A DISK-BASED GRAPH DATABASE SYSTEM WITH INCREMENTAL STORAGE LAYOUT OPTIMIZATION

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Doğukan Çağatay

January, 2016

A DISK-BASED GRAPH DATABASE SYSTEM WITH
INCREMENTAL STORAGE LAYOUT OPTIMIZATION
By Doğukan Çağatay
January, 2016

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Buğra Gedik (Advisor)

Asst. Prof. Dr. Can Alkan

Asst. Prof. Dr. Gültekin Kuyzu

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

A DISK-BASED GRAPH DATABASE SYSTEM WITH INCREMENTAL STORAGE LAYOUT OPTIMIZATION

Doğukan Çağatay
M.S. in Computer Engineering
Advisor: Assoc. Prof. Dr. Buğra Gedik
January, 2016

The world has become ever more connected, where the data generated by people, software systems, and the physical world is more accessible than before and is much larger in volume, variety, and velocity. In many application domains, such as telecommunications and social media, live data recording the relationships between people, systems, and the environment is available. This data often takes the form of a temporally evolving graph, where entities are the vertices and the relationships between them are the edges. For this reason, managing dynamic relationships represented by a graph structure has been a common requirement for modern data processing applications. Graph databases gained importance with the proliferation of such data processing applications. In this work, we developed a disk-based graph database system, which is able to manage incremental updates on the graph structure. The updates arrive in a streaming manner and the system creates and maintains an optimized storage layout for the graph in an incremental way. This optimized storage layout enables the database to support traversal based graph algorithms more efficiently, by minimizing the disk I/O required to execute them. The storage layout optimizations we develop aim at taking advantage of spatial locality of edges to minimize the traversal I/O cost, but achieves this in an incremental way as new edges/vertices get added and removed.

Keywords: Disk-based graph database, Data streaming, Storage layout.

ÖZET

ARTIMLI DEPOLAMA DÜZEN OPTİMİZASYONLU DİSK TABANLI ÇİZGE VERİTABANI

Doğukan Çağatay
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Doç. Dr. Buğra Gedik
Ocak, 2016

Artık insanların, yazılım sistemlerinin ve fiziksel dünyanın daha önce hiç olmadığı kadar büyük bir hızda, hacimde ve çeşitlilikte veri ürettikleri, herşeyin bağlı ve daha erişilebilir olduğu bir dünyada yaşıyoruz. Telekomünikasyon, sosyal medya ve canlı veri kaydı gibi bir çok uygulama alanında insanlar, sistemler ve bulundukları ortamlar arasında kolayca bir ilişki bulunabilir. Bu ilişkiler çoğunlukla, varlıkları düğümler, aralarındaki ilişkiler ise kenarları gösterecek şekilde zamanla gelişen bir çizge olarak karşımıza çıkabilmektedir. Bu yüzden, bir çizge yapısı içinde dinamik değişebilen ilişkileri yönetmek, modern veri işleme sistemlerinin ortak gereksinimlerinden biri olmuştur. Veri işleme sistemlerinin yaygınlaşması da çizge veritabanlarının önem kazanmasına sebep olmuştur. Bu çalışmada, artımlı güncellemeleri destekleyen, disk tabanlı bir çizge veritabanı geliştirdik. Veritabanına çizge güncelleştirmeleri (ekleme/çıkarma) bir akış şeklinde gelir, sistem gelen veriye göre bir depolama düzeni oluşturur ve en optimize olacak şekilde bu düzeni yönetir. Böylece, optimize depolama düzeni üzerinde düşürülen disk girdi-çıktısı, gezinme tipi algoritmaların daha verimli olarak koşulabilmesini sağlar. Geliştirdiğimiz depolama düzeni optimizasyonları, artımlı biçimde çizge güncellemeleri gelirken oluşan çizge kenarlarının boyutsal lokalitelerini hesaba katarak, çizge üzerinde gezinme sırasında oluşan disk girdi-çıktı sayısını düşürmeyi amaçlamaktadır.

Anahtar sözcükler: Disk tabanlı çizge veritabanı, Veri akımı, Depolama düzeni.

Acknowledgement

First of all, I would like to thank my supervisor, Assoc. Prof. Dr. Buğra Gedik for his guidance, patience, and support.

I am grateful to my jury members, Asst. Prof. Dr. Can Alkan and Asst. Prof. Dr. Gültekin Kuyzu for reading and reviewing this thesis.

I would like to acknowledge TÜBİTAK for their financial support.

I would like to thank my family and my fiance for their endless love and supporting me with all my decisions during my thesis work. And my friends Mecit, Begüm, Abdurrahman, and Semih, thank you for being there for me whenever I needed you.

Contents

1 Introduction **1**

2 System Overview **5**

2.1 Basic Concepts 5

2.1.1 Graph Structure 5

2.1.2 Storage Backend 7

2.2 System Architecture 10

2.2.1 Graph Manager 11

2.2.2 Graph 13

2.2.3 Node 15

2.2.4 Edge 16

2.2.5 Storage Connector 18

2.2.6 Subgraph 19

2.2.7 Buffer Manager 22

2.2.8 Memory Buffer 23

2.2.9 Block Manager 23

2.3 Storage Layout 24

2.3.1 Layout Policies 24

2.3.2 Select Policy 24

2.3.3 Split Policy 25

3 Experimental Evaluation **29**

3.1 Experimental Setup 29

3.2 Experimental Scenarios 30

3.3 Results 31

3.3.1 Insertion Results 31

3.3.2 Query Results 32

3.4 Evaluation Summary 41

4 Related Work **45**

5 Conclusion **47**

List of Figures

2.1 The graph structure used in GraphDB 6

3.1 Insertion Result Graphs 33

3.2 PageRank Query on First Available Select Policy created layout with edge degree of 5. 35

3.3 PageRank Query on First Available Select Policy created layout with edge degree of 10. 35

3.4 PageRank Query on Random Select Policy created layout with edge degree of 5. 35

3.5 PageRank Query on Random Select Policy created layout with edge degree of 10. 36

3.6 Connected Components query on First Available Select Policy created layout with edge degree of 5. 37

3.7 Connected Components query on First Available Select Policy created layout with edge degree of 10. 37

3.8 Connected Components query on Random Select Policy created layout with edge degree of 5. 37

3.9 Connected Components Query on Random Select Policy Created Layout with Edge Degree of 10. 38

3.10 BFS Walk Query on First Available Select Policy Created Layout with Edge Degree of 5. 38

3.11 BFS walk query on First Available Select Policy created layout with edge degree of 10. 40

3.12 BFS walk query on Random Select Policy created layout with edge degree of 5. 40

3.13 BFS walk query on Random Select Policy created layout with edge degree of 10. 40

3.14 Random walk quert on First Available Select Policy created layout with edge degree of 5. 42

3.15 Random walk quert on First Available Select Policy created layout with edge degree of 10. 42

3.16 Random walk quert on Random Select Policy created layout with edge degree of 5. 42

3.17 Random walk quert on Random Select Policy created layout with edge degree of 10. 43

List of Tables

3.1 Generated disk layouts with select and split policies. 30

3.2 Random Split Policy average running time differences compared to other policies on insertion with First Available Select Policy. . 31

3.3 Random Split Policy average running time differences compared to other policies on insertion with Random Select Policy. 31

3.4 Random Select Policy running time compared to the First Available Select Policy on average. 32

3.5 Split Policy comparison for PageRank algorithm for Random Select Policy against Random Split Policy. 34

3.6 Split Policy comparison for PageRank algorithm for First Available Select Policy against Random Split Policy. 34

3.7 Split Policy comparison for Connected Components Algorithm, Random Select Policy against Random Split Policy. 36

3.8 Split Policy comparison for Connected Components Algorithm for First Available Select Policy against Random Split Policy 36

3.9 Split Policy comparison for BFS algorithm for Random Select Policy against the Random Split Policy. 39

3.10 Split Policy comparison for BFS algorithm for First Available Select Policy against Random Split Policy. 39

3.11 Split Policy comparison for Random Walk algorithm for Random Select Policy against Random Split Policy 41

3.12 Split Policy comparison for Random Walk algorithm for First Available Select Policy against Random Split Policy 41

Chapter 1

Introduction

Today, the world is experiencing a data deluge. Improvements in technology has led us to a point where data became the most significant role-player for business as well as research. Data generation is ubiquitous now, as various types of devices generate streams of data, in a continuous way. Examples include airplane motors, servers, computers, cell phones, etc. As humans, we also generate data with our actions on social media such as Facebook and Twitter, and via our queries on search engines, such as Google. The increase in the amount of data produced has reached a point where it is impractical to store and process the data with conventional methods in most of the cases.

An important portion of this data takes the form of *graphs*, where nodes in the graph represent entities, and the relationships between these entities represent edges. Accordingly, there has been various recent works on scalable graph processing, management, and storage systems. Examples include Neo4j [1], Hadoop [2], Giraph [3], Pregel [4], GraphChi [5], GraphLab [6], etc. Each of these systems has their differences in their graph processing schemes, graph algorithm execution models, data structures, programming models and graph storage subsystems.

In this work, we present a graph database system called *GraphDB*, which stores a graph structure on disk and accepts *streaming updates* to continuously refresh

this structure so that ad-hoc queries can be answered in an efficient manner. GraphDB dynamically changes the graph layout on disk with incoming streaming data, using incremental disk layout optimization algorithms. Updating the on-disk graph layout while the data is streaming requires creating and reordering graph partitions in an on-the-fly manner, so that online queries can be answered more efficiently.

Storing/querying data into/from graph databases requires organizing the graph data using a series of disk blocks. This can be seen as a form of graph partitioning and ideally nodes that are "close" in terms of their connections should be co-located on the same disk blocks as much as possible. In other words, the goal is to store graph data on the disk with high *locality*. If data is static, the disk blocks can be created by looking at the already existing connections in the graph. There are very good performing partitioning algorithms on static graphs, such as METIS [7] and ParMETIS [8]. However, streaming graph data dynamically changes, and thus a static partitioning cannot be used. As a result, GraphDB aims at dynamically updating the disk blocks in order to maintain high locality in the presence of streaming updates.

Graphs are ideal for representing relationship data and graph databases are ideal for accessing data associated with a related set of nodes within a graph. For example, assume that Facebook stores user data in a graph database and represent a user as a node in the graph. When we want to access the data associated with a user and all its friends, the graph database will be able to answer this query by reading a few disk blocks, since due to locality of storage, data for these users will mostly be co-located in a few disk blocks. However, in relational databases, the data for a user and its friends may be scattered over different disk blocks of a table. The latter is because the locality of nodes cannot be captured by a simple key order in a relation table.

