating work where it is needed most. River's main goal is to disallow non-uniformities in the processing nodes from affecting the computation adversely. MQO [12]

is a grid middleware platform which uses a distributed index cache to answer multiple-range queries efficiently. While not directly applicable to the storage and analysis of graph data, the techniques put to work in MQO to deal with the massive amount of data, and the distributed nature of the application may also be appropriate in this research. The Parallel Boost Graph Library [54] is meant to provide a generic framework and template functions for graph analysis on parallel machines.

Indexing is a well-known technique for reducing the latency associated with retrieving data elements stored on disk. The B-tree (and its variants) is a well-known, efficient data structure used for indexing single-dimensional data. Various index structures have been proposed for retrieval of multidimensional data. Examples include VQ-Index [63], kdb trees [60], hB tree [45], R* tree [11], R tree [34], SS tree [67], TV tree [44], X tree [14], Pyramid Technique [13], Hybrid Tree [24]. While graph data is different from spatial data, it is still likely that some of the techniques employed in these indexing schemes are applicable.

While scalable and communication-efficient parallel graph algorithms and I/O-efficient external memory algorithms exist separately, and some concurrent research has investigated merging parallel and out-of-core techniques in a breadth-first search algorithm [49], there is no framework or middleware platform which allows for the practical development of algorithms which are meant to run on parallel architectures and utilize external memory. MSSG intends to be that framework, which will enable these types of algorithms to be designed and implemented.

There are a number of commercial solutions which are worth considering for this type of graph analysis, one of which is marketed by Netezza [48]. We believe that our solution is preferable to that of the Netezza search appliance because MSSG allows an unrestricted

set of operations to be performed on the graph data, while the Netezza solution's set of operations are restricted naturally by the computational resources (in this case, an FPGA) which are close to the data stored on disk. By using MSSG, the algorithm developer is only constrained by the aggregate size of the memory space and the speed of the computational resources available, both of which can be improved when required. Additionally, our solution does not required that the computational resources are devoted to the database application. A distributed memory parallel machine can be used for a myriad of other scientific and engineering applications, beyond that of data storage and processing. Moreover, our solution allows parties interested in graph search to leverage their current resources without needing to buy extra expensive, proprietary hardware.

CHAPTER 3

ARCHITECTURAL DESIGN

The MSSG framework is designed to provide storage, retrieval and processing of large scale-free graphs. It consists of one or more front-end nodes which provide an entry point for the user queries as well as graph data ingestion, and a set of back-end nodes that are responsible for storing and processing the graph data (Figure 3.1). MSSG has been built on top of DataCutter [15,16] and its functionality is provided by a set of modular, customizable services implemented as DataCutter components and pluggable interfaces. The *Ingestion Service* provides an entry point for data storage and it is responsible for clustering and declustering of the graph data to the back-end storage nodes. The *Query Service* allows for analysis of the stored graph, while the *GraphDB Service* provides a unified mechanism for storing and accessing graph data.

Both the Ingestion and GraphDB services can draw parallels from the parallel file-system domain, particularly the Parallel Disk Model [64, 65] (PDM). The PDM provides a generic model for use in designing OOC algorithms and in calculating upper and lower bounds for OOC algorithm performance. In a sense, the Ingestion service's declustering of the input graph is equivalent to striping the graph data intelligently across multiple disks in a uniprocessor system. Within the Ingestion and GraphDB implementations, there are also PDM optimization techniques which may be applicable.

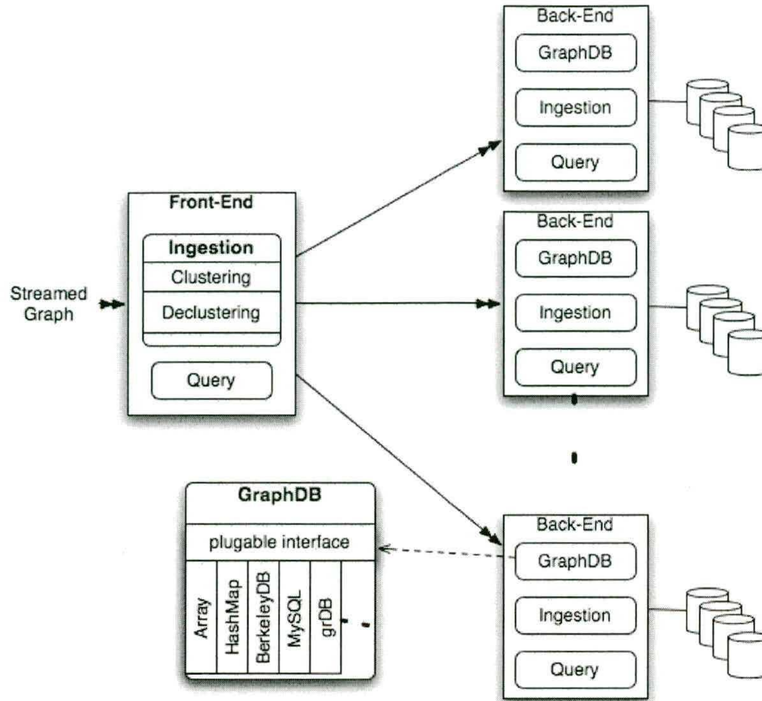


Figure 3.1: MSSG Overall System Architecture

In the following sections, we first present a brief overview of DataCutter and then discuss the details of each service.

3.1 DataCutter

DataCutter [15, 16] is a component-based middleware framework [1, 5, 6, 23, 38, 52, 55] designed to support coarse-grain dataflow [8] execution on heterogeneous environments. In DataCutter, the application processing structure is implemented as a set of components, referred to as *filters*, that exchange data through *logical streams*. A *stream* denotes a unidirectional data flow from one filter (i.e., the producer) to another (i.e., the consumer). A filter is required to read data from its input streams and write data to its output streams only.

The DataCutter runtime system supports both data- and task-parallelism. Processing, network, and data copying overheads are minimized by the ability to place filters on different platforms. The filtering service of DataCutter performs all steps necessary to instantiate filters on the desired hosts, to connect all logical endpoints, and to call the filter's interface functions for processing work. Data exchange between two filters on the same host is carried out by memory copy operations, while a message passing communication layer (e.g. TCP sockets or MPI) is used for communication between filters on different hosts.

3.2 Ingestion Service

The Ingestion service provides the entry point for graph data to the MSSG system. Its job is to cluster and decluster (distribute) the ingested data appropriately to the GraphDB instances on the back-end nodes. Due to the sheer size of target graphs, these operations should be very efficient. The ideal approach is to perform these operations while the graph data is being ingested by the system via streaming.

The goal of this clustering and declustering is to achieve fast query processing by reducing the total number of disk I/Os incurred to access the database and increasing the parallelism during the query processing. Parallelism is relatively easier to achieve for queries which require processing a large portion of the dataset compared to queries that require processing only a localized portion of the data. For example, if a query were in the form of search between two vertices, it would be ideal if the vertices of the graph that are close to source of the search were clustered together in one GraphDB instance to reduce the I/O overhead. However, we also would like those vertices to be spread out to the nodes of the distributed storage system in order to achieve better parallelism.

A graph can be clustered and stored mainly at two granularity levels: 1) at the vertex level by storing all the edges incident to a vertex together and 2) at the edge level by storing each edge as an independent entity. MSSG supports both granularity levels. Similarly, streaming updates can arrive at those two granularity levels. Adding new vertices and new edges using vertex- and edge-level granularity respectively, however, necessitates novel clustering techniques. If vertex granularity is selected at the storage side, it would largely dictate the clustering and declustering of streaming updates. That is, if a vertex has been already clustered and assigned to storage node, all the new edges incident to that vertex have to be added to the same cluster to which the vertex belongs. Clustering would be simpler in this paradigm, but updating the data each time a new edge is stored can be very costly. Smart caching and blocking techniques help reduce the number of disk I/Os due to updates.

For clustering streaming data, MSSG processes the ingested data in *blocks* (or *windows*) of a predetermined size, each of which fits into memory. Any streaming data can be converted into this format by accumulating incoming data to construct a block. Clustering algorithms will work on those blocks one by one. These algorithms must be very efficient in order for decisions to be made in real-time. Furthermore, these algorithms should keep some additional summary information about the data that has been already clustered and distributed to the nodes of the storage system. Using the summary information, the clustering algorithms should be able to make more intelligent decisions on where to send blocked data.

Figure 3.2 shows a simple example of a graph being partitioned and distributed to three back-end database nodes. In this example, the block which is to be partitioned contains nine vertices and the edges which interconnect them as shown. The partitioning algorithm

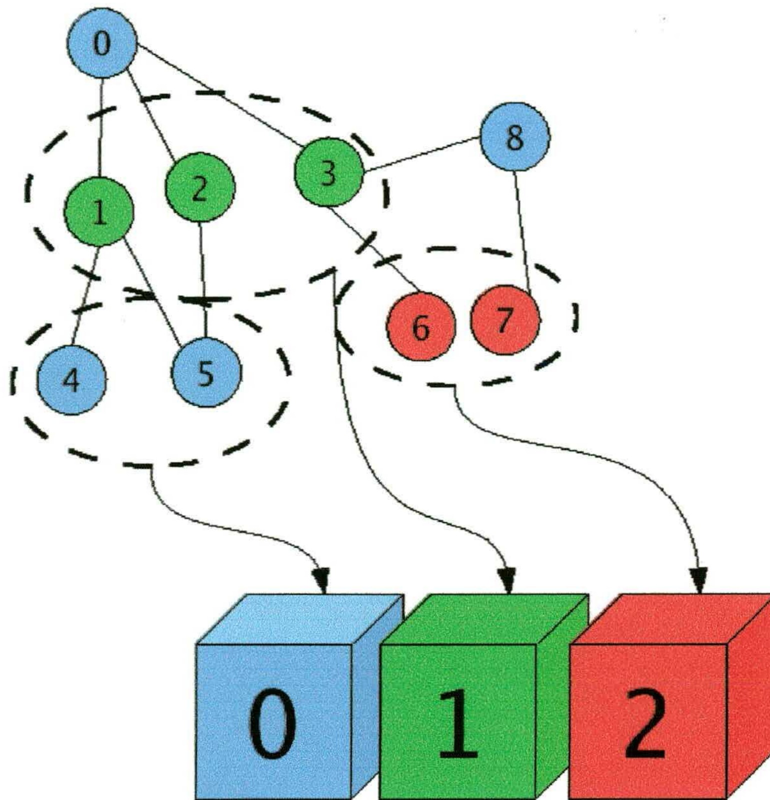


Figure 3.2: An example of graph partitioning

has chosen vertices 4, 5 to go to database node 0, vertices 1, 2, and 3 to go to database node 1, and vertices 6 and 7 to go to database node 2. If an edge in a block needs to be partitioned when the source vertex has already been assigned to a back-end node (in the case of vertex granularity partitioning), the Ingestion service needs to keep track of the owner of that vertex's edges. Various methods of accomplishing this goal exist, spanning from using a deterministic function to map the vertex id's to back-end node id's to simply broadcasting all edges to all back-end nodes and required the back-end node to store the edge, if appropriate.

