PK-Graph: Partitioned K^2-Tree Graph

Bruno Alexandre Coimbra Morais

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Manuel Antunes Veiga

Examination Committee

Chairperson: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Bruno Emanuel Da Graça Martins

November 2021

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Abstract

Graphs are becoming increasingly larger, having millions of vertices and billions (or even trillions) of edges in some cases. As a result, it's becoming harder and harder to fit the entire graph into the main-memory of a single machine. This may lead to significant overhead by having to read the graph from secondary storage. Thus causing an impact in the performance of queries and the storage requirements of the system. It is relevant to try to minimize the storage requirements of the graph data without degrading access time and, ideally, even improving it. Current graph storage systems store their graphs in an uncompressed format, either in a shared architecture, leading to high space overhead and the inability to store the entire graph in main memory or a distributed architecture, in which the entire graph is partitioned over a cluster of machines and each machine stores only a fragment of the graph in main memory. Our solution extends a distributed graph processing system to utilize a compressed representation of a graph while still allowing to update the graph data, all while maintaining the same processing performance and ideally even improving it.

Keywords

graph databases; graph processing systems; graph representation; optimization; compression

Resumo

Os grafos estão a tornarem-se cada vez maiores, tendo milhões de vértices e bilhões (ou até trilhões) de arestas em alguns casos. Desta forma, está se a tornar cada vez mais difícil colocar o grafo inteiro na memória principal de uma única máquina. Isto pode causar uma sobrecarga significativa por ter que ler o gráfico de armazenamento secundário. Por sua vez, também pode ter um impacto no desempenho no processamento e nos requisitos de armazenamento do sistema. É relevante tentar minimizar os requisitos de armazenamento dos dados do grafo sem degradar o tempo de acesso e, idealmente, até mesmo melhorá-lo. Os sistemas atuais de armazenamento de grafos armazenam estes num formato descompactado, seja numa arquitetura compartilhada, levando a uma grande sobrecarga de espaço e à incapacidade de armazenar o grafo inteiro em memória principal ou numa arquitetura distribuída, na qual todo o grafo é particionado por um grupo de máquinas e cada máquina armazena apenas parte do grafo total em memória principal. A nossa solução estende um sistema de processamento de grafos distribuído para utilizar uma representação compacta de um grafo, que ao mesmo tempo permita atualizar os seus dados, mantendo o mesmo desempenho de processamento e, idealmente, até melhorando-o.

Palavras Chave

base de dados de grafos; sistema de processamento de grafos; representação de grafos; optimizações; compressão;