Most graph algorithms are *traversal-based*, which is an access pattern efficiently supported by graph databases. In a typical graph traversal, at each iteration, you visit a node, access its data, perform an operation and probably move on to perform similar operators for one or more neighbors of the current node. If

this kind of access pattern results in too many disk accesses, the traversal will be very slow. GraphDB aims to optimize the on-disk graph layout so as to lower the number of disk access operations needed when executing graph algorithms that rely on traversals. Number of disk access operations is key to speed up the processing of a graph, because access speed of a disk is significantly lower than that of memory. Even though access speed is drastically increased with *Solid State* disks, frequent data retrieval from disk is still an expensive operation.

In order to provide access to a large graph, GraphDB partitions the graph into sub-graphs and creates disk blocks out of these sub-graphs. Main memory is used a buffer cache to store sub-graphs. However, since GraphDB is designed for incremental updates, this partitioning is incrementally updated as new nodes and edges are inserted and removed. This online repartitioning aims at improving the disk layout of the graph in order to reduce the cost of queries.

The design of GraphDB introduces several challenges. First, online updates to the graph structure means that we cannot apply traditional methods for partitioning the graph and creating the on-disk layout. Instead, *incremental* algorithms are required for this purpose. Second, the incremental algorithms should be lightweight, so as not to significantly increase the cost of graph updates. Third, the incremental partitioning algorithm should result in a disk layout that decreases the disk access cost for traversal based graph algorithms.

Our contributions in this work include creating a system called GraphDB that can effectively store a dynamically growing graph structure on-disk and provide efficient access to it for running traversal-based graph algorithms. We assume that at first we don't have any data and the graph structure doesn't exist and as the data starts to arrive, GraphDB starts to create the graph structure, incrementally updating its on-disk layout to adapt to the structure of the data. For this purpose we propose effective and lightweight partition selection, partition splitting, partition reordering algorithms, and disk layout creation algorithms.

In summary, contributions of this thesis include:

- We developed a disk-based graph database system that supports incremental updating of the on-disk layout to improve traversal-based graph data access.
- We developed lightweight incremental graph partitioning algorithms for updating the on-disk layout of the graph as graph updates are processed.
- We experimentally evaluated the performance of our system and showcased the effectiveness of our algorithms and techniques.

The rest of this thesis is organized as follows:

First, an overview of the system architecture of GraphDB is given in Chapter 2. Basic concepts are presented in Section 2.1 and a detailed system architecture is given in Section 2.2. Experimental evaluation follows in Chapter 3. The related work is given in Chapter 4. Finally, Chapter 5 concludes the thesis.

Chapter 2

System Overview

2.1 Basic Concepts

2.1.1 Graph Structure

We assume that the graph or graphs to be managed by our system are large enough that a memory-based solution is not feasible and thus the graphs are to be stored on the disk. Our general approach is to maintain a graph as set of partitions such that each partition is a relatively small piece of the graph that fits in memory. Partitioning the graph is also important from storage point of view. Storing and loading a large graph from storage would take too much time. Instead, reading smaller partitions of the graph as they are needed, is a much effective approach from performance point of view. Hence, our graph structure consists of partitions that composes the whole graph. We call those partitions *subgraph*s in our system and throughout this thesis. In our graph structure, we don't store individual nodes, but subgraphs. The subgraph holds the nodes and edges associated with its partition of the nodes. *Cross edges*, which are edges that connect vertices belonging to different subgraphs, are stored in both subgraphs. In order to be able to tie any node in a graph to a subgraph we use a global index over the nodes that maps nodes to their subgraphs. This global index is also

persisted as part of the graph storage. Since a graph in our system is composed of subgraphs and a node index, all we need to store is those two types of data on the storage medium. Therefore, the storage method we use to store the graph is composed of two components, a block which represents a subgraph on the storage, and an index to identify which node is in which block. The high level view of the graph structure is given in Figure 2.1.

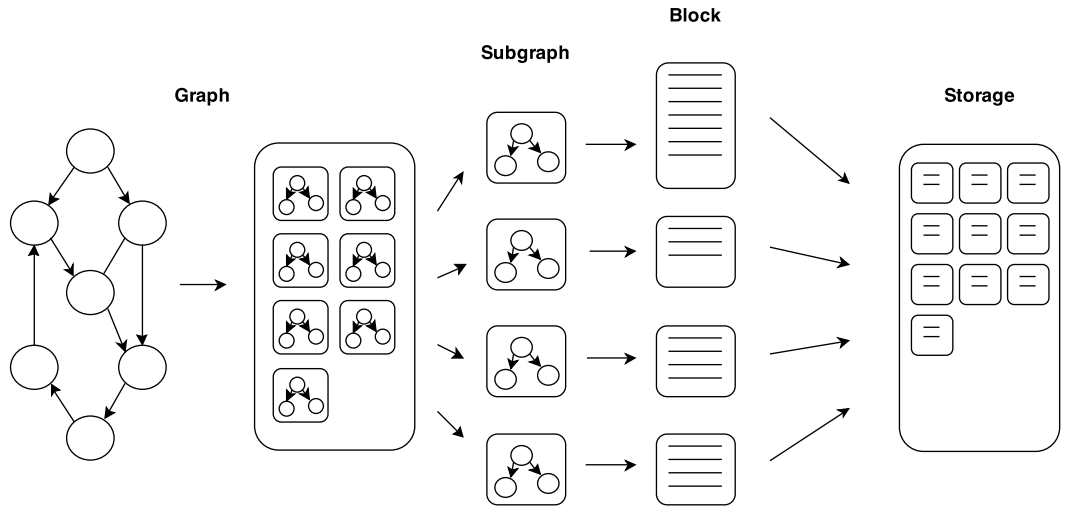


Figure 2.1: The graph structure used in GraphDB. Graph is partitioned into subgraphs. Subgraphs are in-memory data structures that hold the nodes and the edges. In order to store the graph on storage and load from storage we serialize/deserialize subgraphs/blocks.

In the process of storing a graph in our database, we are serializing subgraphs into disk blocks, together with the global index. Loading the graph from storage is the opposite of this process. The system loads blocks from the storage and deserializes graph data from the blocks into subgraphs with the help of the global index stored on the storage. The details of serialization of subgraphs and deserialization of blocks are explained in Section 2.2.6 and 2.2.6.

In GraphDB, the block size is a system configurable constant. This ensures that a fixed number of partitions fit into the main memory reserved for the operation of the database. Recall that GraphDB is not designed to load all the blocks from storage to main memory at any point, unless the graph fits in to the memory in the first place. Even if the graph fits into the memory, GraphDB will not load the

unneeded blocks of the graph. However, having a constant block size creates a challenge for some graphs. Particularly, when there are nodes that have so many edges that a single node does not fit into one block when serialized, we have a problem. The edges of a node take considerable space in a block, because each edge take a fixed space plus additional space for its attributes. The purpose of creating GraphDB is to work with very large graphs, so this situation is very likely to occur in real-world graphs with power law properties. Such graphs contain a few nodes with very large number of connections. If this kind of situation is observed, our one-to-one subgraph block correspondence wouldn't work. Hence, we pair the subgraphs and blocks in a way that a subgraph might represent more than one block in the memory but one or more associated blocks will create only one subgraph. This type of multiple block correspondence of a subgraph only happens when a node doesn't fit in a block when serialized.

As new nodes and edges are added to a graph, the subgraphs start to grow. Subgraphs that get too big to fit into a single block are split to create smaller subgraphs. In the rare case that a single node becomes too large in size (in terms of its neighbors), it starts to occupy its own subgraph. Such subgraphs are marked and does not admit other nodes. With this solution, when a marked subgraph is serialized, it produces potentially more than one disk block. In GraphDB, there is no limit to the number of disk blocks that can be created from a marked subgraph. However, since a marked subgraph has to fit into memory, the only restriction is that a single node, with its edges and edge properties, has to fit into the main memory reserved for the operation of GraphDB. The main memory reserved for holding the subgraphs is referred to as the *subgraph buffer*.

2.1.2 Storage Backend

In an earlier iteration of GraphDB, the system was designed to support only a file system backend for storage. However, there are many storage systems that we could make use of and there will be new ones in the future. Since the GraphDB system is designed to process large graphs, alternative storage

systems might be necessary in some cases. For instance, distributed storage systems like HDFS [2], HBase [9], Cassandra [10], etc., are strong candidates and can be used successfully as a storage backend of GraphDB, because of their scalability and speed. Therefore, we decided to make the storage backend of GraphDB transparent to the rest of the system. In other words, the rest of the system does not know what storage medium is being used. In order to achieve this functionality, we designed an interface for GraphDB to talk to any storage backend. The interface is what one really needs to know to create a connector that will integrate GraphDB with a new backend. For example, if one wants to connect GraphDB to MySQL Database [11], then a StorageConnector should be implemented using the libraries of MySQL Database System and GraphDB should be configured to use that connector to access the storage backend. In Section 2.2.5, `StorageConnector` interface will be explained in detail.

Storage Layout

By partitioning the graph, GraphDB creates a storage layout for the graph. The storage layout determines the performance and efficiency of the system. In GraphDB we experimented with many algorithms for creating different storage layouts and measured their performance, which will be explained in detail in Section 3.3. The storage layout of a graph determines the performance of the system because, as mentioned earlier, significant amount of time is lost when loading data from disk or other storage medium, because storage mediums are often very slow compared to main memory. Today, a fast mechanical hard disk's raw data bandwidth is about 180MB/s [12] and a SATA Solid State Disk can read compressed data at around 500MB/s and minimum bandwidth of most common PC3-12800 DDR3 SDRAM is 12.8 GB/s [13], with DDR4 SDRAMs this bandwidth can increase to 17-25GB/s. As can be seen, even the fastest SSD is not even comparable to main memory. Therefore, we need a storage layout to increase the performance of GraphDB by decreasing the number of disk accesses made by the system. GraphDB makes disk accesses to read blocks of data, it deserializes the blocks in memory into subgraphs, and uses the subgraphs to serve

requests. Unless a node that is not in the subgraphs currently loaded in the subgraph buffer is requested, no further data is read from the disk. In the worst case, a storage layout that does not consider locality of access would make GraphDB go to disk too frequently. In GraphDB, we tried to decrease the number of disk accesses with locality-aware partitioning algorithms that would create locality-aware graph layouts on storage. Locality awareness is important in GraphDB because we assumed the algorithms we run on GraphDB will be traversal-based algorithms where nodes are traversed with respect to their relationships. In most of the cases, after a node is processed an adjacent node will be selected to be processed. Therefore, in our storage layout we needed to put adjacent nodes together into the same block, in order to reduce the need to load another block from disk.