MSSG provides a customizable interface for developing clustering and declustering techniques. By default, the MSSG framework provides simple declustering techniques such as vertex- and edge-based round-robin declustering.

3.3 Query Service

The Query service provides the query interface for the client and orchestrates the execution of data analysis queries. In the MSSG framework, data analysis techniques are implemented as DataCutter filter graphs communicating via DataCutter's filter-stream interface. All implemented data analysis techniques are registered with the system and can be queried by the user. Any data analysis accesses the stored graph data via the unified graph interface provided by GraphDB Service. Since each GraphDB instance only provides direct access to local data stored in each node, data analysis service instances need to be implemented in such a way as to take this data distribution into account. For example, if the graph is stored using vertex-level granularity, the complete adjacency list of any arbitrary vertex will be stored in only one node. Hence, any operation that requires accessing the adjacency list of a vertex needs to be either delegated to that node or the adjacency list needs to be transfered to the node that initiated the access. As an example Query Service instance, a relationship analysis method based on breadth-first search is described in more detailed in Section 4.2.

3.4 GraphDB Service

The GraphDB Service's job is to interface with a number of disparate storage mediums, such as various in-memory data structures, relational databases, and other disk-based storage methods.

```java
public interface Graph {

  public void storeEdges(List<Edge> edges) throws
      GraphStorageException;

  public int getMetadata(long vertex) throws GraphStorageException;

  public void setMetadata(long vertex, int metadata)throws
      GraphStorageException;

  // operation is defined as follows:
  //   -2 : ignore metadata and return all neighbor vertices
  // or return an neighbor to vertex if its metadata is:
  //   -1 : not equal to input metadata
  //    0 : equal to input metadata
  //    1 : greater than input metadata
  //    2 : less than input metadata
  public void getAdjacencyListUsingMetadata(long vertex,
      FastLongArrayStorage adjlist, int metadata, int operation);
}
```

Listing 3.1: GraphDB Service Interface

One of the main innovations of the MSSG API consists of a Java interface *Graph* (Listing 3.1) which exposes the smallest complete set of graph operations possible, along with one higher-performance method which implements a slightly higher-level graph function, for performance reasons. In order to be complete, a graph-storage service only needs to store edges and retrieve lists of distance-1 neighbors (adjacent vertices). It is important to note that none of the methods in the GraphDB Service interface perform any communication or remote operations. All of the methods listed operate on data local to the back-end node.

Currently, there are several concrete classes which implement the graph interface and store the actual graph data in different formats and different storage mediums. Two of the default implementations are based on efficient in-memory storage for graphs that could fit in memory of the MSSG installation. We also provide four disk-based implementations of GraphDB services. Two of them are based on open-source databases, BerkeleyDB and

MySQL. One of them is based on the notion, borrowed from research into Active Disks [4], that database searches which touch a large portion of the data will benefit by simply streaming the entire data set into the CPU and processing it in chunks. The last one, *grDB*, is a novel disk-based graph database designed for massive scale-free graphs.

3.4.1 grDB: Graph Database

We propose a novel graph database, *grDB*, which is intended to allow the efficient out-of-core storage and retrieval of scale-free graphs. A grDB instance is comprised of two components; the *storage component* and the *block cache component*. The storage component is responsible for the storage and retrieval of *blocks* which store partial adjacency lists of one or more vertices. The block cache component provides in-memory caching of the storage blocks for improved performance.

A scale-free graph in grDB is stored in multiple files that are composed of *blocks*. Blocks are smallest unit of I/O for grDB. While the optimum block size is determined by the performance characteristics of the physical storage system, we nevertheless expect the optimum block size will not be smaller than the filesystem's block size. Each block will be further divided into *sub-blocks* that are uniquely addressable. A sub-block is used to store a vertex's partial adjacency list. A grDB instance contains multiple levels of storage files. At level ℓ, each sub-block of a storage file can store up to d_ℓ adjacent vertices, where $d_\ell \geq 2 \times d_{\ell-1}$ for $\ell \geq 1$. Since our target graphs exhibit the power-law degree distribution, we suggest choosing d_ℓ values that also follow an exponential curve, such as $d_\ell = 2^{2^\ell}$.

Each vertex in grDB will have a b-byte unique Global ID (GID) in the range between 0 to n, where n is the number of vertices in the graph; hence, each sub-block in level ℓ is $b \times d_\ell$ bytes. Since each block can store one or more sub-blocks, block size B_ℓ at level ℓ

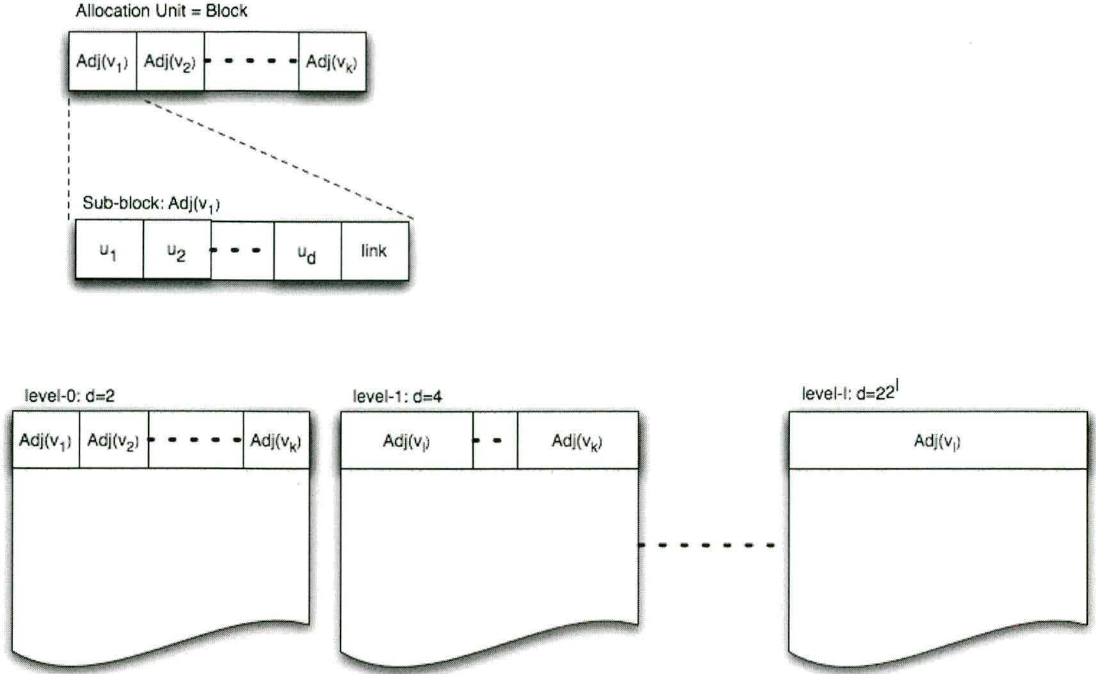


Figure 3.3: An illustration of grDB file format

is computed as $B_\ell = k_\ell \times b \times d_\ell$, for an integer $k_\ell \geq 1$. Because of file system limitations as well as performance reasons, at each level ℓ graph data is stored in multiple files with a maximum size of M-bytes, or equivalently $N_\ell = M/B_\ell$ blocks. The location of a sub-block s in the disk at a level ℓ can be found using simple modulo arithmetic as follows. Since each block stores k_ℓ sub-blocks, sub-block s is stored in s/k_ℓ-th block, which is stored in $s/k_\ell/N_\ell$-th file at offset $B_\ell \times ((s/k_\ell) \,\%\, N_\ell) + b \times d_\ell \times (s \,\%\, k_\ell)$.

The beginning of the adjacency list of a vertex v in grDB is stored in v-th sub-block at level 0. If a vertex has d_0 or less number of adjacent vertices, they are directly stored in that level. If v has more adjacent vertices than d_0, the first $d_0 - 1$ adjacent vertices are stored in the level 0 sub-block, and the last location in the level 0 sub-block is used as a pointer into

the higher level files. During the ingestion of the graph data set, if the adjacent vertices are added in small groups, the adjacency list of a vertex could have entries in multiple levels of the grDB. For example, if vertex v already has d_0 adjacent vertices and one more adjacent vertex is added, a new sub-block is allocated for that vertex in level 1. A link from the v-th sub-block at level 0 to the newly allocated sub-block at level 1 is created. When the degree of vertex v achieves $d_0 + d_1$, a new sub-block is allocated for that vertex at level 2, and either all of the contents of the sub-block at level 1 are moved to the new sub-block at level 2 and subsequent new adjacent vertices are added to that sub-block, or the sub-block at level 1 is left unchanged and simply links to the newly allocated sub-block at level 2. The former approach necessitates extra copy operations during the insertion, while the latter creates fragmentation in the adjacency list. One approach is to leave the adjacency lists fragmented during the ingestion, and later during "idle" time, the grDB service can defragment these multi-level adjacency lists in the background. Figure 3.3 illustrates the file format of grDB, and a small example is shown in Figure 3.4.