Contents

1 Introduction **2**

1.1 Graph Storage Architecture . . . 3

1.2 Lossless Graph Compression . . . 3

1.3 Dynamic Graph . . . 4

1.4 Graph Partitioning . . . 4

1.5 Current Shortcomings . . . 5

1.6 Goals and Contributions . . . 5

1.7 Document Structure . . . 5

2 Related Work **6**

2.1 Graph Processing Systems . . . 7

2.1.1 Relevant Systems . . . 9

2.2 Graph Databases . . . 12

2.2.1 Relevant Systems . . . 14

2.3 Optimized Graph Representations . . . 17

2.3.1 Relevant Research Works . . . 19

2.4 Analysis and Discussion . . . 27

3 Architecture **29**

3.1 Overview . . . 30

3.2 Graph . . . 32

3.3 Vertices . . . 34

3.4 Edges . . . 36

3.5 Dynamism . . . 40

3.6 Partitioning . . . 40

3.7 Discussion . . . 43

4 Implementation **44**

4.1 K^2-Tree . . . 45

4.1.1 Construction . . . 46

4.1.2 Iteration . . . 48

4.2 Graph . . . 49

4.2.1 Partitioning . . . 50

4.2.2 Dynamism . . . 50

4.3 Partitions . . . 51

4.3.1 Construction . . . 52

4.3.2 Processing . . . 53

4.4 Analysis . . . 55

4.4.1 Limitations . . . 56

5 Evaluation **58**

5.1 Metrics . . . 59

5.2 Microbenchmarks . . . 60

5.2.1 Memory . . . 60

5.2.2 Build . . . 62

5.2.3 Iterator . . . 63

5.2.4 Aggregate Messages . . . 63

5.2.5 Dynamism . . . 65

5.3 Macrobenchmarks . . . 66

5.3.1 Setup . . . 66

5.3.2 Datasets . . . 67

5.3.3 Memory Overhead . . . 67

5.3.4 Build . . . 68

5.3.5 Iteration . . . 69

5.3.6 PageRank . . . 70

5.3.7 Triangle Count . . . 71

5.3.8 Throughput . . . 72

5.3.9 CPU Usage . . . 73

5.4 Optimization . . . 74

5.5 Analysis . . . 77

6 Conclusion **78**

6.1 System Limitations and Future Work . . . 80

Bibliography **80**

List of Figures

2.1 Graph Processing Systems Taxonomy 7
2.2 Graph Databases Taxonomy 13
2.3 Optimized Graph Representations Taxonomy 17
2.4 Graph and corresponding adjacency matrix and k^2-tree representation. 20
2.5 Schema component of the AttK^2-tree 20
2.6 Data components of the AttK^2-tree 21
2.7 Relations component of the AttK^2-tree 21
2.8 dk^2-tree representation and the corresponding T_{tree} and L_{tree}. 22
2.9 Example of the Dynamic Trie representation. 23
2.10 Diagram of the internal representation used by the DEX graph database. 24
2.11 Example of the compressed sparse row representation. 25
2.12 Example of the packed compressed sparse row representation. 25
2.13 Example of the GraphZIP compression method. 26
2.14 Example of the *GSSC* compression method. 27

3.1 Architecture overview of our system. 31
3.2 Graph component in our architecture 32
3.3 Interface of the PKGraph class 33
3.4 Distributed graph work in a cluster 34
3.5 Vertex components in our architecture 34
3.6 Interface of the ShippableVertexPartition class 35
3.7 Edge components in our architecture 36
3.8 Interface of the PKEdgePartition class 37
3.9 Example of vertices exchanging messages. 38
3.10 Interface of the PKReplicatedVertexView class 39
3.11 Interface of the DynamicGraph class 40
3.12 Adjacency matrix partitioning scheme 41
3.13 Overview of the existing partition strategies 42

3.14 Interface of the PartitionStrategy class 42

4.1 Diagram of how a K^2-tree is represented using a bitset. 45
4.2 Example of calculating the *K2TreeIndex* for an edge. 46
4.3 Example of building a K^2-tree. 47
4.4 Transforming the builder to the final compressed K^2-tree. 47
4.5 Example of iterating a K^2-tree. 49
4.6 Metadata of the *PKEdgePartition* class 51
4.7 Example of a edge partition with an offset from the global adjacency matrix. 52
4.8 Example of building a new edge partition. 53
4.9 Example of testing the intersection between two partitions. 54
4.10 Example of building vertex clusters in a edge partition. 56
4.11 Example of data locality of adjacency matrices. 57

5.1 Edge partition memory overhead in relation to the number of edges 61
5.2 Edge partition memory overhead with poor data locality 61
5.3 Edge partition build latency compared with partition size 62
5.4 Edge partition iteration latency compared with number of edges 63
5.5 Edge partition aggregate messages latency compared with number of edges 64
5.6 Edge partition aggregate messages with source index with varying percentage of active vertices 64
5.7 Results of adding new edges to an existing edge partition 65
5.8 Results of removing existing edges from an existing edge partition 66
5.9 Overview of the cluster used to execute spark jobs. 66
5.10 Results of the memory overhead for each dataset 68
5.11 Results of the build latency for each dataset 69
5.12 Results of the iteration latency for each dataset 70
5.13 Results of the latency of the Page Rank algorithm for each dataset 71
5.14 Results of the latency of the Triangle Count algorithm for each dataset 72
5.15 Results of the average throughput for each dataset 73
5.16 Results of the average Central Processing Unit (CPU) usage for each dataset 74
5.17 Example of assigning row major indices to a parent node 75
5.18 Example of the optimized iterator representation of a standard k^2-tree 75
5.19 Example of iterating the edges of the optimized iterator representation. 76
5.20 Edge partition iteration results using the optimized iterator 76
5.21 Edge partition memory overhead results using the optimized iterator 77