Storage Medium

Essentially in our original design of GraphDB, the storage medium of GraphDB was the disk. However, we changed the design of GraphDB and it would work on other storage mediums such as network disks or distributed storage structures, databases, distributed databases, etc., as long as necessary connector interfaces are implemented for GraphDB. The terms disk and storage medium is used in this thesis interchangeably because our evaluation is done only on disk as the storage medium.

Storage Structure

As explained earlier, disk access is a very significant factor for the efficiency of GraphDB system, where it directly impacts the access time when saving or loading blocks of graph to/from disk. A key-value store is basically a non-relational database where data is held by addressing it with an identifier key. It is like a dictionary or a hash table where with the key you can address corresponding data in a very fast way. A key-value database can be a complex distributed system like Google's BigTable [14] or Amazon's Dynamo [15], or simpler standalone system

like BerkeleyDB [16]. In GraphDB we used a Java implementation [17] of LevelDB [18] which stores the graph blocks containing node and edge data, as well as graph metadata like the name of the graph, the number of nodes and various counters. LevelDB is a disk-based fast key-value database. It stores data sorted by its key as byte arrays addressed by a byte array key. LevelDB uses LSM-trees [19]. In GraphDB, data is held in LevelDB mainly as graph blocks that are addressed by their unique id. Non-relational databases doesn't have tables in the relational sense, but for the sake of understandability, we will call them tables. GraphDB maintains three tables on a LevelDB database. First table holds stateful data of graphs existing in the system, where the state is determined by its unique name, unique id, number of nodes, and node counter. This table is read by the system once in every execution and when a new graph is added, the system updates this table. The second table is for the nodes and their block(s) where they reside. This table contains the block id(s) identified by id of the node and graph id combined. For example, when system wants to load the block of node with id nid in graph with id gid, GraphDB concatenates gid and nid ($gid + nid$) and finds the id(s) of block(s) wherein that node resides. The third table is the one that holds the serialized block data addressed by block id. After finding the id(s) of block(s) in the first step it asks LevelDB to bring the block(s) with id(s) from the third table. LevelDB is designed as a fast database for random writes, but it is also fast on random reads.

2.2 System Architecture

Today's trend for deriving information from various digital sources has led to huge amounts of data being collected and processed. Graph databases are excellent choice for storing and processing the data since many times the data contains relationships and algorithms on graphs are mature enough to derive useful information from these relationships. However, when the data is big enough that it doesn't fit into memory, graph databases remain slow compared to other solutions. We designed a graph database framework called GraphDB, a graph traversal and processing system for big graphs which doesn't fit in memory. Our

framework basically holds the graph in partitioned blocks on a storage medium. When a graph is being processed, the relevant blocks are brought from the storage medium as partitions, converted to a subgraph, processed and written back to the storage medium as partitioned blocks again. The blocks that are being processed are held in a buffer. GraphDB consists of two main component groups, namely *base graph elements* and *architectural elements*. The base graph elements are *graph manager*, *graph*, *node*, *edge*, *property*. The data that is stored is attached to nodes and edges as property objects. Properties attached to a node or edge represent data attached to that particular element. The architectural elements of the system are *storage backend*, *storage connector*, *memory buffer*, *subgraph*, *block*, *buffer manager*, *block manager*, and *graph manager*. In this section, the graph structure of GraphDB will be explained first, then the components of the system that implement the graph structure we created will be explained.

2.2.1 Graph Manager

The GraphManager is one of the main components we use for managing the system and the graphs. GraphManager provides functionality related to graph manipulation and system management. On the system management side, it is responsible for maintaining in memory and on storage state of the graphs. It can initiate the process to bring a block from storage if needed. On the graph manipulation side, the graph manager manages the graph creation, and orchestrates the graph retrieval and update. It is implemented as a singleton class, so only one instance of it could be created and used. This design choice is preferred because there are elements that the graph manager instance needs to track and in order to prevent synchronization between different GraphManager instances we designed it as a singleton. In the programming order for the user the graph manager comes first and must be instantiated in the beginning in order to be able to start the system functions. Without instantiating the graph manager, user cannot access any graphs on the database or create graphs. That is why we designed the system to start with the instantiation of GraphManager for the first time. There are two other manager classes that are used by the system and

are instantiated when the system starts. These are the *BufferManager* and the *BlockManager* classes. They are purely related to the background (system related) jobs which is hidden from the user. These elements will be explained later in Section 2.2.7 and Section 2.2.9.

GraphManager instantiation is triggered with the call to its `getInstance()` method for the first time. At first the system tries to create the two other system elements `BufferManager` and `BlockManager` respectively. The details of these operations are too much interconnected so, it is explained here from a higher point of view. The GraphManager initiates the instantiation of BufferManager which searches for an existing database on the storage medium. In our initialization procedure, BufferManager is started when any graph manipulation operation of the GraphManager is called. If the system has never been started up before, system is lacking the necessary database files, it initiates the system properties for the first time. If a graph is added in that state, GraphManager would save the state. Next time, GraphDB is started, GraphManager reads that initiated state. Otherwise, initiates system initialization. If it can find it, it loads the last state of the system which are called *system properties*. The state of the system is defined with `numberOfGraphs`, `blockIdCounter` and the metadata of the latest states of the graphs.

In GraphDB, blocks are addressed by their unique ids. The state of a graph can be defined with `graphName`, `graphId` and `nodeIdCounter`. The retrieved data related to graphs is passed on to GraphManager. The GraphManager instantiates the graph objects (SuperGraph) with `graphName`, `graphId` and `nodeIdCounter` data for each graph. Hence, the graphs that were in the system before becomes available after GraphManager initiates in their previous state.

System shutdown is also initiated by GraphManager, the rest of the shutdown process is carried out in control of BufferManager. In the system shutdown, the opposite of system initialization is done. The system state is saved to the storage medium. The details of shutdown process can be found in the Section 2.2.7. The system needs to save the `blockIdCounter` while shutting down.

2.2.2 Graph

Graph element in GraphDB is used for handling graph operations within the system. It has two faces. The first is the system-level implementation and the second is the end user interface. We created an interface called Graph, which is visible to the end user. For managing graphs at the system level, an implementation of this graph interface, SuperGraph, is defined. The reason behind creating two interfaces is that we tried to give a basic graph abstraction to the end user which can be handled with a small set of commands, without having visibility into what is happening behind the scenes. In other words, we tried to hide the background processes from the end user and show the user only what she needs to see.

Graph creation in the system is one of the straight forward operations. A graph is created in control of the GraphManager. A graph may have a descriptive *graph name*, but it must have an integer *id*. The id of the graph is given by the GraphManager, which is stateful. When the system shuts down, its state is written to the storage medium for future usage. The graphs are held by their unique `id`, but for the convenience, users can locate graphs via their name.

When the graph is created, an object of type SuperGraph is instantiated. Since the SuperGraph class is an implementation of the Graph interface, when user needs a Graph handle, system returns a Graph type reference which only includes the user interface methods. Hence, the SuperGraph object will be able to used by the user (using Graph interface) with restricted operations as well as, by the system itself to achieve management goals of the system. The user interface is used for end user operations and allows higher level access for graph manipulation. On the other side, the system interface allows various atomic operations and low-level access to the graph operations. Therefore, the number of operations for low-level access is considerably larger than the high-level access operations.

When a graph is loaded from storage, the graph objects are instantiated again in their last state when the system was shut down. However a graph load operation from storage doesn't require that the nodes are loaded from the storage medium.

From the user point of view, the object with Graph type is just a handle to access graph manipulation operations. The operations of Graph interface may make the system load nodes from storage medium on demand. For example, calling the method for getting the identifier name of the graph doesn't require of any node to be brought from storage and consequently does not trigger any block loading operation from storage medium. However, adding a node to the graph may require some blocks to load from storage medium.

From the system point of view, the SuperGraph class is not just a handle. It implements the Graph interface so that, it can be used as a handle to graph interface in the user view. In the user view, the user cannot access the operations of SuperGraph class. It can access only the Graph interface operations. The main data held in the SuperGraph is the *node id counter*, *node count*, *subgraph count*. These are metadata associated with the graph structure we use. The state of the graph that has to be preserved is in the *node id counter*. The function of the node id counter field is that the graph gives new nodes identifier long integers consecutively, so when we save the graph on storage medium, we need to be able to revive the graph with the exact state that it had been before. Saving state of a graph is more involved, so there are a few more items we use to preserve the state of a graph, which will be mentioned in the future sections.

Tracking subgraphs is another job that SuperGraph is responsible for. However, SuperGraph is only responsible for Subgraphs in the memory. Subgraphs have ids, but the id of a subgraph is not unique and is not an identifier for the subgraph since it is only a image of a block or blocks in the memory. Therefore, subgraphs are identified by the id(s) of the blocks that they are associated with. We don't need a counter for subgraphs. When it is written on the storage medium, the subgraph id loses it significance and when the same partition is loaded from storage, the subgraph id may be different than before. It is just for tracking partitions in memory. The node counter is kept just for convenience, in case the user wants to get the total number of nodes in that graph. It is a relatively small state field that is preserved.

Nodes in GraphDB do not belong to SuperGraph element but the subgraph. A

SuperGraph consists of subgraphs and they consist of nodes and edges. However, SuperGraph keeps track of the nodes that are loaded into the memory (as part of the subgraphs). Determining a node is in memory or on the storage medium is done with the help of this information.

2.2.3 Node

Node class is one of the classes that is being used just as a handle to provide a functional interface to the end user. The only data that a Node instance holds is the node id in its graph, a reference to its graph and its dummy state. Thus, the operations that the node methods offer are internal operations requiring deeper access to the system internals, so the reference to its graph is held as a SuperGraph object. The dummy state of a Node object tells us whether the subgraph of that node is in memory or not. It becomes *true* when the subgraph of a node is not in the memory and the subgraph of another node that has an edge with the other node is in memory. The subgraph retrieval from storage is triggered from almost all actions that node object takes. Triggering subgraph retrieval is ignored by the system if the subgraph is already loaded to the memory otherwise the retrieval procedure is started. For example, node A and node B has an edge and they reside in different subgraphs. Consider a situation where the subgraph of B is in memory but the subgraph of A is not, so the node for A is created but marked as dummy, which means the node is not in the memory. If user tries to call a method of a dummy node, A, the subgraph of A would be loaded from storage. This situation happens in a couple of cases. For example, the subgraph of a node is being loaded, and some of the nodes in the subgraph have cross edges. The subgraphs of the incident nodes of those cross edges are not loaded so dummy nodes are created for those incident nodes. Another case would be when a subgraph is being written to storage (see invalidation in Section 2.2.6) for any reason. Then the nodes of the invalidated subgraphs, which has cross edges with incident nodes that are in memory at that moment, are deleted from node index in SuperGraph but marked as dummy. As such, when they are needed to take action, their subgraphs can be reloaded to the memory from storage. In

GraphDB a node can have its properties (data) identified by Property object, which is held by the database.