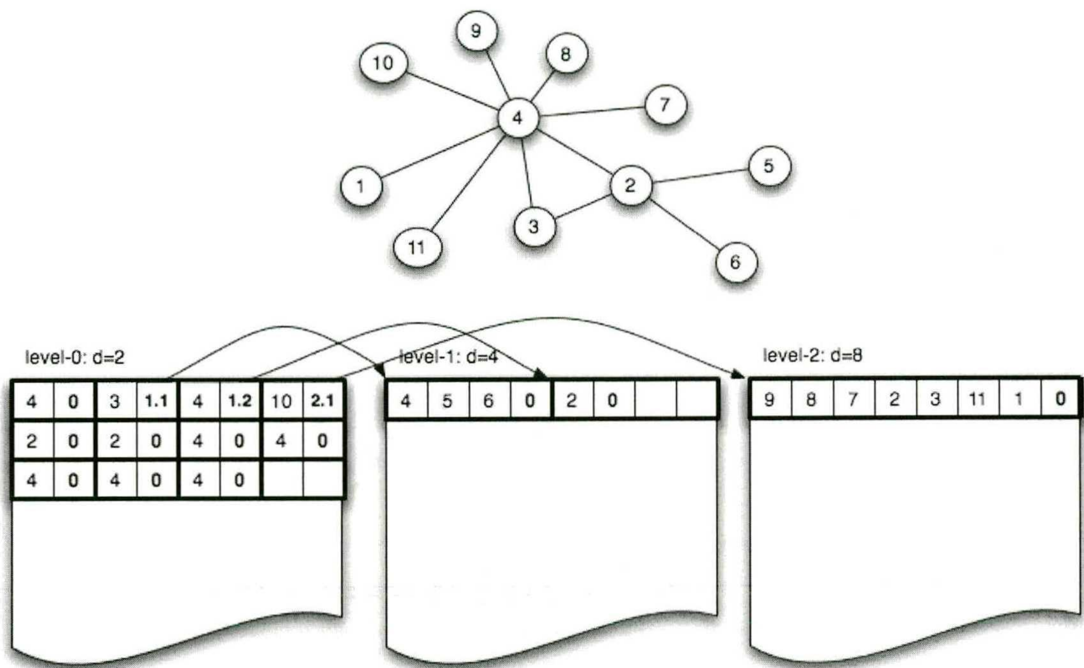


Figure 3.4: A sample graph and its storage representation in a 3-level grDB instance with $d_0 = 2$, $d_1 = 4$ and $d_2 = 8$

CHAPTER 4

PROTOTYPE IMPLEMENTATION

We have implemented a prototype of the MSSG framework in Java. The framework allows analysis services to be implemented in Java using DataCutter's filter interface. We have implemented an analysis service which uses breadth-first search. Below we present the details of the prototype MSSG middleware and the breadth-first search analysis plug-in.

4.1 Customization of GraphDB Service

We have implemented six different instances which meet the generic GraphDB interface contract, two in-memory versions and four out-of-core versions using various persistent storage managers. Although the graphs we are targeting will not fit in memory, the two in-memory implementations provide a solid base-line comparison for our out-of-core implementations. In a sense, they represent the lower-bound we could achieve with the out-of-core implementations. Here is a brief description of these six GraphDB instances:

4.1.1 Array

The first in-memory implementation uses the standard *compressed adjacency list* format to store the graph in memory. When using the compressed adjacency list format, a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is stored using two arrays of size $|\mathcal{E}|$ and $|\mathcal{V}| + 1$, respectively. The first

array, adj, contains the adjacency list of all vertices concatenated one after the other, while the second array, $xadj$, stores a pointer to the beginning of adjacency list for each vertex. That is, the adjacency list of vertex v is stored at $adj[xadj[v]], \ldots, adj[xadj[v+1]-1]$. The advantage of this format is it provides very efficient access to the adjacency list of each vertex, by using the highest-performance in-memory data structure possible. However, this format has three major issues. First, Java only allows 32-bit integers as array indices, which restricts the input graph size. Second, this storage format is not suitable for dynamically-growing graphs. Third, it is poorly suited for storing graphs distributed to multiple machines; unless a block distribution of vertices is used, each node has to store the full $xadj$ array. Therefore, the array-based storage format's memory requirement does not scale with increasing numbers of back-end nodes. For small graphs that fit into main memory, the first and third concerns will not cause problems. For the input stage, when the graph is streaming in from the front-end nodes, we have actually used the `HashMap` implementation (see 4.1.2) with integer IDs as temporary storage. After flushing the graph to disk, the `Array` GraphDB instance loads the graph into the compressed adjacency list arrays. The `Array` implementation is useful as a lower-bound on the search execution times.

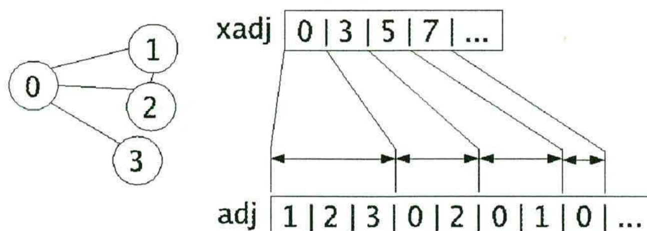


Figure 4.1: An example of the Array GraphDB Service instance

The Figure 4.1 shows a small example of how a graph might be mapped into the `Array` GraphDB instance. For instance, the vertex 0 has adjacent vertices 1, 2, and 3. These neighbor vertices to vertex 0 are located in adj from index $xadj[0] = 0$ to index $xadj[1] - 1 = 2$. The adjacency list of vertex 1, consisting of vertices 0 and 2, is stored immediately after that of vertex 0, beginning at $xadj[1] = 3$ and continuing to $adj[2] - 1 = 4$. The other adjacency lists are stored adj in a similar manner, and their beginning and ending indices are stored in $xadj$ similarly also.

4.1.2 HashMap

By using a *hash* data-structure one can improve on the memory requirements of the `Array` implementation when dynamic or distributed graph storage is required. There are two possible implementations, the first of which is to use a hash data-structure to map global vertex IDs to local vertex IDs. The compressed adjacency list array implementation can then be used with the local, renumbered vertices. This implementation requires *global-to-local* and *local-to-global* vertex ID translations and (like the `Array` instance) is not very suitable for dynamically growing its storage during the ingestion. The other implementation option entails storing the adjacency lists of each vertex separately and using a hash data-structure to store and retrieve the pointers to those adjacency lists. Although only global vertex IDs (64-bit longs in Java) are used and there is no need for global-to-local and local-to-global ID translation, accessing the adjacency list of a vertex still requires a hash look-up. We have implemented the latter approach using Java's `HashMap` data structure, which gives this GraphDB instance its name. As already mentioned, this implementation's memory requirement scales well when increasing the number of back-end nodes, at the expense of additional hash look-up time in order to access the adjacency list for a vertex.

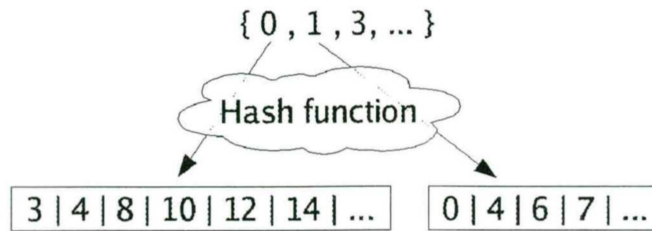


Figure 4.2: An example of the `HashMap` GraphDB Service instance

The Figure 4.2 shows a simple example of how a graph might be stored in a `HashMap` GraphDB instance. In this example, the vertex 0 has at least 6 adjacent vertices stored in an array. The key in the `HashMap` in our implementation is simply the vertex id 0 and the value is a pointer to the adjacent vertex storage object. The actual location of 0's adjacency list storage is unimportant, since the `HashMap` will always hold the pointer to that object. This is a major advantage in storing graphs which will have edges and vertices added to them in a dynamic fashion, as opposed to cases where the entire graph will be known before it is stored in the GraphDB Service instance.

4.1.3 MySQL

We have implemented an out-of-core graph database instance using MySQL 4.1.12 [47]. With a standard {*src, dest*} table model, the overhead of retrieving the adjacency list of a vertex will be prohibitively high for a table with conceivably hundreds of millions of rows. Therefore, we have chosen to store the adjacency list of a vertex in one or more MySQL records indexed by the vertex ID. In order to provide a level of performance that approaches the other implementations we also have chosen to serialize the adjacency list into a *BLOB* data type in a table which is indexed by the source vertex. Since BLOBs can be of arbitrary

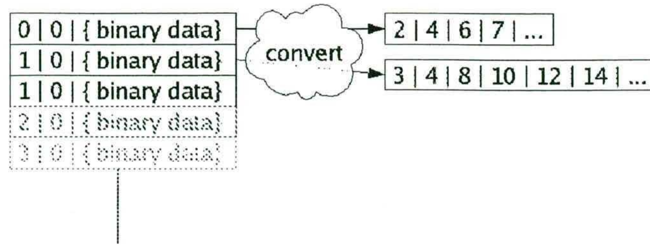


Figure 4.3: An example of the MySQL and BerkeleyDB blocking

size, however, we chose to chunk the adjacency list into standard-sized blocks (8 KB), as suggested by the MySQL documentation and demonstrated in Figure 4.3. The figure shows a database table with three columns. The first column is the vertex id. The third column is the adjacency list storage area, in binary format. The second column is simply a bookkeeping column to keep track of adjacency lists of too large a size to fit into one binary data object. That is, if the adjacency list of a vertex is too large to fit into one row, it is split over multiple rows and the second column in the table is used as a unique identifier for each row.

4.1.4 BerkeleyDB

We have also implemented an out-of-core graph database instance using BerkeleyDB version 1.7.1 [61]. The BerkeleyDB is a programming API which gives the user easy access to persistent, transactional, and storage without the overhead of using a relational database server. The chunking technique used in the MySQL implementation is also used here.