List of Tables

2.1 Graph Processing System Classification 10

2.2 Graph Database System Classification. 15

2.3 Optimized Graph Representation Classification. 19

5.1 Table describing the datasets used in our benchmarks. 67

List of Algorithms

3.1 Algorithm to iterate the vertices of a given partition 35
3.2 Algorithm to iterate the edges of a given partition 39

Acronyms

API	Application Program Interface
AWS	Amazon Web Services
BSP	Bulk Synchronous Parallel
CPU	Central Processing Unit
CSR	Compressed Sparse Row
CSC	Compressed Sparse Column
EMR	Elastic Map Reduce
GAS	Gather, Apply and Scatter
HDFS	Hadoop Distributed File System
IO	Input/Output
LSM	Log Structured Merge
PCSR	Packed Compressed Sparse Row
PMA	Packed Memory Array
POS	Predicate-Object-Subject
PSO	Predicate-Subject-Object
RAM	Random Access Memory
RDD	Resilient Distributed Dataset
RDG	Resilient Distributed Graph
RDF	Resource Description Framework
S3	Simple Storage Service
UML	Unified Modeling Language

1 Introduction

Contents

1.1 Graph Storage Architecture . . . 3
1.2 Lossless Graph Compression . . . 3
1.3 Dynamic Graph . . . 4
1.4 Graph Partitioning . . . 4
1.5 Current Shortcomings . . . 5
1.6 Goals and Contributions . . . 5
1.7 Document Structure . . . 5

Graphs [1] are now more relevant than ever, being used in social networks [2, 3], biology [4, 5], the web [6, 7], cryptocurrency [8, 9], and many more fields (e.g., managing community clouds [10]). Their popularity arises from the fact that they naturally model problems that other data structures cannot. Graphs are also becoming increasingly larger, having millions of vertices and billions (or even trillions) of edges in some cases [11, 12]. As a result, the space requirements of a graph have increased. It is becoming increasingly more difficult to fit the entire graph into the main memory of a single machine. This may lead to a significant overhead by having to read the graph from secondary storage. Thus it is relevant to try to minimize the storage requirements of the graph without degrading access time and, ideally, even improving it.

1.1 Graph Storage Architecture

There are two common architectures for graph storage systems. A shared architecture [13, 14], where the entire graph is stored in a shared memory location and accessed by other processors. Typically, the graph is not stored in main memory due to space requirements, and thus must be stored in secondary memory, leading to a significant overhead when accessing the graph.

A distributed architecture [15–19] in which the entire graph is partitioned over a cluster of machines and each machine stores only a shard of the graph. Seeing as the graph is split across multiple machines, it becomes much more feasible to store a significantly smaller portion of the graph in main-memory. Still, partitioning a graph into many partitions can result in each partition being too large to store in machines with standard configuration, which in turn would lead to higher space requirements for each machine, resulting in higher operational costs.

Shared architectures rely on hardware that allows processing large graphs in a single machine. For example, Mosaic [20] is able to process a trillion-edge graph in a single machine, using fast storage media and massively parallel co-processors. On the other hand, distributed architectures rely on distributed deployments, based on multiple machines with standard configurations.

This indicates that with a sufficiently powerful machine one can match, or even outperform, distributed solutions. However, not all users have access to specialized hardware, making distributed solutions relevant and our focus for this work.

1.2 Lossless Graph Compression

Current solutions, using both shared and distributed architectures, store graphs in a uncompressed format [13–19, 21]. By using a lossless graph compression technique, it is possible to store the graph in a compressed format that can be stored in the main memory of a single machine [22–25]. All while maintaining the same performance, or even better, when accessing the graph.

This type of compression reduces the storage requirements needed without removing any existing information about the structure (lossless), while still allowing for direct access to the underlying data. This means that the data does not need to be decompressed every time it is accessed. Since compressed representations can more efficiently represent the graph structure and allow for direct access to the underlying data, they typically have better performance than the corresponding uncompressed representations [22–24].

The graph topology can also play a big role in determining the effectiveness of the compression technique, seeing as some compressed representations are more efficient when used in sparse graphs as opposed to dense graphs [22].

1.3 Dynamic Graph

It also may be relevant to modify the graph, such as adding or removing edges/vertices, without having to reconstruct the entire graph, i.e., to support a fully updatable or dynamic graph. For example, some popular graph algorithms (i.e., PageRank) require the attributes stored in the vertices/edges to be mutated. Current solutions do not allow adding new vertices/edges to a graph [13, 15, 17–19] but still allow to modify the attributes of the graph. In these solutions, adding new elements requires the construction of a new graph, with the added elements, presenting a severe space overhead and entailing processing effort in generating the resulting complete new graph. Thus, it is relevant to provide a solution that allows for a fully dynamic graph that can add new elements without having to construct the entire graph again.