2.2.4 Edge

An edge is represented with two Nodes in GraphDB. It is a wrapper class for an edge in the system at the structure level. Edge class functionality is provided in parallel with the rest of the wrapper classes. It has two references to the identifiers of two nodes it connects, source and destination, which are `long integers`, and a reference to its SuperGraph object. The reference to its graph object is explained in Section 2.2.3 as part of Node class.

Edge creation is only done through the provided end user interface, with Node objects. An edge may have node references where one or both of the referenced nodes are not loaded to the memory (written to the storage medium) at a moment in execution, but the edge is still valid and open to take action by the user because the referenced nodes are marked as `dummy` by the system when they are written to their blocks on storage. Thus, whenever an action has to be taken with that edge, if the action requires a block to be fetched from storage, it is fetched. Edges are held in subgraph objects in an adjacency list format, which can be accessed via the adjacent Nodes of the Edge. Since the system can hold directed graphs, the subgraph has two adjacency lists having that Edge, one is incoming, and the other one is outgoing. If the Nodes of an edge resides in two different subgraphs/blocks, only incoming or outgoing part of the adjacency list is kept on each subgraph. However, when the subgraph is serialized into a block the node identifiers are written to both nodes' blocks. In this case, when the block is retrieved from storage, the edge is created with two references to its nodes, but one of the Node references are marked as dummy if that node is not already loaded to the system memory. With this type of behavior, we achieve lazy loading when an edge with connections to two different blocks is retrieved. If the edge connections are to the nodes in the same block, then both of the nodes' adjacency lists will contain the edge when serialized.

Like nodes, edges might also have properties attached to them, which enables the system to hold data for the each edge and use that data for processing. Therefore, the edges can hold data. Edge properties are held in the same fashion as the Node properties. A property is described with a key, which is a *String* instance, and the data, which can be of arbitrary type. It can be saved, retrieved or deleted with that key. The details of the Property class and management of properties is described in Section 2.2.4.

Property

In our graph definition, we declared that nodes and edges might have data attached to them. Since, the system is a database, we must provide a way for user to store the data related to the graph elements. Accordingly, we created the Property class. It provides APIs for modification, removal, and addition of properties where data is accessed via a unique `String` key. A property object can hold a number of atomic data. Property attached to nodes and edges is basically a hash table, which holds data with a `String` key. Attached data can be of any type, but it must be serializable. Otherwise, GraphDB gives an error, indicating given property is invalid. Therefore user-defined classes must implement the `Serializable` interface of Java when user is attaching custom class property values to nodes and edges. An improper implementation of *Serializable* interface would result more bytes to be serialized than needed. The user must be careful using user-defined classes with the Property class. In each serialization procedure of a subgraph, the Property objects attached to nodes and edges are serialized with elements in them and put into the same block. Edges that have two nodes in different blocks repeat the property data in each block, trying to make less disk access. If we hold only one copy of the data in two blocks, we always need to load the block that contains the data.

2.2.5 Storage Connector

Our system is designed in a way that the storage medium can be customized for different storage types and needs. This separation eases interfacing GraphDB with other systems that can be used as a storage medium. For example, you can use the file system, HDFS or MySQL, etc. as a storage medium on GraphDB. In order to gain this ability, first one needs to implement the `StorageConnnector` interface, providing intended system methods. Then one can configure GraphDB to use your custom connector. For our experiments, we implemented storage interfaces for LevelDB.

In order to separate the connector implementation further from the GraphDB system, the provided and requested data are in byte array form. This enables the implementation not to include or require one to use any library/class from the GraphDB system. The implementation only needs to include its storage system specific libraries, if there are any. For example, if you want to use the file system as the storage, you need to implement the `StorageConnector` interface using Java file input/output functions.

In our storage implementation, the data is kept in table-like structures. We will call them tables for convenience. In the storage structure, there are three tables, the *graph table*, the *node table* and the *block table*. Graph table holds the graph id to graph properties mapping, since this is a graph database it may hold more than one graph and each graph has two properties for saving their state, *node id count*, and *node count*. The graph table also holds the main database properties, which includes *block id count*.

The node table holds the block id list that the node resides on. The row mapping of the table is *node id* $\rightarrow$ *block id*. The node id is not a unique value throughout the database, so we added the `graph id` to the mapping and it became *graph id + node id* $\rightarrow$ *block id list* in the node table. When a node is requested by the system, the block id(s) are looked up from the node table and brought into memory when needed.

In the GraphDB system, a node is held with its edges and its data, so if the node has too many edges a block may not present enough storage are for that node, and thus it is split into multiple blocks. Therefore, we are mapping node id to a list of block ids.

The third table GraphDB uses is the block table. It holds a *block id* $\rightarrow$ *block* mapping, where block ids are unique in the system. In terms of size, this is the table that holds the most graph data, as all node data, edge data, and their attached property data is held in this table.

2.2.6 Subgraph

A subgraph in GraphDB represents a partition of a graph. As many partitions makes up a full graph, one or many subgraphs creates a graph. Our graph structure consists of a graph and inside many subgraphs. A subgraph in our system holds all data about the partition. Nodes, edges, their properties, are all held in the Subgraph element. The graph element is only responsible from keeping track of the nodes (global node index) that are in memory and the subgraphs. All other graph operations related to nodes and edges of the graph are redirected from Graph to the corresponding Subgraph. Finding the corresponding Subgraph is the responsibility of Graph.

As we said before, a subgraph represents a partitions of a graph but, a block on the storage cannot be directly mapped to a partition. Subgraphs are images of blocks when they are loaded to the memory.

Subgraphs that belongs to a graph are held in the same Graph object but there is a limit that how many subgraphs, the system may hold according to the running machines memory configuration. However it is not infinite, so we brought a limit to the number of subgraphs that the system may hold in its memory which is determined by the size of the block that a subgraph represents.

Subgraph Types

We have two different subgraph types, *partitioned* and *non-partitioned*. The purpose of creating two kinds of subgraphs is that we designed the storage structure to contain blocks with certain capacities, because we wanted the capacity of a block to be tunable, since the storage backend is flexible and it might be crucial from performance point of view that the block size is tunable for different storage backends.

Non-Partitioned

Non-partitioned subgraphs are basic subgraphs that hold nodes, edges, and their data. These subgraphs take only one block-size in the buffer.

Partitioned

Partitioned subgraphs are for the exceptional subgraphs, where the capacity of a block is not enough to hold a node, its edges, and their data. In this case we create a special Subgraph object that contains only one node, its edges and data attached to them. This type of subgraphs are serialized into blocks in a different way than non-partitioned subgraphs. Normally, subgraphs are serialized into byte array and the byte array is written to storage with the given block id and necessary updates/inserts are done with the block id in the database tables. However, Partitioned subgraphs hold nodes that do not fit into a block with the specified capacity. In this case, we serialize the subgraph into a byte array as if it is not a partitioned subgraph, but at the final step we run a split procedure that splits up the byte array into pieces that would fit in a block. These subgraphs hold more than one block-size in the buffer. Since writing this type of subgraph would trigger writing multiple blocks into storage. The only difference when updating the database tables is that we update node table with a list of block ids instead of a single block id. The details of serialization part will be explained in

Section 2.2.6.

Block

A block represents the serialized version of a subgraph or part of a subgraph in the GraphDB. The block resides on storage mainly, and it is loaded into memory to be deserialized and when a subgraph will be written to storage, it is serialized into a block or blocks. The capacity of a block is determined by the system. In our system, we set the block size to $10KB$ in our experiments. A block has a unique id and it is universally unique, a block of any graph in the system cannot have the same id. The block id is a long integer. The track of blocks are handled by GraphDB by saving the state of block id generating system, which is done by saving the last given block id to system properties. Next time the system boots up, GraphDB knows which block id to generate next.

Blocks are held in storage by their ids in the block table. The block ids are also listed in the node table to indicate the node's block(s). At any time in the system, the number of blocks loaded from storage is known and and is limited by the buffer size. The system holds the raw blocks in the buffer, but it is actually not a buffer but a tracking mechanism. It only tries to limit the number of blocks that can be loaded from storage and when it is full, it forces the system to write a block or blocks to storage. When a block write operation is initiated, system checks if the block belongs to a partitioned subgraph, if it is it will be writing all other blocks to the storage too, along with the meta data of blocks.

Serialization

Serialization phase of a subgraph is takes place when the system buffer is full and some of the buffer space should be released by writing a subgraph into the storage medium. As explained in Section 2.2.6, there are two types of subgraphs, and they are serialized differently. The non-partitioned type of subgraph would correspond to one block of bytes. The process starts with the serialization request of a

subgraph from the buffer. Each and every element of that subgraph is serialized into bytes. In this process, GraphDB uses descriptors to separate and identify the data that is being serialized. The nodes are represented with long integers, edges are represented with two long integers. The related data of nodes and edges are also serialized. That's why the serialization interface implementation is a must for the data that is being attached to a node or edge. GraphDB serializes their related data after the descriptor and identifier of a graph element is serialized.

In case of serialization of a partitioned subgraph, the serialization mechanism is similar. All elements of the subgraph are iterated and serialized with the help of descriptors. One difference is that, a partitioned subgraph corresponds to not one but multiple blocks of data. The partitioned subgraph is converted to a big block of graph data which exceeds the number of bytes that limits the block size, and then it is divided into maximum block size of blocks. Afterwards, the generated blocks are sent to be written to storage medium and subgraph is deleted from the buffer. The number of disk blocks that a subgraph would need to be serialized is known and calculated at each update in the subgraph. Therefore, we can ensure that the memory holds exactly memory buffer size of bytes.

Deserialization

Deserialization process is the direct opposite of the serialization process, where the blocks are read from storage into memory buffer, and with the help of descriptors and element identifiers. GraphDB reconstructs the subgraph as it was before serialization. For partitioned subgraphs, merging of the multiple blocks is performed on the fly while subgraph elements are being reconstructed.

2.2.7 Buffer Manager

Buffer manager is responsible for managing the blocks in the buffer. There can be multiple blocks in the buffer and the size of the buffer can be adjusted according to the memory amount configured by the user for the buffer. Buffer manager

uses StorageConnector interface methods for reading, writing and querying. A custom StorageConnector implementation should be provided initially through configuration utilities.