4.1.5 StreamDB

We have implemented a basic streaming database which stores the edges to disk as they are received by the backend node. No sorting or clustering of the edges is performed, which makes this GraphDB instance simple to implement and able to exhibit high performance

during ingestion of graph data. However, this simplicity and ingestion speed is obtained by accepting lower search speed and higher search algorithm complexity. This design of the database prevents an instance from returning a vertex's adjacency list without scanning through the entire edge set. The expansion of a fringe in a breadth-first search, for example, would be very expensive if each vertex were considered one at a time. Therefore, any search algorithm which needs the adjacent vertices to another set of vertices to proceed must post a request for all of the 'fringe' vertices at once, thereby allowing the database to only scan through its data once. This database design was inspired by the active disk work in [4].

4.1.6 grDB

We have implemented the proposed grDB discussed in Section 3.4.1 in Java using the standard `RandomAccessFile` class. Global vertex IDs and file sub-block address pointers are 64-bit long integers where the 3 most significant bits are reserved for the grDB's internal use to mark when the value is a pointer to a higher-degree storage file. With 3 bits acting as the pointer indicator, we still allow 61 bit vertex numbers which will be sufficient for graphs with up to 2 quintillion vertices ($2 * 10^{18}$). In our experiments we have restricted the maximum file size to be $M = 256\ MB$. We have used a 6-level instance of grDB with d_ℓ values are equal to 2, 4, 16, 256, 4K and 16K, and with a block size, B_ℓ, of $4\ KB$ in the first 4 levels and 32 KB and 256 KB for the last two levels.

4.2 Parallel Out-of-core Breadth-First Search

In the prototype we provide an example instance of the Query Service which implements a parallel out-of-core Breadth-First Search (oocBFS) algorithm. BFS is one of the basic algorithms for relationship analysis [42, 69].

Algorithm 1 Parallel Out-of-core Breadth-First Search Algorithm

```
1: function OOCBFS(𝒢 = (𝒱, ℰ), s, d)
2:   Initial data distribution: 𝒢 is divided into p subgraphs G_1 = (V_1, E_1), ..., G_p =
        (V_p, E_p) where E_1, ..., E_p is a disjoint partition of the edge set ℰ and V_i ⊆ 𝒱
        for 1 ≤ i ≤ p.
3:   on each processor P_i, 1 ≤ i ≤ p.
4:     level[v] = ∞ for v ∈ 𝒱
5:     F ← adj_{G_i}(s)
6:     level[s] ← levcnt ← 0
7:     while 'found' message has not been received do
8:       levcnt ← levcnt + 1
9:       for v ∈ F do
10:        for u ∈ adj_{G_i}(v) do
11:          if u = d then
12:            send found message to all processors and return levcnt
13:          else if level[u] = ∞ then
14:            level[u] ← levcnt
15:            N ← N ∪ {u}
16:      if 𝒢 is stored with vertex-level granularity and vertex mapping map[u] is
           known by every processor then
17:        for u ∈ N do
18:          S_{map[u]} ← S_{map[u]} ∪ {u}
19:        send S_q to processor P_q for 1 ≤ q ≤ p.
20:      else
21:        broadcast N to all processors
22:      Receive R_q from P_q for 1 ≤ q ≤ p.
23:      F ← ∪_q R_q
24:  return infty
```

Algorithm 1 outlines the parallel oocBFS algorithm, which is not much different than a sequential BFS algorithm. The main differences come from graph data distribution and storage. If an edge-level granularity is used to store the graph, it is possible that the adjacency list of a vertex is distributed to multiple nodes. In such a case, the successive search level's frontier vertices N need to be broadcasted to all the processors. A similar situation arises when vertex-level granularity is used but the vertex mapping is not known globally by every processor. In this case, frontier vertices have to be broadcasted to all processors

also. In both of these cases, the frontier set F will be identical for all processors. However, if vertex-level granularity for distribution is used with a globally-known mapping (such as $GID \% p$, where p is the number of back-end nodes), on each processor P_i the frontier set F will only contain the vertices assigned to P_i. Algorithm 1 handles all of these cases naturally by using a simple graph interface which returns the empty set when an adjacency list of a vertex that is not assigned to that processor is requested (steps 5 and 10).

In the current implementation, we rely heavily on the underlying database's caching and clustering capabilities to hide the latency that comes from accessing the adjacency list on disk. This dramatically simplifies the oocBFS implementation and our experiments show that both BerkeleyDB and grDB benefit from using a cache.

Algorithm 2 outlines the pipelined parallel oocBFS algorithm. This algorithm differs from algorithm 1 in that the communication is overlapped with computation as much as possible. This overlap is achieved by splitting the next level fringe into chunks and sending these chunks to the other processors in a pipelined fashion in lines 16 to 22. Each processor only sends a fringe chunk to another processor when the chunk which contains the vertices owned by that processor is past a certain size, *threshold*. In the case where vertex ownership is not globally known, all of the fringe chunks must be sent to all other processors. After sending (line 17) this fringe chunk to the appropriate processor, the local next-level fringe container is cleared, so as to not send redundant data (line 19). However, if the vertex-level ownership is not known, then the next level fringe must be broadcast (also line 17) to all of the other processors. It is then transferred to local storage before the container is cleared. Since sending a small message from one DataCutter filter to another filter is a non-blocking operation, this allows the algorithm to return to processing local fringe vertices while the communications subsystem actually sends the message.

Algorithm 2 Pipelined Parallel Out-of-core Breadth-First Search Algorithm

1: **function** POOCBFS($\mathcal{G} = (\mathcal{V}, \mathcal{E}), s, d$)
2: *Initial data distribution*: $\mathcal{G}$ is divided into p subgraphs $G_1 = (V_1, E_1), \ldots, G_p = (V_p, E_p)$ where $E_1, \ldots, E_p$ is a disjoint partition of the edge set $\mathcal{E}$ and $V_i \subseteq \mathcal{V}$ for $1 \leq i \leq p$.
3: *Vertex mapping* map *is initialized such that* $map[u]$ *gives the processor which owns vertex* u*. If the global vertex distribution is not known,* $map[u] = 0$ $\forall$ $u \in \mathcal{V}$*,* N_0 *will be the broadcast queue, and* N_1 *will be the local next level queue.*
4: **on each** processor P_i, $1 \leq i \leq p$.
5: $level[v] = \infty$ for $v \in \mathcal{V}$
6: $F \leftarrow adj_{G_i}(s)$
7: $level[s] \leftarrow levcnt \leftarrow 0$
8: **while 'found'** message has not been received **do**
9: **for** $v \in F$ **do**
10: **for** $u \in adj_{G_i}(v)$ **do**
11: **if** $u = d$ **then**
12: send **found** message to all processors and **return** $levcnt$
13: **else if** $level[u] = \infty$ **then**
14: $level[u] \leftarrow levcnt$
15: $N_{map[u]} \leftarrow N_{map[u]} \cup \{u\}$
16: **if** $N_{map[u]} \geq threshold$ **then**
17: send $N_{map[u]}$ to processor $map[u]$ or **broadcast**
18: **if** $map[u] \neq 0$ **then**
19: $N_{map[u]} \leftarrow \emptyset$
20: **else**
21: $N_1 \leftarrow N_1 \bigcup N_0$
22: $N_0 \leftarrow \emptyset$
23: $F \leftarrow \emptyset$
24: **if** message R is waiting **then**
25: **for** $v \in R$ **do**
26: **if** $level[v] = \infty$ **then**
27: $F \leftarrow F \bigcup \{v\}$
28: **else**
29: $levcnt \leftarrow levcnt + 1$
30: send N_i to each other processor P_i
31: **if** $map[0] = 0$ **then**
32: $F \leftarrow N_1 \bigcup N_0$
33: **else**
34: $F \leftarrow N_{myid}$
35: **for** $1 \leq i \leq p$ **do**
36: $N_i \leftarrow \emptyset$
37: **return** $infty$

The performance of these algorithms can be further optimized by introducing some pre-fetching of the adjacency lists of the vertices in the frontier. Further optimization for performance might include sorting the pre-fetch disk accesses by file offsets to reduce the seek overhead. As part of our future work, we will investigate both of these options.

CHAPTER 5

EXPERIMENTAL RESULTS

We carried out the experimental evaluation of the MSSG framework on a 64-node Linux cluster owned by the Department of Biomedical Informatics at The Ohio State University. Each node of the cluster is equipped with dual 2.4 GHz Opteron 250 processors, 8 GB of RAM and two 250 GB SATA drives providing 500 GB of local storage via software RAID0. The nodes are interconnected with switched gigabit Ethernet and Infiniband.

The tests were performed using vertex declustering during ingestion; the vertex ownership knowledge was leveraged during the search phase. Additionally, the search performance tests were performed with an in-memory visited data structure, with the exceptions where noted. We wish to characterize the operation of the actual graph storage; the simplest way to obtain a fair comparison is to simply fix the visited data-structure. All the tests were performed with one of three graphs; two real-world semantic graphs, PubMed-S and

Graph	Vertices	Und. Edges	Min. Deg.	Max. Deg.	Avg. Deg.
PubMed-S	3,751,921	27,841,339	1	722,692	14.84
PubMed-L	26,676,177	259,815,339	1	6,114,328	19.48
Syn-2B	100,000,000	999,999,820	1	42,964	20.00