1.4 Graph Partitioning

When using a distributed architecture, it is important to define how the graph will be partitioned throughout all machines in the cluster. The partitioning strategy employed should try to evenly distribute the graph through all machines, to guarantee that the workload is uniformly distributed. The strategy should also guarantee that the space requirements of all machines are similar and that the portion of the graph assigned to each machine can be stored in main memory.

Current solutions are centered in partitioning graphs based on edges, to better distribute work among computing nodes [13, 15]. This leads edges to be assigned to unique partitions and vertices to be replicated throughout various partitions. In a worst-case scenario, a vertex would need to be replicated throughout all partitions. This approach is used because the number of edges is typically much higher than the number of vertices, leading to smaller storage requirements when replicating vertices.

A possible solution would consist of placing edges with matching vertices in the same partition, thus reducing the need to replicate the vertices throughout many partitions. However this solution no longer guarantees an evenly distributed workload. An ideal solution would try to minimize the replication of vertices while still evenly distributing the graph data throughout the cluster.

1.5 Current Shortcomings

Our solution aims to tackle several shortcomings that current solutions present, such as: i) Not being able to store large graphs in main memory, requiring access to secondary storage which is much slower; ii) Storing graphs in a uncompressed format, leading to worse storage efficiency and processing performance than compressed representations; ii) Immutable graphs that do not support removing or adding vertices/edges, requiring the entire graph to be re-constructed from scratch when adding new elements (even if a single one, or a few, in a graph with millions).

1.6 Goals and Contributions

The main goal of this work is to design and develop an extension to the storage component of a relevant distributed graph processing system so that the processed graph is made more space-efficient by using a lossless compressed representation. The solution should achieve similar performance to the uncompressed version, with relatively reduced overhead. The solution should also allow for the graph to be fully dynamic, by being possible to mutate attributes and add new vertices and edges. To accomplish this goal, we set the following objectives:

1. Survey the state-of-the-art work in graph databases, graph processing systems, and optimized graph representations and processing.
2. Use an optimized dynamic graph representation to efficiently represent an attributed graph.
3. Implement an extension to the storage component of a relevant graph processing system supporting the optimized dynamic graph representation.
4. Perform a detailed evaluation of our implementation to determine to what extent it fulfills our requirements and present the results obtained.

1.7 Document Structure

This document is structured as follows: Chapter 2 presents a survey of the state-of-the-art work done in graph processing systems, graph databases and optimized graph representations and processing. In Chapter 3, we present the architecture of our solution, in Chapter 4, we detail our implementation of the solution. In Chapter 5, we describe the evaluation methodology and the results obtained for our implementation. Finally, in Chapter 6 we conclude by summarizing the work, present our thoughts on the topic and discuss some possible future work.

2 Related Work

Contents

2.1 Graph Processing Systems 7
2.2 Graph Databases 12
2.3 Optimized Graph Representations 17
2.4 Analysis and Discussion 27

In this chapter, we present important work on the topic of graph representation and graph processing. We start by presenting relevant graph processing systems (Section 2.1), focusing on how they handle graph storage and processing, and graph database systems (Section 2.2). In Section 2.3 we describe state-of-the-art optimized graph representations and processing.

2.1 Graph Processing Systems

In this section we present a taxonomy to study, analyse, and classify the most relevant graph processing systems [26, 27]. We classify graph processing systems according to six features: **Architecture**, **Programming Model**, **Partitioning**, **Storage**, **Fault Tolerance** and **Dynamicity**. Figure 2.1 presents a taxonomy of the state-of-the-art work in graph processing systems.