When the buffer is full, the blocks are written to the storage in the least recently used (LRU) fashion. If the graph can fit in the buffer memory, it would be served from the memory until the program completes. When the program completes, the buffer is flushed, all blocks in the buffer are written to the disk. The buffer flush function can be manually called by the user at any time. When the graph does not fit into the memory, the blocks are brought from storage to the memory as needed and expired blocks are written to the storage when the buffer is full.

2.2.8 Memory Buffer

In GraphDB, memory buffer is composed of blocks, where each block corresponds to a block/partition on disk. However, when a block is brought from disk to memory, it is not kept as it was on the disk. It goes over the deserialization procedure and is deserialized into subgraphs. Vice verse, this process has serialization counterpart where the graph is serialized into disk blocks. Due to the serialization/deserialization process, the space that blocks hold on disk and subgraphs hold on memory will be different.

2.2.9 Block Manager

Block manager keeps track of the blocks that have been created. Each block has an id which is used to describe that block in the database. A block is defined as a byte stream, it may have nodes, edges of nodes, node data and edge data. The data is written on a block using integer identifiers followed by the identified data.

2.3 Storage Layout

When a graph us started to be inserted into GraphDB, as graphs dynamically changes, the layout on disk will change as well. Storage layout is basically the result of partitioning model that GraphDB uses. Storage layout is the key element which directly effects the efficiency of the system.

2.3.1 Layout Policies

In GraphDB partitioning models are called layout policies. They are divided into two main categories, *Select Policies* and *Split Policies*. Each category looks at the graph from different aspects and effect the partitioning from a different perspective, which gives us the ability to control partitioning in detail.

2.3.2 Select Policy

Select policy determines where a node with no other connection in the graph goes to. In our streaming model, when a node is added to the system it would have no connections in the graph to other nodes, it can be the first node in the graph or the last node. After that node is added, the connections will be made and that node may be moved to another subgraph or not, which is determined by the *Split Policy* in GraphDB. There are two select policies we used in our system.

Random Select Policy

In this select policy, when a new node insertion is initiated, it is placed in a random subgraph at the time of insertion. If there are no subgraphs in the memory at that time, a new empty one will be created and the node will be placed into the new subgraph. For load balancing purposes, randomization in this case is useful.

Algorithm 1: executeSplitPolicy(G, S, P)

```
Param : G, graph
Param : S, subgraph
Param : P, split policy
if size(S) >= MaxBlockSize then    ▷ If subgraph size exceeded max block size
    if isNotPartitioned(S) then                    ▷ Split if S is not partitioned
        S1, S2 ← split(P,S,G)      ▷ Execute the split policy for the subgraph
```

First Available Select Policy

In this policy, when a new node insertion is initiated, the currently loaded subgraphs will be processed and the node will be inserted into the first subgraph that has available space in it. When insertion is initiated, if there are no subgraphs loaded in the memory, First Available Select Policy will create a new subgraph and insert the node into that subgraph. This method provides superior subgraph/block density compared to Random Select Policy. It creates denser subgraphs/blocks compared to Random Select Policy.

2.3.3 Split Policy

Split policies are the main algorithms for creating the disk layouts. In our streaming model, subgraphs grow larger and larger as graph elements are populated, and at some point these subgraphs should be broken into parts to ensure the block size we specified in the configuration of GraphDB. There are four split policies we use in GraphDB. These are: *Affinity Based Subgraph Split Policy*, *BFS Subgraph Split Policy*, *Half Subgraph Split Policy*, and *Random Subgraph Split Policy*. Split policies are executed when a graph element is being inserted to a subgraph that reaches maximum block size after insertion. After each insertion, GraphDB checks impacted subgraphs and decides if a split is needed or not. Algorithm 1 gives the pseudo code of the split policy execution. The `moveNode()` function is used in all split policiesl which simply moves the node and all its incident edges and properties from one subgraph to another subgraph.

Algorithm 2: AFFINITYSPLITPOLICY(S, G)

Param : S, subgraph
Param : G, graph
HALFSPLITPOLICY(S, G) ▷ call directly Half Subgraph Split Policy

Affinity Based Subgraph Split Policy

Affinity Based Subgraph Split Policy uses different rules for moving nodes compared to other split algorithms. This policy basically measures a node's affinity and decides whether it should stay where it is or moved to another subgraph. Unlike other policies, this policy does not only move nodes to other subgraphs when *executeSplitPolicy* presented at Algorithm 1 is called. At split time, it uses Half Subgraph Split Policy to partition the subgraph to ensure that the locality is not affected negatively (see Algorithm 2). It moves nodes when a new edge is added to the graph using Algorithm 3. Add edge action uses the Equation 2.1 to calculate the affinity of each subgraph in the memory at that time and decides to move the node to one of the subgraphs or not. In the equation, T represents the number of total edges of a node, A represents the number of cross edges to a subgraph, f a tuning parameter.

$$1 - \exp^{\frac{T-A}{f}} \tag{2.1}$$

BFS Subgraph Split Policy

BFS Subgraph Split Policy is a graph partitioning policy where half of the nodes in one subgraph is moved to a new subgraph (or subgraphs) according to the BFS execution order. The policy stops moving nodes when 50% of the nodes have moved from the splitted subgraph.

Algorithm 3: affinityAddEdgeAction(E, S, G)

```
Param : E, edge
Param : S, subgraph
n_src ← getSource(E)                          ▷ get the source node of the edge
S_src ← getSubgraph(n_src)                    ▷ get subgraph of source
for for each s_i ∈ {S}_memory do              ▷ for each subgraph in memory
    M_i ← calcAffinity(n_src, S_src, s_i)     ▷ calculate affinity for s_i
if getMaxAffinitySubgraph(M) ≠ S_src) then
    moveNode(n_src, maxAffinitySubgraph(M))   ▷ move node to the max
    affinity subgraph
n_dest ← getDestination(E)                    ▷ get the destination node of the edge
S_dest ← getSubgraph(n_dest)                  ▷ get subgraph of destination
for for each s_i ∈ {S}_memory do              ▷ for each subgraph in memory
    M_i ← calcAffinity(n_dest, S_dest, s_i)   ▷ calculate affinity for s_i
if getMaxAffinitySubgraph(M) ≠ S_dest) then
    moveNode(n_dest, maxAffinitySubgraph(M))  ▷ move node to the max
    affinity subgraph
```

Half Subgraph Split Policy

Half Subgraph Split Policy partitions a subgraph into two (or more) subgraphs with equal number of nodes. The order of assignment of nodes to subgraph is done according to the id of the node.

Random Subgraph Split Policy

Random Subgraph Split Policy partitions a subgraph into one or more pieces by distributing nodes to subgraphs in random order without regarding the locality of other measures. Algorithm 6 shows the Random Subgraph Split Algorithm internals.

Algorithm 4: BFSSPLITPOLICY(S, G)

```
Param : S, subgraph
Param : G, graph
S_new ← CREATENEWSUBGRAPH(G)                          ▷ create a new subgraph
n_root ← SELECTRANDOMNODE(S)                          ▷ start from a random node
Q ← CREATEQUEUE()                                     ▷ create an empty queue
numMoved ← 0                                          ▷ initialize number of nodes moved
ADDTOQUEUE(n_root, Q) while                           ▷ !isEmpty(Q) && numMoved ¡
subgraph.numNodes()/2 do
    n_temp ← PEEK(Q)                                  ▷ peek the first element of queue
    for for each e ∈ getEdges(n_temp) do              ▷ for each edge of n_temp
        n_adj ← GETADJACENT(e, n_temp)                ▷ get an adjacent node of n_temp
        if !isVisited(n_adj) then
            ADDVISITED(n_adj) ADDTOQUEUE(n_adj, Q)
    MOVENODE(n_temp, S_new)                           ▷ move node to new subgraph
    POLL(Q) if isPartitioned(S_new) then
        S_new ← CREATENEWSUBGRAPH(G)                  ▷ create a new subgraph
```

Algorithm 5: HALFSPLITPOLICY(S, G)

```
Param : S, subgraph
Param : G, graph
num_S ← NUMNODES(S)                                   ▷ get number of nodes in S
S_new ← CREATENEWSUBGRAPH(G)                          ▷ create a new subgraph
for for i → num_nodes(S)/2 do                         ▷ for half of the nodes in S
    MOVENODE(n_i, S_new)                              ▷ move node to new subgraph
    if isPartitioned(S_new) then
        S_new ← CREATENEWSUBGRAPH(G)                  ▷ create a new subgraph
```

Algorithm 6: RANDOMSPLITPOLICY(S, G)

```
Param : S, subgraph
Param : G, graph
S_new ← CREATENEWSUBGRAPH(G)                          ▷ create a new subgraph
for each node n ∈ S do                                ▷ for each node in subgraph
    if GENERATERANDOMBOOLEAN() then
        MOVENODE(n_i, S_new)                          ▷ move node to new subgraph
        if isPartitioned(S_new) then
            S_new ← CREATENEWSUBGRAPH(G)              ▷ create a new subgraph
```

Chapter 3

Experimental Evaluation

3.1 Experimental Setup

Hardware platform. We conducted our experiments on a server equipped with two Intel Xeon E5-2620 2.00 Ghz CPUs, 32 GB DDR3 RAM in NUMA scheme, and on the disk side, five 300 GB 15K SAS disks in RAID 5 configuration.

We experimented with six graphs that are generated with Erdos-Renyi generator of Boost Graph Library [20], in three different sets. These sets have 10K, 100K, 1M nodes and 5 and 10 edge degrees. In the experiments, these graphs are inserted into GraphDB in a streaming manner.

For the insertion of the graphs, we experimented with different combinations of policy algorithms that will create unique disk layouts. The select policies that are used in the experiments were explained in Section 2.3.2, and split policies in Section 2.3.3.

In order to be able to see the performance effect of different disk layouts we ran some traversal based queries on graphs that are created with our layout policies.

3.2 Experimental Scenarios

In the first part of the experiments, combinations from split and select policies, which are listed in the Table 3.1, are used to create a different graph layout on disk with each combination, as the graph is inserted into the system. Even though the graph elements are inserted every time in the same order, and the Affinity Split, BFS Split, and Half Split policies deterministically partition the subgraphs; since the selection algorithms have randomness, it results in variation in the generated layouts, even with the same settings and the same workload.

Layout No	**Split Policy**	**Select Policy**
1	Affinity Split	Random Select
2	BFS Split	First Available Select
3	BFS Split	Random Select
4	Half Split	First Available Select
5	Half Split	Random Select
6	Random Split	First Available Select
7	Random Split	Random Select

Table 3.1: Generated disk layouts with select and split policies.