Table 5.1: Statistics for graphs used in experiments

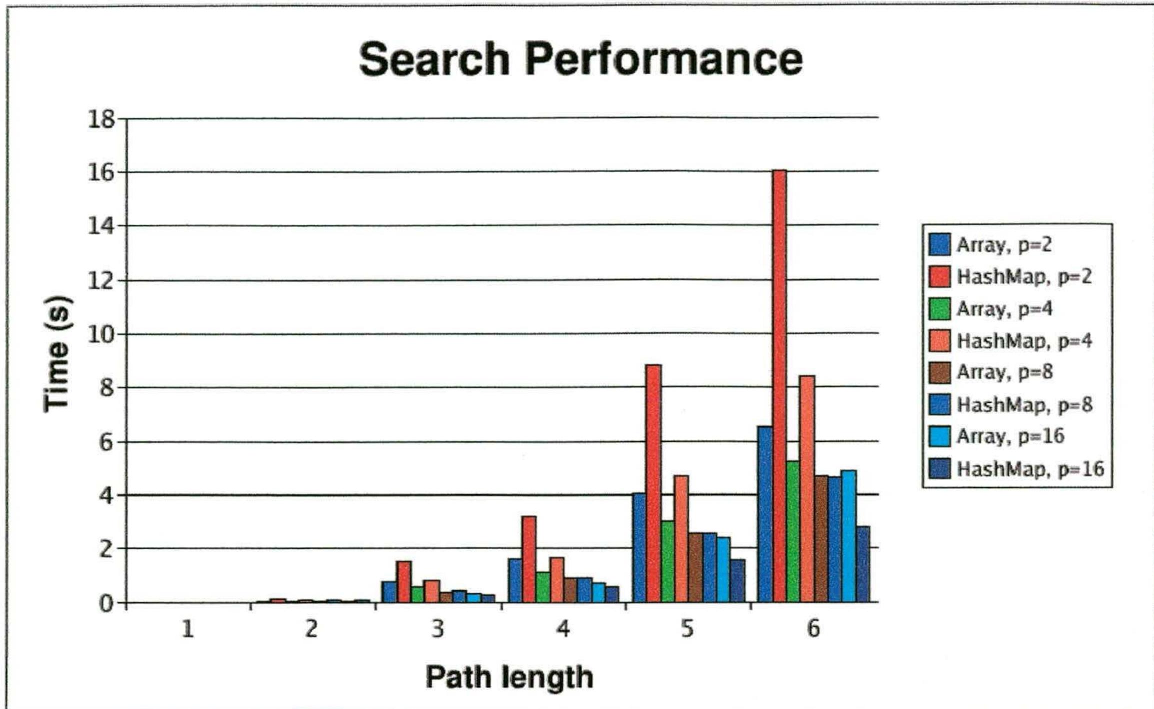


Figure 5.1: Search performance of in-memory GraphDB implementations on PubMed-S graph

were performed with one of three graphs; two real-world semantic graphs, PubMed-S and PubMed-L, were extracted from the PubMed document database, while the third, Syn-2B was created to exhibit the scale-free properties which MSSG targets. The graph information is given in table 5. While these graphs are smaller than the trillion-edge graphs MSSG targets, examining the outcome of these experiments is a necessary first step in order to scale to larger graphs.

The first results displayed in Figure 5.1 represent a base-line comparison of the search performance of the MSSG framework using in-memory implementations of GraphDB. In this experiment 100 random BFS queries were executed on 16 nodes against the PubMed-S graph, and the query execution times are averaged based on the path length between the

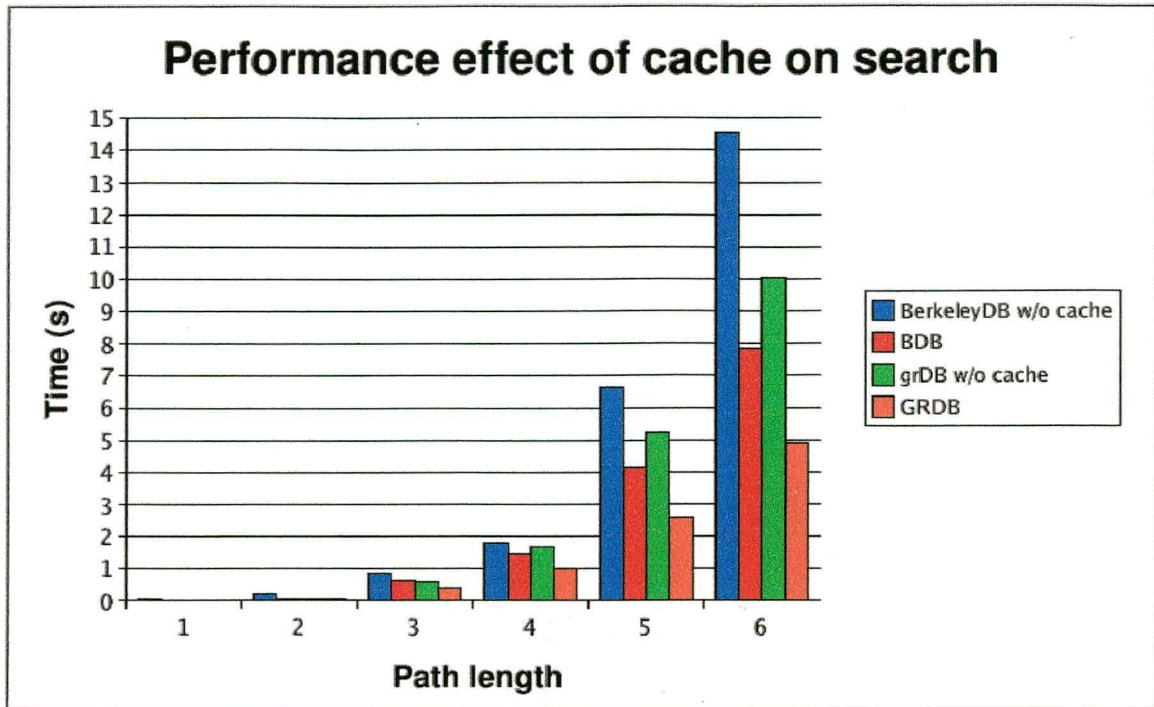


Figure 5.2: Search performance of BerkeleyDB and grDB on PubMed-S graph with and without cache

source and destination vertices. As seen in the figure, the Array graph storage performs much better than the HashMap implementation, as expected. The HashMap GraphDB storage requires a hash lookup to access the adjacency list of a vertex; with large graphs, this overhead becomes significant. This is especially true as the path length increases, since the size of the fringe at each search level increases exponentially. However, when increasing the number of processors, this overhead is spread over multiple processors and the difference between Array and HashMap is lessened.

The second set of experiments shows the importance of caching effects when dealing with out-of-core data structures. Figure 5.2 displays the average performance on 16 nodes of BerkeleyDB and grDB on 100 random queries against the PubMed-S graph, with their

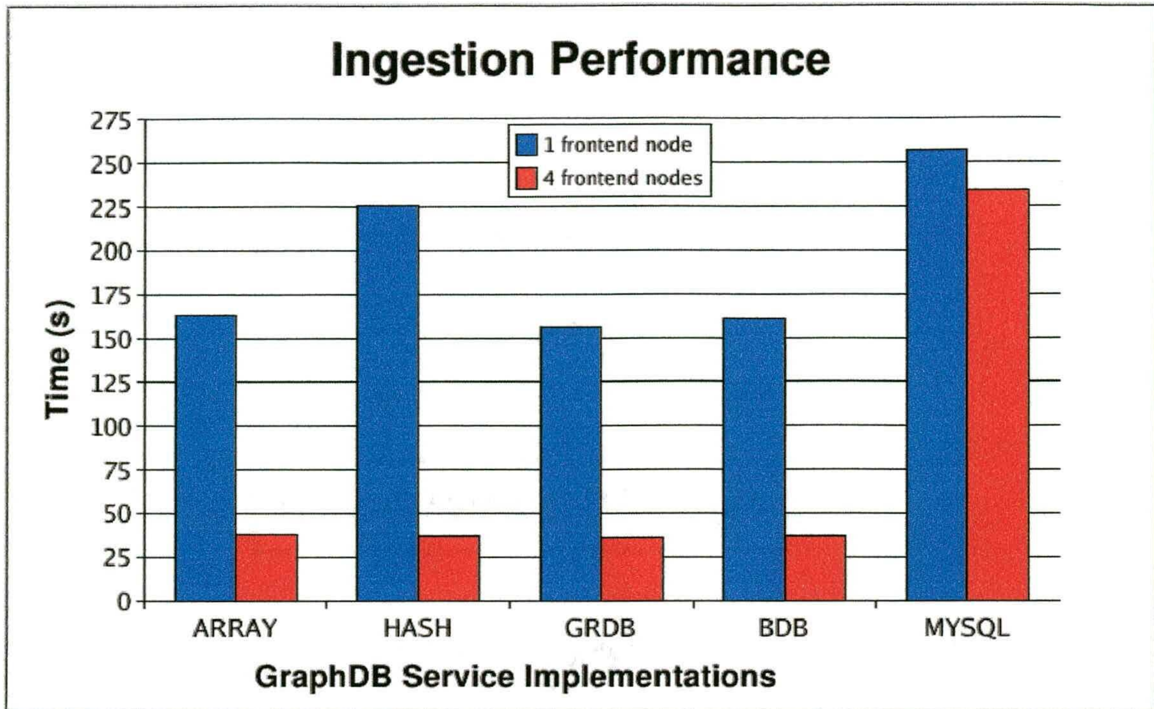


Figure 5.3: Ingestion performance comparison of five GraphDB implementations on PubMed-S graph

internal (block) caches enabled and disabled. As seen in the figure, caching can reduce the execution time up to 50% on both implementations, especially for longer path queries. Therefore, further results will only come from our cache-enabled out-of-core implementations.