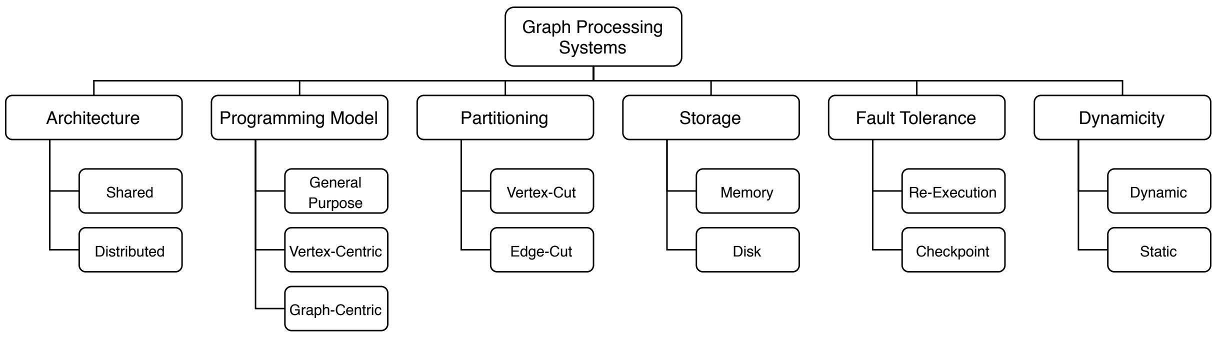


Figure 2.1: Graph Processing Systems Taxonomy

Architecture: The architecture of graph processing systems is divided into two categories: Shared and Distributed.

Shared architectures store the entire graph in a centralized location, allowing multiple processors to access the same memory. Given specialized hardware, this type of architecture can achieve better results than distributed architectures. However, because the entire graph is stored in a single machine the storage requirements are much higher. It also does not provide a very scalable solution.

Distributed architectures partitions the graph throughout a cluster of processors, where each processor stores only a fraction of the total graph. This type of architecture is much more scalable than the shared architecture. However, it typically requires for the processors to communicate with each other trough a network for synchronization purposes, leading to some overhead.

Programming Model: The programming model (PM) relates to the high-level programming interface available to the user, we divide this feature into 3 categories: **G**eneral **P**urpose, **V**ertex-**C**entric and **G**raph-**C**entric.

General Purpose programming models are not made specifically to handle graphs, such as the *MapReduce* framework. Because of this, these models offer a substantial reduction in performance when compared to programming models made specifically for graph processing. Typically graph processing systems that use these models are extensions to already existing general purpose frameworks.

Vertex-Centric models, also sometimes referred to as *"think-like-a-vertex"*, execute a user-defined function in the context of each vertex, iteratively. In each iteration, each vertex executes the user defined function, in parallel, manipulating only local data. The vertices can also send and receive messages from their neighbors. The algorithm terminates when no messages are exchanged between any vertices. In distributed systems the vertices may be located in different partitions, involving some communication overhead when exchanging messages.

Graph-Centric models are focused on performing computations in the context of a sub-graph. This model tries to address the issue of communication latency between vertices located in different partitions. Instead of storing different vertices in each partition it proposes to store sub-graphs. If the sub-graph is well partitioned, the communication overhead between vertices can be substantially reduced, since the communication now occurs locally.

Partitioning: This feature relates to how a graph is partitioned to later be processed in parallel by several processors. We divide this feature into two categories: Vertex-Cut and Edge-Cut.

Vertex-Cut partitioning divides the graph by its vertices, leading vertices to be distributed across multiple partitions and edges to be assigned to a unique partition. For many real-world graphs where the degree distribution follows a power law, a vertex-cut leads to a more balanced partitioning. Typically in these scenarios, the computations are expressed from the context of an edge to allow for more efficient parallel computations.

Edge-Cut partitioning divides the graph by its edges, leading to edges to be distributed across multiple partitions and the vertices to be assigned to a unique partition. Since graphs typically have many more edges than vertices, this type of partitioning leads to higher space usage than the vertex-cut partitioning.

Storage: This feature relates to how the graph data is stored, either in **M**emory or in **D**isk, in cases where the graph is too large to fit into memory.

Memory storage, or main memory, although typically smaller than disk storage, has much faster access times. Most distributed graph processing systems use this approach, since they have access to a cluster of machines, where the main memory of each machine is used to store part of the graph. This results in the overall system having more available main-memory than a single machine.

Disk storage, or secondary-memory, is typically larger than in-memory storage, thus allowing to store much larger graphs. However, the access time is substantially larger than in-memory solutions. Some systems use a hybrid approach, storing the graph first in memory and using secondary storage to store the rest of the graph.

Fault Tolerance: This feature describes the type of fault tolerance the graph processing system has. We divide this feature into two categories: Re-Execution and Checkpoint.

Re-Execution relates to graph processing systems that require the state to be recomputed when a faults occurs. This type of fault tolerance can be very costly depending on the size of the entire state and how often faults occur.

Checkpoint relates to graph processing systems that use a checkpoint system to store the current state. This type of fault tolerance avoids having to recompute the entire state and instead periodically stores the current state to later be reused when a fault occurs. This type of solution has higher space requirements than the re-execution approach, but, depending on how often faults occur, may have better processing performance.