When experimenting with First Available Select Policy, we only experimented with three of our split algorithms, because our Affinity Split Algorithm uses Random Selection Algorithm by default.

In the second part of the experiments, we ran traversal based queries using algorithms listed below, on the graphs that were inserted into GraphDB as part of the first part of our experiments.

- PageRank
- Clustering Coefficient
- BFS (Breadth First Search) Walk
- Random Walk

Edge Degree	BFS Split Pol.	Half Split Pol.	Avg.
5	-9.26%	-5.10%	-7.18%
10	-8.99%	-5.09%	-7.04%
Avg.	-9.13%	-5.10%	-7.11%

Table 3.2: Random Split Policy average running time differences compared to other policies on insertion with First Available Select Policy.

Edge Deg.	Affinity Sp. Pol.	BFS Sp. Pol.	Half Spl. Pol.	Avg.
5	-13.81%	-4.36%	3.73%	-4.82
10	-24.53%	-7.49%	-2.53%	-11.52
Avg.	-19.17%	-5.93%	-0.60%	-8.17%

Table 3.3: Random Split Policy average running time differences compared to other policies on insertion with Random Select Policy.

3.3 Results

3.3.1 Insertion Results

In the insertion results, in Figure 3.1, we can see for graphs with edge degree 5 (Figures 3.1a and 3.1c) and with edge degree 10 (Figures 3.1a and 3.1c) the fastest algorithm was Random Split Policy. Especially for graphs with edge degree 10, the average running time difference between Random Split Policy and other algorithm is 11% for the Random Select Policy and 7% for the First Available Select Policy. Tables 3.3 and 3.2 show the detailed run time difference of Random Split Policy compared to other policies.

The edge degrees have a big impact on the performance. The denser graphs would have more edges per node, so as we add edges to the graph, more blocks are required to be brought from the storage medium. That is why the difference between algorithms would increase as the graphs get denser.

First Available Select Policy has slower graph insertion times compared to Random Select Policy as we can see in Table 3.4, even though there are a few cases in which the former performs better. However, in general Random Select Policy is faster. The key difference between those algorithms, although both algorithms

Edge Deg.	Nodes	BFS Sp.	Half Sp.	Random Sp.	Avg.
5	10K	35.61%	46.36%	42.27%	41.42%
10	100K	56.00%	59.44%	57.25%	57.56%
5	100K	-0.84%	-0.24%	-19.82%	-6.97%
10	100K	8.43%	7.68%	5.83%	7.31%
5	1000K	5.89%	-0.57%	-1.29%	1.34%
10	1000K	2.99%	-1.83%	-1.25%	-0.03%
Avg.		18.01%	18.47%	13.83%	16.77%

Table 3.4: Random Select Policy running time compared to the First Available Select Policy on average.

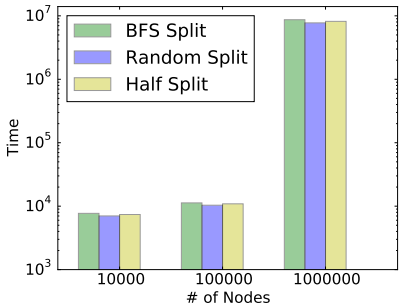
have $\mathcal{O}(n)$ run time in the worst case scenario (where randomly selected subgraph is not accepting nodes because it is full or for other reasons), First Available Select Policy checks the subgraphs in the order of creation and Random Select Policy checks subgraphs randomly. First Available Select Policy creates more denser graphs since it tries to fill up the capacities of subgraphs in some order, which means when edges are inserted to the system number of split operations will increase, since the subgraphs are already filled up. However, this might not always be the case for the Random Select Policy.

3.3.2 Query Results

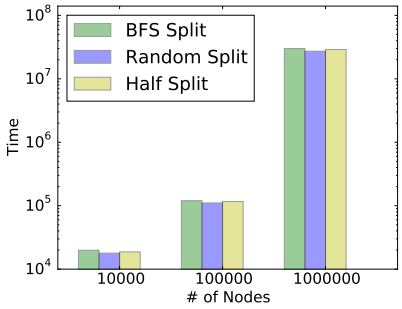
PageRank Algorithm

Figures 3.2, 3.3, 3.4, and 3.5 show us comparative results of the Pagerank [21] algorithm that was run on graphs with edge degrees 5 and 10, inserted with First Available Select Policy, Random Select Policy, Affinity Split Policy, BFS Split Policy, Half Split Policy, and Random Split Policy. At first, we can clearly see that using Random Split Policy draws a worst case baseline for us. It performs almost always the worst for both of the select policies. For the PageRank algorithm, Random Split Policy took 11% more time on 5 degree graphs and 29% more time on average (Table 3.5) compared to the best alternative.

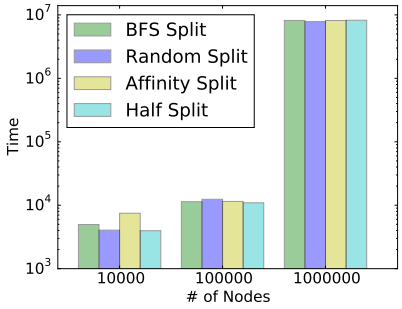
The best performing Split Policy for Random Select Policy was Affinity Split



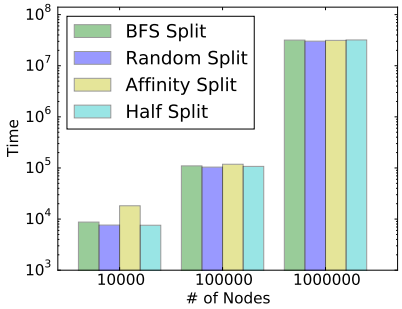
(a) Insertion with First Available Select Policy for graphs with edge degree of 5.



(b) Insertion with First Available Select Policy for graphs with edge degree of 10.



(c) Insertion with Random Select Policy for graphs with edge degree of 5.



(d) Insertion with Random Select Policy for graphs with edge degree of 10.

Figure 3.1: Insertion Result Graphs

Edge Deg.	Affinity Sp. Pol.	BFS Sp. Pol.	Half Sp. Pol.	Avg.
5	10.80 %	12.98 %	10.18 %	11.32 %
10	37.80 %	32.77 %	18.39 %	29.65 %
Avg.	24.30 %	22.87 %	14.28 %	20.48%

Table 3.5: Split Policy comparison for PageRank algorithm for Random Select Policy against Random Split Policy.

Edge Deg.	BFS Sp. Pol.	Half Sp. Pol.	Avg.
5	16.51 %	12.53 %	14.52 %
10	38.75 %	22.45 %	30.60 %
Avg.	27.63 %	17.49 %	22.56 %

Table 3.6: Split Policy comparison for PageRank algorithm for First Available Select Policy against Random Split Policy.

Policy for the edge degree of 5 and BFS Split Policy for the edge degree of 10.

In the case of First Available Select Policy, split policies took 14% and 30% less time compared to Random Split Policy, respectively for edge degree 5 and edge degree 10 (Table 3.6).

The PageRank algorithm performed best with the BFS Split Algorithm on the graphs inserted with First Available Select Policy.

In terms of select policy PageRank algorithm ran 3.24% faster on graphs inserted with First Available Select Policy compared to Random Select Policy.

Connected Components Algorithm

In our experiments CC algorithm took considerably more time to compute than PageRank algorithm. The results shows us the concept from another angle. When we compare the split algorithms to the Random Split Algorithm, for the Random Select Policy, the edge degree 5 graphs took 7% and edge degree 10 graphs took 20% less time to compute CC. These numbers were higher when we were doing the same comparison for PageRank.

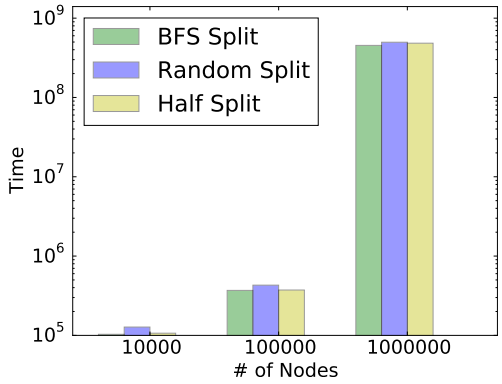


Figure 3.2: PageRank Query on First Available Select Policy created layout with edge degree of 5.

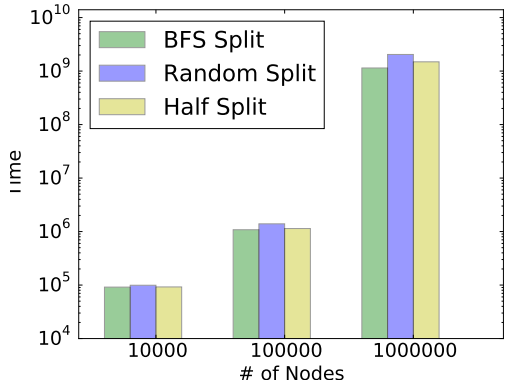


Figure 3.3: PageRank Query on First Available Select Policy created layout with edge degree of 10.

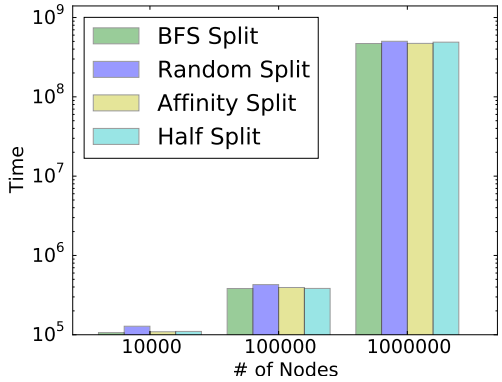


Figure 3.4: PageRank Query on Random Select Policy created layout with edge degree of 5.

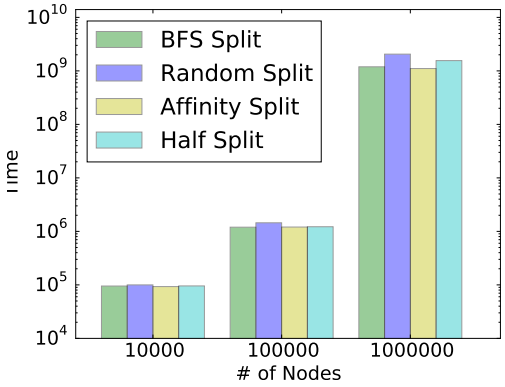


Figure 3.5: PageRank Query on Random Select Policy created layout with edge degree of 10.