Figure 5.3 displays a comparison of the five different GraphDB implementations on 16 nodes using the PubMed-S graph. To investigate the effect of increasing the number of front-end ingestion nodes, we repeated the ingestion of PubMed-S multiple times and varied the number of ingestion nodes. The result shown in Figure 5.3 shows that Array, BerkeleyDB, and grDB achieved similar performance in both cases (1 ingestion node vs 4 ingestion nodes). However, the HashMap and MySQL implementations were slower

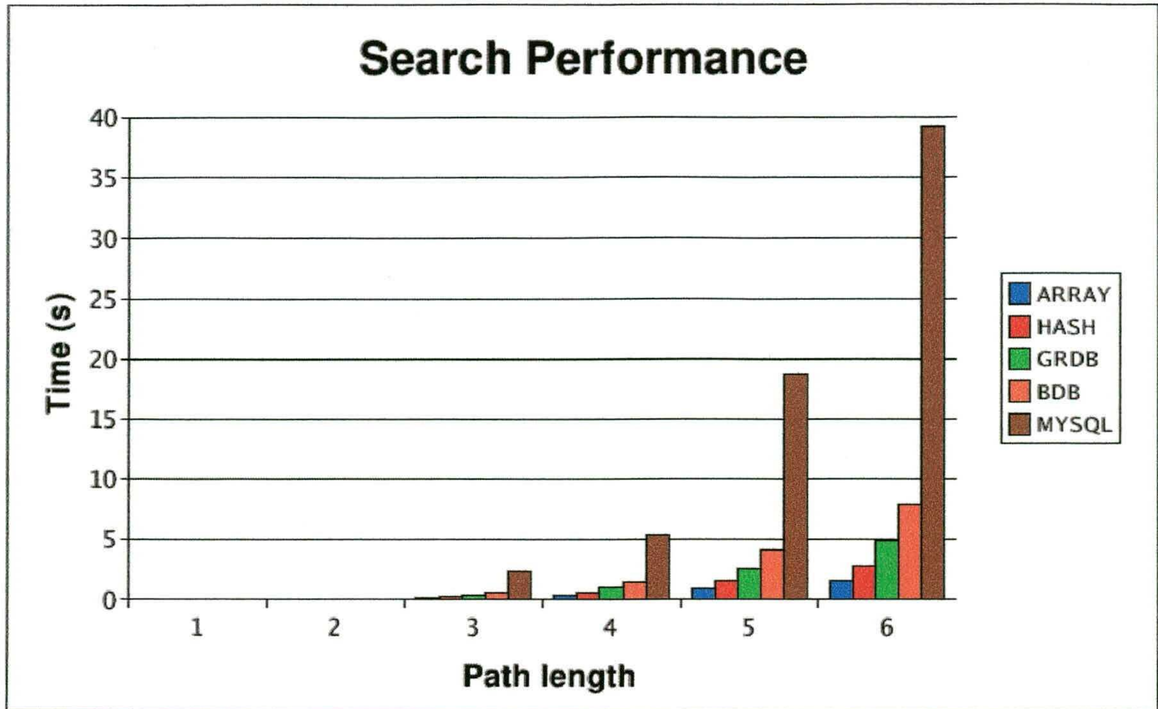


Figure 5.4: Search performance comparison of five GraphDB implementations on PubMed-S graph

when only 1 ingestion node was used. When using 1 ingestion node, the ingestion speed is clearly limited by the I/O and network performance of that node. Furthermore, even though the ingestion node reads a block (window) of edges and then distributes them to back-end nodes, edge ordering in the streaming input graph could negatively affect the load balance on the back-end nodes. By increasing the number of ingestion nodes, we both remove the bottleneck from the front-end node and also achieve better load balance on the back-end nodes. The results show that the ingestion performance is more or less the same for all approaches, except for MySQL, which is slower than all other GraphDB storage implementations.

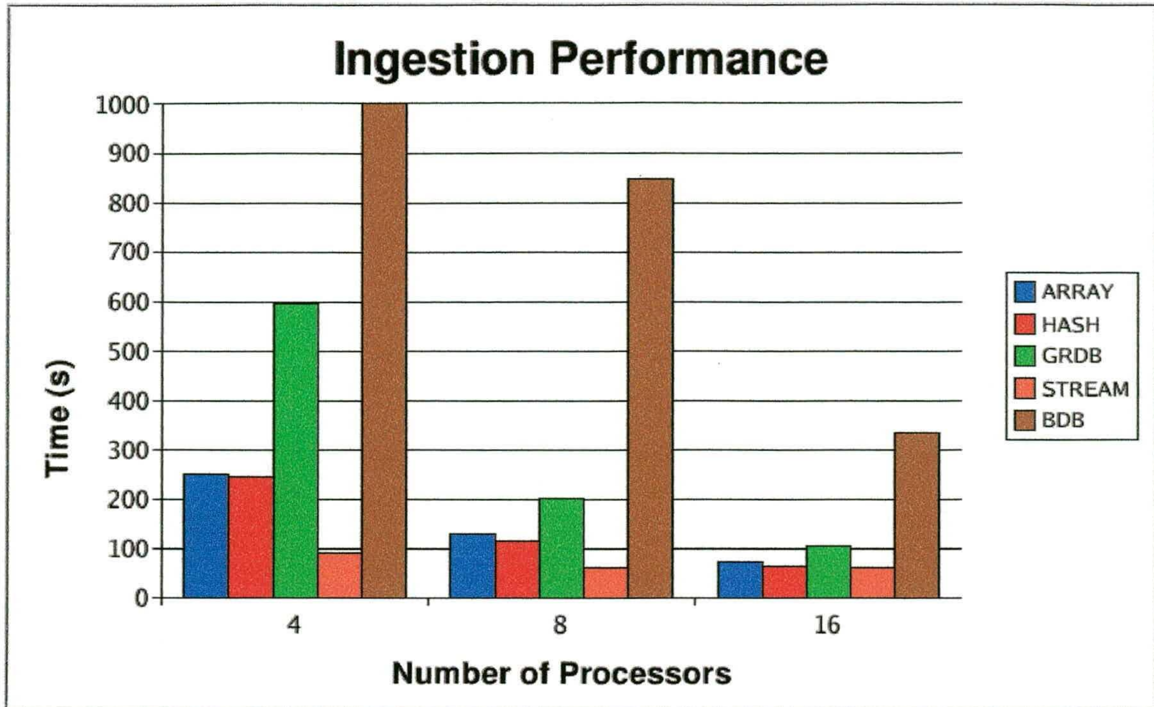


Figure 5.5: Ingestion performance comparison of five GraphDB implementations on PubMed-L graph: 8 front-end ingestion nodes, vary back-end storage nodes

As seen in Figure 5.4, the Array implementation gives the lowest search time. Not surprisingly, the second best results are achieved with the other in-memory implementation, HashMap. MySQL performs significantly worse than all other implementations. The fastest of the three out-of-core GraphDB implementations, grDB, performs an average of 33% faster than the next fastest out-of-core implementation, BerkeleyDB. When comparing grDB with the in-memory implementations, grDB is only 1.7 times slower than HashMap and about 2.9 times slower than Array, on average. Finally, the search times for short paths (and hence small fringe sizes) are negligible for all GraphDB implementations. As such, future results will only show longer path lengths.

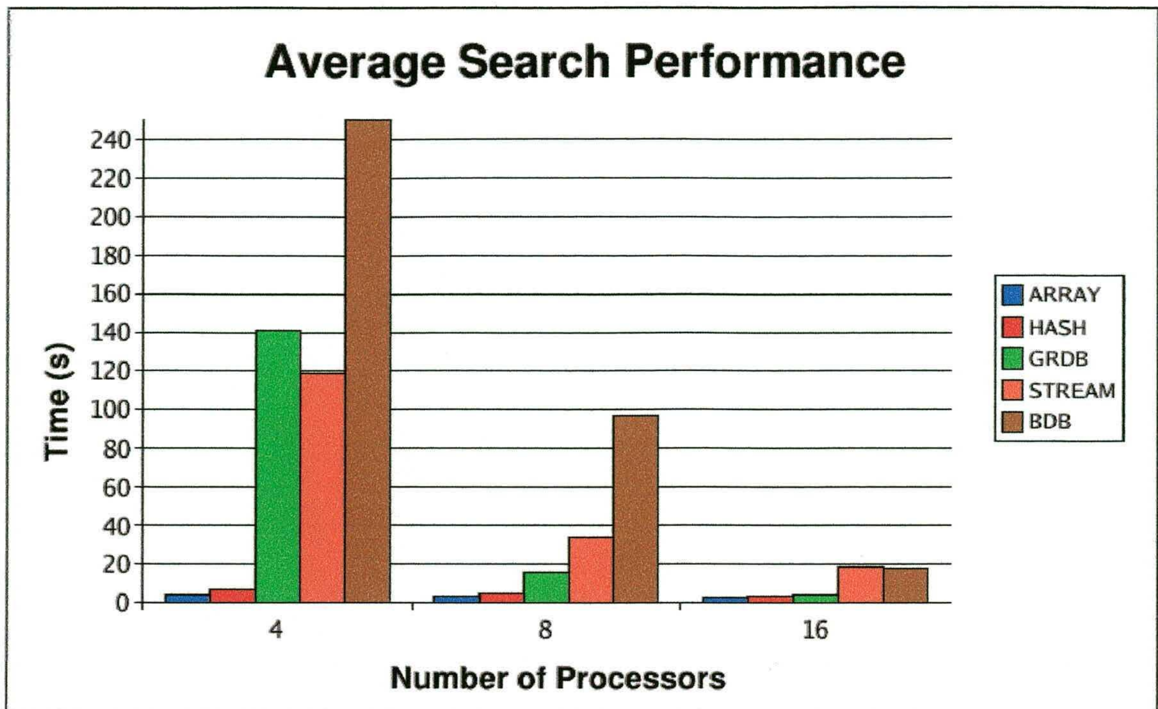


Figure 5.6: Execution time search performance comparison of five GraphDB implementations on PubMed-L graph

The results of the ingestion experiments in Figure 5.5 show that when the graph size increases, grDB begins to have a significant advantage over the BerkeleyDB database. The time for BerkeleyDB to ingest the graph is actually over 1,600 seconds, but the chart is zoomed to give some resolution to the lower values. Further, we see that the StreamDB instance has unrivaled ingestion performance. This is to be expected, since it is simply writing the edge set for each node directly to disk. What's more, the output format is more efficient than the ingestion node format for these experiments; the output format is binary, while the input data is ASCII.