Dynamicity: This feature relates to the type of dynamicity that is allowed on the graph data. This feature is divided into two categories: Dynamic and Static.

Dynamic graphs allow the data to be mutated, either by changing attributes or adding or removing vertices/edges. This type of solution typically has higher space requirements since the dynamic structures require more information to efficiently mutate the graph. Most graph processing systems do not provide a mechanism to mutate the graph. Some systems only allow for the attributes of vertices and edges to be changed.

Static graphs do not allow the data to be mutated. Graph processing systems that only support static graphs are limited in the algorithms that can be implemented. For example, the *PageRank* algorithm requires the attributes of vertices to be mutated. This type of solution typically requires less storage requirements since the data structures are immutable and can be optimized as such.

2.1.1 Relevant Systems

In this section we present the most relevant graph processing systems. Table 2.1 shows the most relevant Graph Processing Systems identified in our research, classified according to the previously described taxonomy, highlighting their distinctive features.

GraphX: Spark's Application Program Interface (API) for graphs and graph-parallel computation [15]. Data storage is handled by Spark's Resilient Distributed Dataset (RDD) which represents an immutable collection of elements that allow for several transformations (e.g., map, filter) and that can be processed in a distributed fashion by splitting elements into various partitions and having different machines in the cluster process different partitions.

The data is obtained from either a user-configured source (Hadoop Distributed File System (HDFS), in-memory collection) or by applying transformations to other RDDs. All transformations create a new

System/Work	Architecture	PM	Partitioning	Storage	Fault Tolerance	Dynamicity
GraphX [15]	Distributed	VC	Vertex-Cut	M/D	Checkpoint	Static
Gelly [19]	Distributed	VC	Vertex-Cut	Memory	Checkpoint	Static
Giraph [16]	Distributed	VC	Edge-Cut	Memory	Checkpoint	Dynamic
Giraph++ [16]	Distributed	GC	Edge-Cut	Memory	Checkpoint	Dynamic
GBase [17]	Distributed	VC	Edge-Cut	Disk	Re-Execution	Static
GPS [18]	Distributed	VC	Edge-Cut	Memory	Checkpoint	Static
GraphLab [13]	Shared	VC	Vertex-Cut	Memory	Checkpoint	Static
GraphChi [14]	Shared	VC	Edge-Cut	Disk	Re-Execution	Dynamic

Table 2.1: Graph Processing System Classification

RDD that will be evaluated lazily. RDDs can also be persisted, either to disk or to memory in both a serialized and deserialized format. GraphX implements a Resilient Distributed Graph (RDG), an immutable graph made up of vertices and edges that can be split into various partitions using a vertex-cut. A vertex-cut splits the graph along the vertices, meaning that vertices span multiple machines in the cluster while edges are evenly assigned.

The graph is represented using three unordered horizontally partitioned tables implemented using RDDs: i) edge table that stores all edges grouped by their assigned partitions, each edge storing both the start and end vertex identifier and user-defined data; ii) vertex data table, similar to the edge table but stores the vertices; iii) and the vertex map that maps each vertex identifier to their assigned partition, possibly spanning multiple partitions.

Gelly: Graph API for Flink [19] providing methods to create, transform and modify graphs, as well as a library of graph algorithms. Opposed to Spark, Flink is a datastream–oriented big data framework focusing on processing a potentially indefinite stream of events within a user-defined pipeline. Similar to GraphX, Gelly implements its graph abstraction using the equivalent to RDDs in Flink, DataSets, and supports three different computation models for the distributed processing of graphs: a *vertex-centric* model; the *scatter-gather* model, where vertices send messages to each other and update based on the messages received; the *gather-sum-apply* model [28], where a user-defined function is executed in parallel on the edges and neighbours of each vertex (gather), producing a partial value that is aggregated to a single value (sum) and then applied to each vertex (apply).

Giraph: An iterative graph processing system [16] built on top of *Hadoop*, designed for high scalability. Implements the *Pregel* [29] model and other features such as master computation[1], sharded aggregators[2], edge-oriented input and out-of-core computation. Computations are done in a serious of iterations called *supersteps*. The programming model used is vertex-centric. It uses the Bulk Synchronous Parallel (BSP) [30] model to allow for distributed parallel computations using a set of workers. One of the workers acts as the master and coordinates with the remaining workers to perform a graph processing job. The set of vertices is partitioned randomly and each vertex is assigned, alongside all of its outgoing edges, to a partition based on its identifier. This partitioning corresponds to an edge-cut where a vertex is present in a single partition and edges can exist in multiple partitions.