Edge Deg.	Affinity Sp. Pol.	BFS Sp. Pol.	Half Sp. Pol.	Avg.
5	9.79 %	6.95 %	5.07 %	7.27 %
10	30.71 %	20.91 %	8.98 %	20.20 %
Avg.	20.25 %	13.93 %	7.02 %	13.73 %

Table 3.7: Split Policy comparison for Connected Components Algorithm, Random Select Policy against Random Split Policy.

The best performing split algorithm for Random Select Policy was Affinity Split Algorithm. Which can be seen in the split algorithm comparison table, that is Table 3.7.

The First Available Select Policy again performed slightly better than Random Select Policy. In particular, the difference is 8.7% on average when we comparing the shared split policies (BFS, Half, and Random Split Policy).

When we compare the split algorithms for the First Available Select Policy, BFS Split Algorithm was the best performing algorithm (Table 3.8) as for PageRank.

Edge Deg.	BFS Sp. Pol.	Half Sp. Pol.	Avg.
5	9.62 %	6.33 %	7.97 %
10	12.87 %	4.66 %	8.76 %
Avg.	11.24 %	5.49 %	8.37 %

Table 3.8: Split Policy comparison for Connected Components Algorithm for First Available Select Policy against Random Split Policy

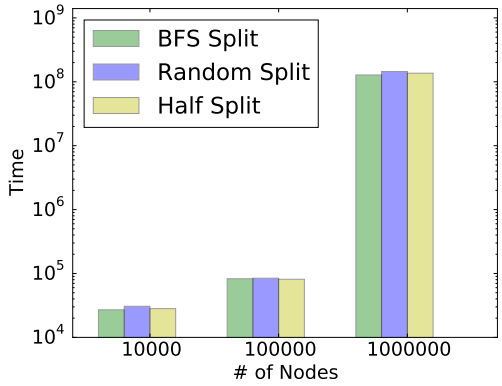


Figure 3.6: Connected Components query on First Available Select Policy created layout with edge degree of 5.

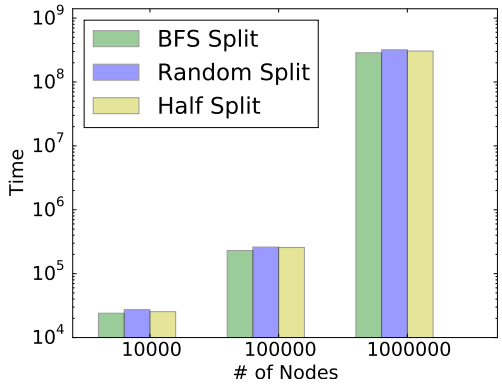


Figure 3.7: Connected Components query on First Available Select Policy created layout with edge degree of 10.

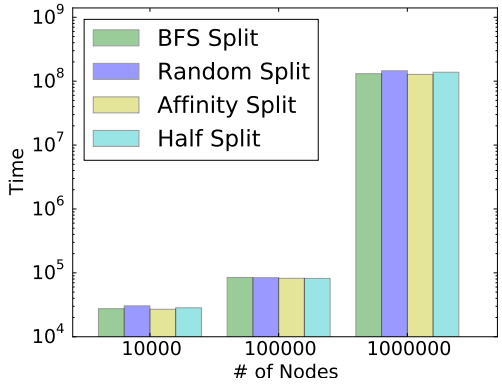


Figure 3.8: Connected Components query on Random Select Policy created layout with edge degree of 5.

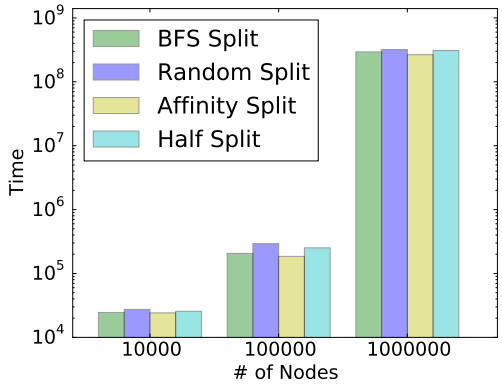


Figure 3.9: Connected Components Query on Random Select Policy Created Layout with Edge Degree of 10.

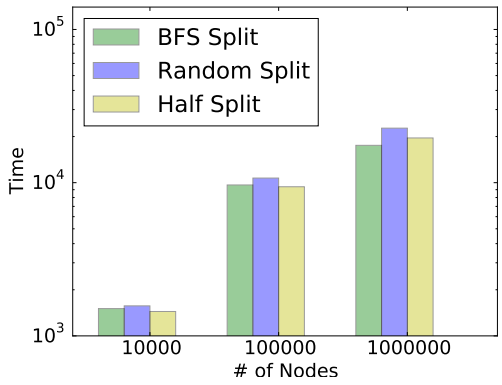


Figure 3.10: BFS Walk Query on First Available Select Policy Created Layout with Edge Degree of 5.

BFS Algorithm

For these experiments, BFS algorithm is implemented to search for a node in the graph. It starts from a node n_{start} in the graph and tries to find n_{target}. For the sake of the experiments we limited the maximum hop count to 30. In the experiments we selected as n_{start} the node with id 0 and as n_{target} an appropriate node in the graph.

When we compare the split algorithms to Random Split Policy with the BFS algorithm on the Random Selection Policy graphs, the average run time difference is 14.6% for the edge degree 5 and 22.7% for the edge degree 10 (Table 3.9). This

Edge Deg.	Affinity Sp. Pol.	BFS Sp. Pol.	Half Sp. Pol.	Avg.
5	17.09 %	14.36 %	12.39 %	14.61 %
10	32.56 %	22.68 %	13.12 %	22.79 %
Avg.	24.83 %	18.52 %	12.75 %	18.70 %

Table 3.9: Split Policy comparison for BFS algorithm for Random Select Policy against the Random Split Policy.

Edge Deg.	BFS Sp. Pol.	Half Sp. Pol.	Avg.
5	14.87 %	12.89 %	13.88 %
10	21.77 %	17.16 %	19.47 %
Avg.	18.32 %	15.02 %	16.67 %

Table 3.10: Split Policy comparison for BFS algorithm for First Available Select Policy against Random Split Policy.

difference decreases to 23.8% and 19.4% respectively for the First Available Select Policy (Table 3.10).

Comparing First Available Select Policy and Random Select Policy, the difference on average is 3.7% between comparable split algorithms in favor of First Available Select Policy. The best performing graph layout is Half Split Policy and First Available Select Policy with 5.3% better results compared to Random Select Policy.

In the BFS algorithm, we can see in both Table 3.9 and Table 3.10, when we increase the edge degree of the graphs the run time percentage increases drastically (in comparison to worst case baseline). As the graph get denser, the locality aware algorithms provides increased run time efficiency.

Random Walk Algorithm

Random Walk Algorithm is implemented in a similar fashion to the BFS algorithm. We start a random walk from n_{start} and take a fixed number of steps. At each step, the algorithm selects a random adjacent node. For the sake of the experiments, to get the correct results, our random number generator is started with a seed, to ensure that as we repeat the test, we go over exactly the same

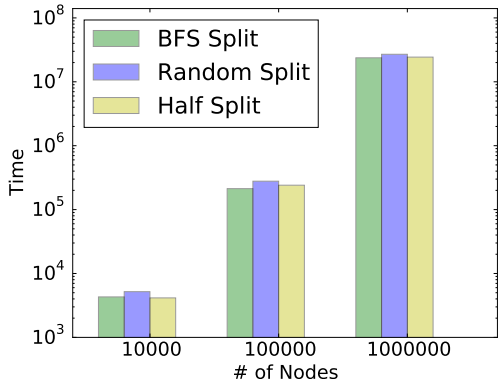


Figure 3.11: BFS walk query on First Available Select Policy created layout with edge degree of 10.

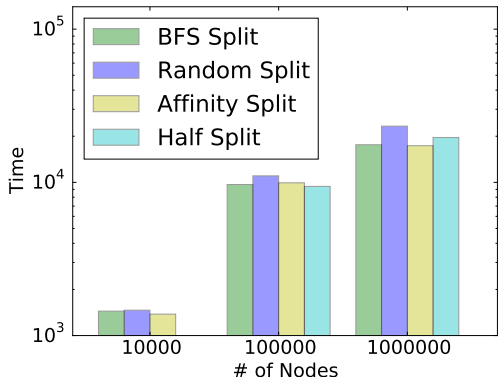


Figure 3.12: BFS walk query on Random Select Policy created layout with edge degree of 5.

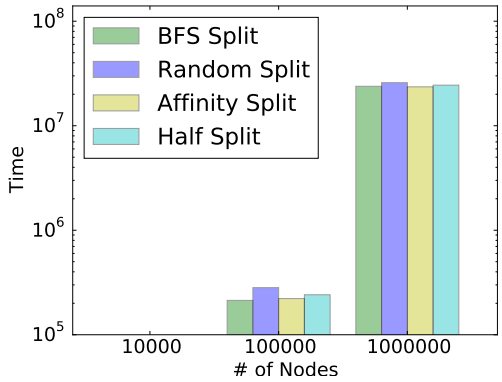


Figure 3.13: BFS walk query on Random Select Policy created layout with edge degree of 10.

Edge Deg.	Affinity Sp. Pol.	BFS Sp. Pol.	Half Sp. Pol.	Avg.
5	4.06 %	9.29 %	4.19 %	5.85 %
10	26.73 %	26.83 %	10.47 %	21.34 %
Avg.	15.39 %	18.06 %	18.06 %	13.59 %

Table 3.11: Split Policy comparison for Random Walk algorithm for Random Select Policy against Random Split Policy

Edge Deg.	BFS Sp. Pol.	Half Sp. Pol.	Avg.
5	9.09 %	4.25 %	6.67 %
10	36.52 %	14.48 %	25.50 %
Avg.	22.80 %	16.08 %	15.39 %

Table 3.12: Split Policy comparison for Random Walk algorithm for First Available Select Policy against Random Split Policy

path.

We observe that for the Random Select Algorithm the split algorithm comparison against the Random Split Algorithm shows (Table 3.11) 5.8% difference for edge degree 5 graphs, and 21.3% for edge degree 10 graphs on average.

In the case of First Available Select Policy the overall results did not change. But the comparison against Random Split Algorithm vs. other split algorithms shows us 6.6% improvement for edge degree 5 graphs and 25.4% improvement on average. When compared to BFS Algorithm results from Table 3.10 and Random Walk Algorithm Results from Table 3.12, the decrease for sparse graphs can be explained with the locality. The denser the graph, the higher the probability of the selected next node to be an unseen one. Therefore, even if we increase the locality, the randomness of the random walk decreases the probability of choosing a node that is already loaded into the GraphDB memory buffer from disk.