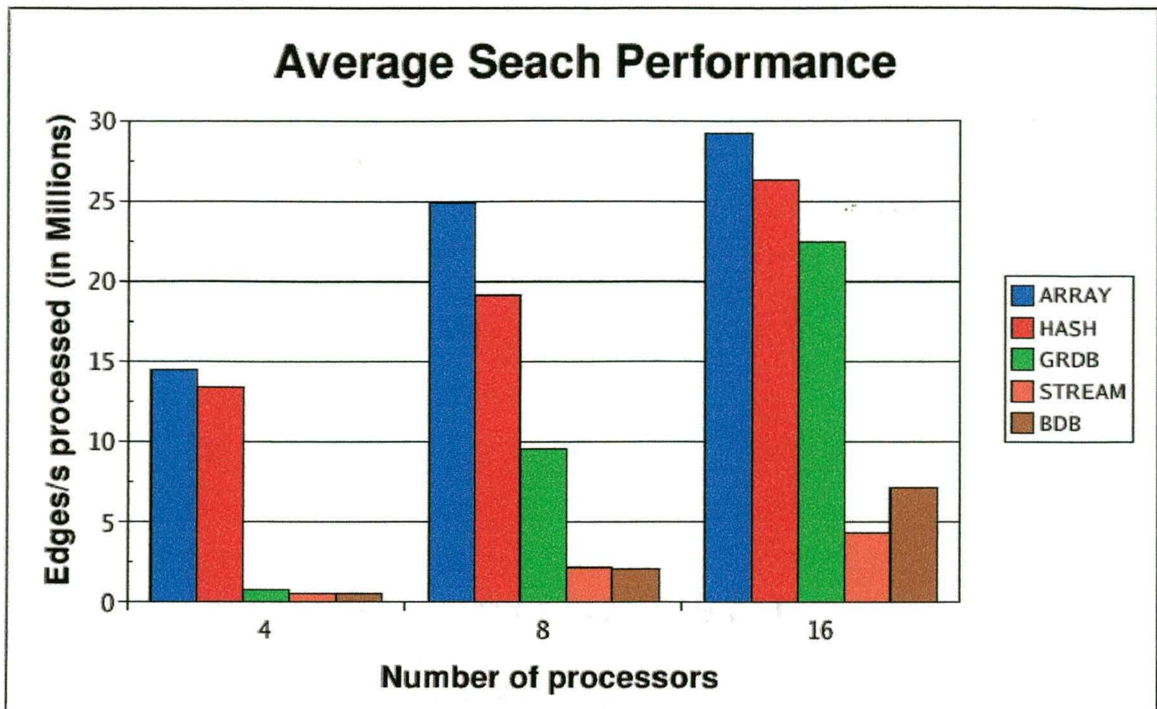


Figure 5.7: Edge/s search performance comparison of five GraphDB implementations on PubMed-L graph

Figure 5.6 shows the results of the search experiments. As before, the Array GraphDB implementation is the fastest of the five instances, followed closely by the HashMap instance. On 8 and 16 processors, grDB performs admirably, but the random access of the graph data forces the performance to drop below that of StreamDB on 4 nodes. In this way, StreamDB acts as a useful lower-bound, indicating when some extra effort needs to be applied to our other out-of-core database solutions to bring their performance to an appropriate level.

Figure 5.7 shows the aggregate edges processed per second during the search experiments. When visiting a large portion of the graph, as in the larger search paths, the Array implementation processes nearly 30 million edges per second. The grDB implementation

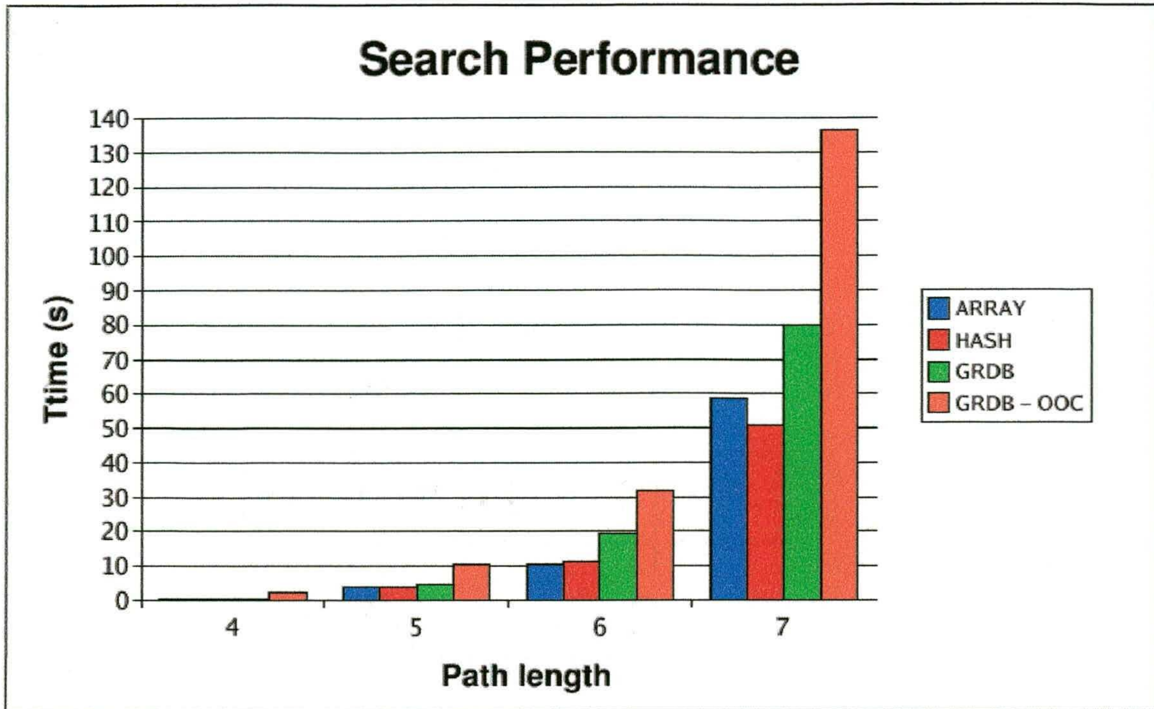


Figure 5.8: Execution time search performance for Syn-2B graph using grDB

reaches 20 million edges per second processed on 16 nodes, but this number drops significantly on 4 nodes. Additionally, while grDB is able to process more edges per second than the StreamDB instance, grDB's search time is actually higher than that of StreamDB. There is some room for improvement in the grDB database instance, when the grDB cache size becomes negligible compared to the size of the graph.

Figures 5.8 and 5.9 show the results of MSSG's performance when working with the Syn-2B graph. The trends seen before are upset somewhat, with the HashMap implementation's performance equaling or bettering that of the Array instance. This is surprising, but the effect of a cache on a randomly accessed data structure can start to rival that of a

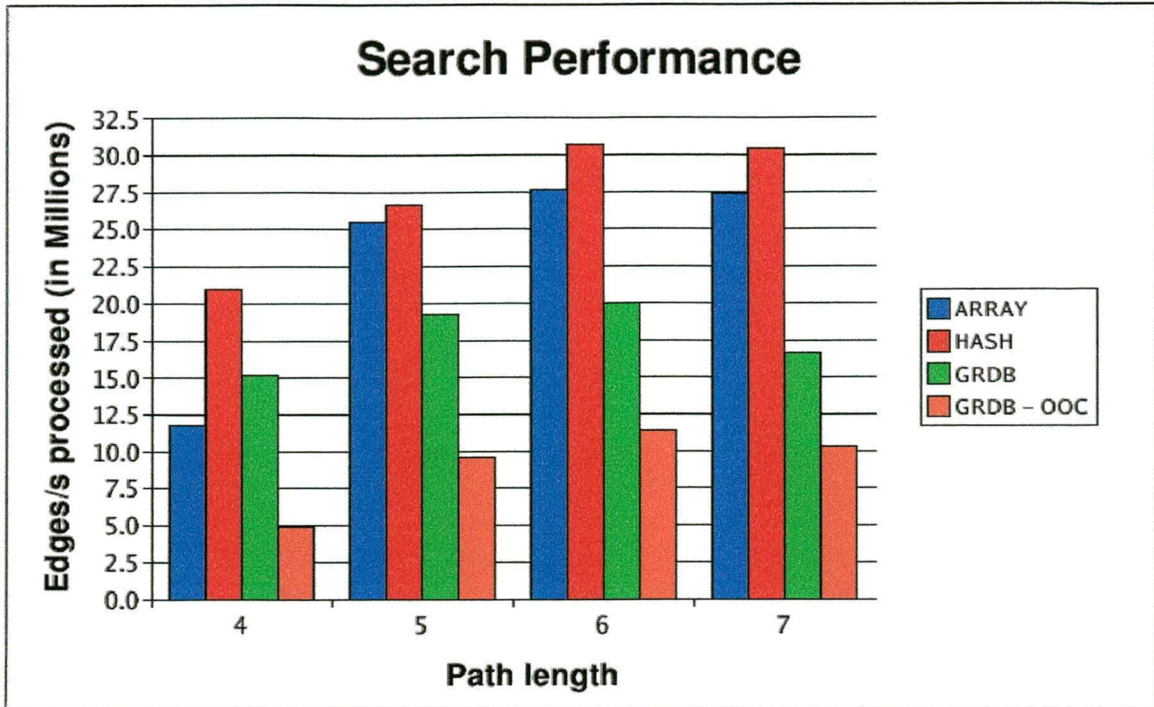


Figure 5.9: Edge/s search performance for Syn-2B graph using grDB

sequentially accessed data structure when those structures become very large, as is the case here. But the out-of-core solution still lags behind that of the in-memory solutions, as is expected.

Additionally, while the use of an external-memory visited data structure adversely affects the performance, this is expected also. Nevertheless, the MSSG system can clearly search very large graphs in reasonable time-frames and provides a high level of performance. When touching a large portion of the graph, as is done in the later levels of any breadth-first search on a scale-free graph, MSSG and grDB can process over 10 million edges per second.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this paper we presented a middleware framework, MSSG, for storing, accessing, and analyzing massive-scale semantic graphs. We proposed and developed a novel disk-based graph database, *grDB*, for massive scale-free graphs. We have also developed a parallel out-of-core breadth-first search algorithm. Experimental evaluations on large real-world semantic graphs and randomly generated large scale-free graphs show that the MSSG framework scales well. Also, grDB outperforms widely used open-source databases, such as BerkeleyDB and MySQL, in storage and retrieval of scale-free graphs.