3.4 Evaluation Summary

In this section, we are going to summarize the results of our experiments presented in Section 3.3. The results of our experiments shows us that the following points.

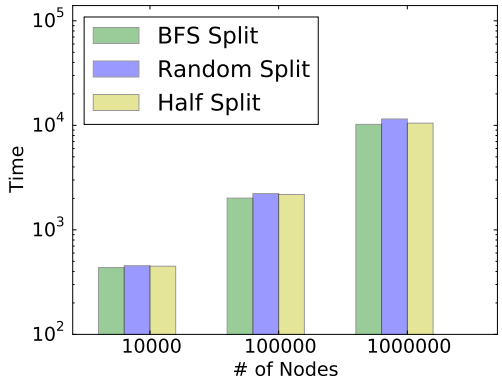


Figure 3.14: Random walk quert on First Available Select Policy created layout with edge degree of 5.

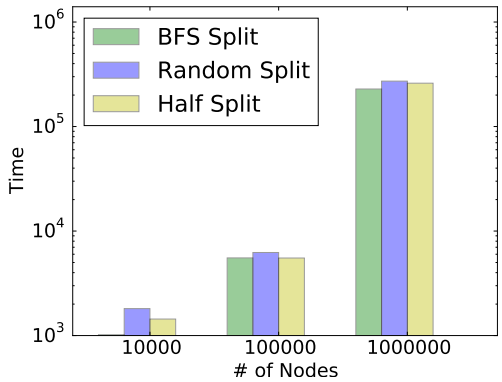


Figure 3.15: Random walk quert on First Available Select Policy created layout with edge degree of 10.

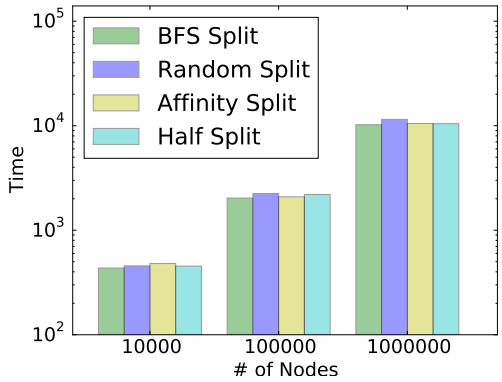


Figure 3.16: Random walk quert on Random Select Policy created layout with edge degree of 5.

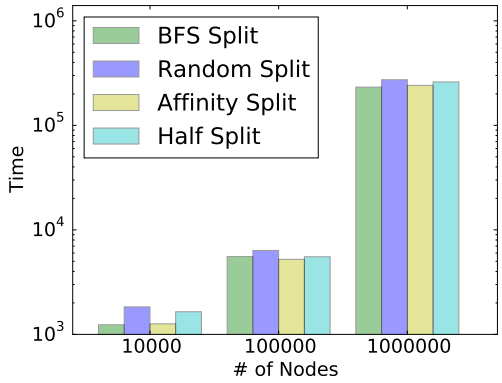


Figure 3.17: Random walk quert on Random Select Policy created layout with edge degree of 10.

First Available Select Policy with BFS Split Policy performed well in most of the experiments. If we look at only the query results, the winner between Affinity Split and BFS Split Policies is Affinity Split. On comparable conditions, Affinity Split took almost 3% less time to complete the query. However, if we take insertion times into account, BFS split beats Affinity Split. The insertion time difference between Affinity Split and BFS Split is almost 6% in favor of BFS Split. Unfortunately, the insertion-query time trade-off doesn't work out well for Affinity Split, the average maximum gain for Affinity Split is 3%. For the BFS Split the average maximum gain is 13%. Inspite of the difference in gain, we cannot say that Affinity Split fails to compete with BFS Split, because in an environment where graph updates are rare and database querying is frequent, Affinity Split Algorithm would be the split algorithm of choice. However, BFS Split would work almost good as Affinity Split in more general use cases.

Creating denser blocks increase the processing speed. FA Select Policy creates denser blocks on disk when compared with Random Select Policy, since it creates a new subgraph when there are no available subgraphs left. The average query time of FA Select with comparable split policies (BFS, Half and Random Split) is 3.8% less than Random Select Policy.

Increasing the edge degree, makes GraphDB run more effectively. As we can see from the split policy comparison tables such as Table 3.11 and 3.8, as

edge degree has gone from 5 to 10 other split policies did a better job compared to Random Split Policy, the base line. On average query algorithms took 9.7% less time to complete with split algorithms(Affinity, BFS, Half Split) on graphs with edge degree 5 and 23.4% on graphs with edge degree 10.

Chapter 4

Related Work

Graph analysis systems and graph databases are not something new, and there is a lot of research being done on the subject. Hadoop [2] is one of the most used distributed platforms and in many graph applications it is being used as a distributed processing and storage system. One example is Giraph [3]. Giraph is an in-memory graph processing/analysis system that runs on the Hadoop platform. It is not a database system, but it runs distributed and very large graphs can be processed with it. The processing system of Giraph is called *vertex-programming*. It is an implementation of Pregel [4]. Pregel is a proprietary software of Google. Vertex-programming is unlike traversal based graph processing. The vertex-centric approach makes graph processing possible by executing a program for each vertex in such a way that each vertex runs their program independently and only communicate via messages. Each vertex has its own state and this state changes at each iteration of the vertex programs. Giraph layouts the graph in the cluster randomly for load balancing. Another example to vertex-centric graph processing systems is GraphLab [6]. Graphlab differs from Pregel-like systems because it uses an asynchronous model for parallelization of the computation. It can be also run as a distributed system that uses MPI based distribution mechanism behind the scenes. Examples up to now were all in-memory systems. An example of a disk based graph processing system is GraphChi [5]. The point of GraphChi is that it runs on a single computer and

is designed to process very large graphs with limited amount of memory. The power of GraphChi comes from its asynchronous bulk processing model with a sliding window processing scheme. All these systems focus on graph processing, and not so much on graph management. The latter is the focus of GraphDB.

There are also similar systems to GraphDB in terms of changing graph layouts as the data is changing, such as Trinity [22], GBASE [23] and Cassovary [24]. These systems are in-memory systems that use graph layout techniques to efficiently process the data. Trinity is a distributed system that tries to efficiently process very large graphs with a reasonable amount of memory in a distributed system. It focuses on partitioning the graph in terms of network communications between cluster computers. GBASE is a graph processing system that focuses on increasing query performance using query optimization and different storage layout techniques. However, none of these systems have investigated the tradeoff between insertion time and query time with respect to adaptive layout.

GraphDB is a graph management system that focuses on both processing and storing graphs. The most notable example to graph databases is Neo4j [1]. It is a disk based graph database that uses disk layouts like GraphDB. However, it is not designed to adapt its layout on the fly while the data is streaming. It provides tools for indexing data using Apache Lucene [25]. It also provides ACID properties. Another example is Titan [26]. Titan is built on top of Hadoop and and similar to GraphDB it uses key-value stores for storage. But unlike GraphDB it uses random layout strategy for load balancing between the machines in the cluster. However, Titan can also be configured to use a basic graph layout that groups together related nodes, with constant number of clusters. Titan is also like Neo4j in that it does not adapt its graph layout as the data is inserted into the database. Titan can also utilize Apache Lucene like indexing mechanisms to accelerate query processing.

Chapter 5

Conclusion

In this work we presented GraphDB, a disk based graph database system that is capable of storing and processing large graphs, when the data is being inserted in a streaming fashion. Excessive data generation is a part of today's problems and the velocity of generated data requires migration from batch data processing systems to streaming data processing systems. Motivated by this need, we proposed a disk based graph database system, GraphDB, that is capable of storing streaming data.

GraphDB, is designed for storing and managing large graphs that may not fit into the main memory of a single computer. It provides traversal APIs to process the data. The novel part our system is that, as the data is streaming the graph layout is updated to improve locality and thus query I/O performance. Therefore, we proposed online layout update algorithms, which are constantly changing the graph layout as the graph data is added to the system, for effective processing of the graph. As part our experimental evaluation, we showed that the proposed techniques have the potential to be used for effective storage and querying of large graphs. For instance, our BFS Graph Split Policy is a good example of how effective online layout updating can be during the processing of the stored graph data.

Bibliography

[1] "Neo4J." `http://neo4j.com/`. retreived 08-Sep-2015.

[2] "Hadoop." `http://hadoop.apache.org/`. retreived 19-Aug-2015.

[3] "Giraph." `http://giraph.apache.org/`. retreived 08-Sep-2015.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[5] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 31–46, USENIX Association, 2012.

[6] C. Guestrin, "Usability in machine learning at scale with graphlab," in *Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management*, CIKM '13, (New York, NY, USA), pp. 5–6, ACM, 2013.

[7] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, Dec. 1998.

[8] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Proceedings of the 1996 ACM/IEEE Conference*

on Supercomputing, Supercomputing '96, (Washington, DC, USA), IEEE Computer Society, 1996.

[9] "HBase." `http://hbase.apache.org/`. retreived 19-Aug-2015.

[10] "Casssandra." `http://cassandra.apache.org/`. retreived 09-Sep-2015.

[11] "MySQL." `https://www.mysql.com/`. retreived 09-Sep-2015.

[12] M. Kryder and C. S. Kim, "After hard drives what comes next?," *Magnetics, IEEE Transactions on*, vol. 45, pp. 3406–3413, Oct 2009.

[13] Wikipedia, "List of device bit rates — Wikipedia, the free encyclopedia," 2015. retreived 19-Aug-2015.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, (Berkeley, CA, USA), pp. 15–15, USENIX Association, 2006.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, Oct. 2007.

[16] O. Y. Margo Seltzer, "A new hashing package for unix," 1991.

[17] "LevelDB Java Implementation." `https://github.com/dain/leveldb`. retreived 03-Sep-2015.

[18] "LevelDB." `https://github.com/google/leveldb`. retreived 03-Sep-2015.

[19] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, pp. 351–385, June 1996.

[20] "Boost Graph Library." `http://www.boost.org/doc/libs/1_55_0/libs/graph/doc/index.html`. retreived 15-Sep-2015.

[21] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," 1999.

[22] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 505–516, ACM, 2013.

[23] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, "Gbase: A scalable and general graph management system," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, (New York, NY, USA), pp. 1091–1099, ACM, 2011.

[24] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "Wtf: The who to follow service at twitter," in *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, (Republic and Canton of Geneva, Switzerland), pp. 505–514, International World Wide Web Conferences Steering Committee, 2013.

[25] "Apache Lucene." `https://lucene.apache.org/`. retreived 28-Nov-2015.

[26] "Titan." `http://thinkaurelius.github.io/titan/`. retreived 28-Nov-2015.