A major goal of this research was to provide a flexible and efficient framework to allow the development and analysis of different graph algorithms which are intended to operate on graphs of previously intractable size. We believe we have provided such a framework. As future work, we will now move to the development of more complex approaches for improving the efficiency of the MSSG framework, in order to move closer to our goal of offering an environment to truly analyze graphs of enormous size.

It is clear when dealing with such massive amounts of data that a good clustering of the graph data is imperative. Due to the properties of scale-free graphs, longer path searches will often search large portions of the graph. This fact will require an efficient solution to store vertices which are close to each other in the graph to be stored close to each other

on the disk. Additionally, while breadth-first search does not perform a large amount of computation on the graph data, overlapping the disk accesses with the computation that is required will still lead to some additional performance.

More experiments on larger graphs are an important part of validating our work in this area. Simply gathering or generating graphs of this size is an interesting problem, one which could be helped by the MSSG framework, and its high ingestion and storage speed.

BIBLIOGRAPHY

[1] The ABACUS project. *http://www.cs.cmu.edu/~amiri/abacus.html.*

[2] James Abello, Adam L. Buchsbaum, and Jeffery Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

[3] James Abello, Mauricio G. C. Resende, and Sandra Sudarsky. Massive quasi-clique detection. In *LATIN*, pages 598–612, 2002.

[4] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 81–91, New York, NY, USA, 1998. ACM Press.

[5] Martin Aeschlimann, Peter Dinda, Julio Lopez, Bruce Lowekamp, Loukas Kallivokas, and David O'Hallaron. Preliminary report on the design of a framework for distributed visualization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1833–1839, Las Vegas, NV, June 1999.

[6] Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson. Dynamic function placement for data-intensive cluster computing. In *the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[7] Lars Arge, Octavian Procopiuc, and Jeffrey Scott Vitter. Implementing I/O-efficient data structures using TPIE. In *ESA*, pages 88–100, 2002.

[8] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster i/o with river: making the fast case common. In *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 10–22, New York, NY, USA, 1999. ACM Press.

[9] David A. Bader and Guojing Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *J. Parallel Distrib. Comput.*, 65(9):994–1006, 2005.

[10] A.-L. Barabási and A. Réka. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[11] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r* tree: An effcient and robust access method for points and rectangles. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, May 1990.

[12] Henrique Andrade Beomseok Nam and Alan Sussman. Multiple range query optimization with distributed cache indexing. In *Proceedings of the Intl. Conf. for High Perf. Computing, Networking, Storage and Analysis (SC06)*, 2006.

[13] S. Berchtold, C. Bohm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 142–153, Seattle, Washington, June 1998.

[14] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: an index structure for high-dimensional data. In *International Conference on Very Large Databases VLDB*, pages 28–39, Bombay, India, September 1996.

[15] Michael D. Beynon, Renato Ferreira, Tahsin Kurc, Alan Sussman, and Joel Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the Eighth Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, pages 119–133. National Aeronautics and Space Administration, March 2000. NASA/CP 2000-209888.

[16] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, October 2001.

[17] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Alan Sussman, and Joel Saltz. A component-based implementation of iso-surface rendering for visualizing large datasets. Technical Report CS-TR-4249 and UMIACS-TR-2001-34, University of Maryland, Department of Computer Science and UMIACS, May 2001.

[18] Shahid Bokhari, Benjamin Rutt, Pete Wyckoff, and Paul Buerger. An evaluation of the osc fastt600 turbo storage pool. Technical Report OSUBMI_TR_2004_n02, The Ohio State University, Department of Biomedical Informatics, Sep 2004.

[19] B. Bollobas. *Random Graphs*. Cambridge University Press, 2001.

[20] Erik G. Boman, Doruk Bozdağ, Umit Catalyurek, Assefaw H. Gebremedhin, and Fredrik Manne. A scalable parallel graph coloring algorithm for distributed memory computers. In *EuroPar 2005*, 2005.

[21] Doruk Bozdağ, Umit Catalyurek, Assefaw H. Gebremedhin, Fredrik Manne, Erik G. Boman, and Füsun Özgüner. A parallel distance-2 graph coloring algorithm for distributed memory computers. In *The 2005 International Conference on High Performance Computing and Communications (HPCC-05)*, 2005.

[22] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery Westbrook. On external memory graph traversal. In *SODA*, pages 859–860, 2000.

[23] Common Component Architecture Forum. *http://www.cca-forum.org*.

[24] Kaushik Chakrabarti and Sharad Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. In *ICDE*, pages 440–447, 1999.

[25] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 139–149, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[26] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Symposium on Discrete Algorithms*, pages 139–149, 1995.

[27] Yi-Jen Chiang, Claudio T. Silva, and William J. Schroeder. Interactive out-of-core isosurface extraction. *vis*, 00:167, 1998.

[28] Terry W. Clark, James Andrew McCammon, and L. Ridgway Scott. Parallel molecular dynamics. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 338–344, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[29] Richard Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[30] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and U.V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proceedings of 20th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006.

[31] Ralf Diekmann, Uwe Dralle, Friedhelm Neugebauer, and Thomas Romke. Padfem: A portable parallel FEM-tool. In *HPCN Europe*, pages 580–585, 1996.

[32] Tina Eliassi-Rad and Edmond Chow. Using ontological information to accelerate search in large semantic graphs: A probabilistic approach. Technical Report UCRL-CONF-20200, Lawrence Livermore National Laboratory, 2005.

[33] R.W. Floyd. Permuting information in idealized two-level storage. *Complexity of Computer Computations*, pages 105–109, 1972.

[34] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD'84*, pages 47–57. ACM Press, May 1984.

[35] Don Heller. A survey of parallel algorithms in numerical linear algebra. *SIAM Review*, 20(4):740–777, oct 1978.

[36] Bruce Hendrickson and Steve Plimpton. Parallel many-body simulations without all-to-all communication. *Journal of Parallel and Distributed Computing*, 27(1):15–25, 1995.

[37] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[38] Carsten Isert and Karsten Schwan. ACDS: Adapting computational data streams for high performance. In *14th International Parallel & Distributed Processing Symposium (IPDPS 2000)*, pages 641–646, Cancun, Mexico, May 2000.

[39] Hong Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333, New York, NY, USA, 1981. ACM Press.

[40] Mark T. Jones and Paul Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, 1993.

[41] G. Karypis, K. Schloegel, and V. Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library, version 3.1. Technical report, Dept. Computer Science, University of Minnesota, 2003. http://www-users.cs.umn.edu/ karypis/metis/-parmetis/download.html.

[42] Tamara Kolda and et. al. Data sciences technology for homeland security information management and knowledge discovery. Technical Report UCRL-TR-208926, Lawrence Livermore National Laboratories, 2004. Report of the DHS Workshop on Data Sciences, September 22-23, 2004.

[43] Elana Konstantinova. Chemical hypergraph theory. Lecture Notes from Combinatorial & Computational Mathametics Center, http://com2mac.postech.ac.kr/, 2000.

[44] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal: Very Large Data Bases*, 3(4):517–542, 1994.

[45] D. B. Lomet and B. Salzberg. The hb-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1998.

[46] Ulrich Meyer and Norbert Zeh. I/o-efficient undirected shortest paths. In *ESA*, pages 434–445, 2003.

[47] MySql Database. *http://www.mysql.com/.*

[48] Netezza Data Warehouse Appliance. *http://www.netezza.com/.*

[49] Robert Niewiadomski, José Nelson Amaral, and Robert C. Holte. A parallel external-memory frontier breadth-first traversal algorithm for clusters of workstations. In *ICPP*, pages 531–538, 2006.

[50] Mark H. Nodine, Michael T. Goodrich, and Jeffrey Scott Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

[51] Mark H. Nodine and Jeffrey Scott Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 120–129, New York, NY, USA, 1993. ACM Press.

[52] Ron Oldfield and David Kotz. Armada: A parallel file system for computational grids. In *Proceedings of CCGrid2001: IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001. IEEE Computer Society Press.

[53] Frank Olken. Graph data management for molecular biology. *OMICS*, 7(1):75–78, 2003.

[54] Parallel Boost Graph Library. *http://osl.iu.edu/research/pbgl/.*

[55] Beth Plale and Karsten Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *IEEE International High Performance Distributed Computing (HPDC)*, August 2000.

[56] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics, 1993.

[57] Michael J. Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Comput. Surv.*, 16(3):319–348, 1984.

[58] Sanguthevar Rajasekaran and John H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989.

[59] Emad Ramadan, Arijit Tarafdar, and Alex Pothen. A hypergraph model for the protein complex network in the yeast. In *Proceedings of 186th International Parallel and Distributed Processing Symposium (IPDPS), Third Workshop on High Performance Computational Biology*, Santa Fe, NM, April 2004.

[60] J. T. Robinson. The kdb-tree: A search structure for large multi-dimensional dynamic indexes. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 10–18, 1981.

[61] Sleepy Cat Software. Berkeley DB. http://www.sleepycat.com/.

[62] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. pages 161–179, 1999.

[63] Ertem Tuncel, Hakan Ferhatosmanoglu, and Kenneth Rose. Vq-index: An index structure for similarity searching in multimedia databases.

[64] Jeffrey Scott Vitter. External memory algorithms and data structures. In James Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms and Visualization*, pages 1–38. American Mathematical Society Press, Providence, RI, 1999.

[65] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. Technical Report Technical report DUKE–TR–1993–01, 1993.

[66] Steven H. Watts, Duncan J.and Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, June 1998. 10.1038/30918.

[67] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *International Conference on Data Engineering ICDE*, pages 516–523, New Orleans, LA, March 1996.

[68] Wanhong Xu, Larkshmi Krishnamurthy, Murat Tasan, Gultekin Özsoyoglu, Joseph H. Nadeau, Z. Meral Özsoyoglu, and Greg Schaeffer. Pathways database system: An integrated set of tools for biological pathways. In *SAC*, pages 96–102, 2003.

[69] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Proceedings of SC2005 High Performance Computing, Networking, and Storage Conference*, 2005. Gordon Bell Finalist.