Giraph++: A *distributed* graph processing system [16], based on Giraph, that utilizes the Graph-Centric model. This model allows the programmer to perform computations in the context of an sub-graph, instead of a single vertex. Each sub-graph corresponds to a partition. The vertices of each sub-graph are classified in either one of two categories: *internal* vertex or *boundary* vertex. Internal vertices are unique in a given sub-graph, called the *owner*, and store the vertex and edge values, while boundary vertices can be present in multiple sub-graphs and only store the vertex value, which is a local copy of the actual vertex value stored in the *owner* sub-graph. Changes to boundary vertices are propagated to their primary copies.

GBase: A distributed and scalable graph processing system [17] that stores graphs using a method based on adjacency matrices. It supports both vertex-centric and edge-centric programming models. The graph vertices are partitioned into several partitions, that are then grouped to form several homogeneous blocks. Each block has a source partition and a destination partition, that is then compressed using standard compression algorithms such as Gzip [31] and Elias-γ[3]. These blocks are stored together with some metadata identifying the row and column of the block on the global adjacency matrix.

GPS: A distributed graph processing system [18] based on *Pregel* and a vertex-centric programming model based on bulk synchronous processing. The graph vertices are distributed randomly to the computing nodes in the cluster. The graph input is stored in HDFS files containing in each line the vertex identifier and its outgoing neighbours. The coordination of the system is done using a master-worker pattern, where the master nodes instruct the works to start a new superstep and to perform checkpoints of their state for fault tolerance.

GraphLab: A *shared* graph processing system [13] that stores graphs *in-memory* and uses the BSP

[1]Workers use ZooKeeper to elect a master that will coordinate computation
[2]Each aggregator is assigned to a worker
[3]Universal code encoding positive integers, commonly used when the upper-bound of the integers cannot be determined beforehand.

model with some modifications. Instead of a single computation function, it uses the Gather, Apply and Scatter (GAS) model where each vertex gathers data from its neighbors, applies the computation function to itself and scatters the relevant information to its neighbors. The partitioning of the graph is done using vertex-cuts to better distributed the work of vertices with large degrees. There exists a global shared state that is kept as an associative map, mapping each key to an arbitrary block of data.

GraphChi: A disk-based graph processing system [14] capable of performing efficient computations on graphs with millions of edges. The graph data is stored on *disk* using a compressed graph representation called Compressed Sparse Row (CSR), which is equivalent to storing the graph as adjacency sets. To efficiently obtain the in-edges of a vertex, the transpose graph is also stored in a graph representation called Compressed Sparse Column (CSC), therefore each edge is stored twice. The system uses a method called Parallel Sliding Windows to process a graph with mutable edge values from disk by splitting the vertices of a graph into multiple intervals associated with *shards*. The graphs can also be *mutated*, more specifically, it is possible to change the graph structure by adding new edges.

Final Remarks

Recent works in graph processing systems include *CIC-PIM* by Zhang, Yongxuan, et al. [32], which trades spare computing power for memory space, allowing for more memory and cache efficient graphs by using the sparse features of large-scale graphs.

Another recent work is *DZiG* by Mariappan, Mugilan, et al. [33], which offers a novel approach in graph processing systems by exploiting the sparsity of graphs in their incremental processing. Incremental processing of graphs with streaming and approximate computing is explored in VeilGraph [34].

2.2 Graph Databases

In this section we present a taxonomy to study, analyse, and classify the most relevant graph database systems [35]. We classify graph databases according to five features: **Architecture**, **Model**, **Query Language**, **Storage** and **Topology**. Figure 2.2 presents a taxonomy of the state-of-the-art work in graph database systems.

Architecture: The architecture of graph databases can be divided into two categories: Native or Non-Native. This distinction relates to how graph storage is handled by the database system.

Native architectures are categorized by an exclusive preference to store graph workloads across its entire stack. Their storage system is specifically tailored to store graph data. This leads to much better performance when handling graphs, compared with more general databases. Another characteristic of native architectures is that of index-free adjacency. This means that vertices physically point to other adjacent vertices by having direct physical Random Access Memory (RAM) addresses, leading to faster

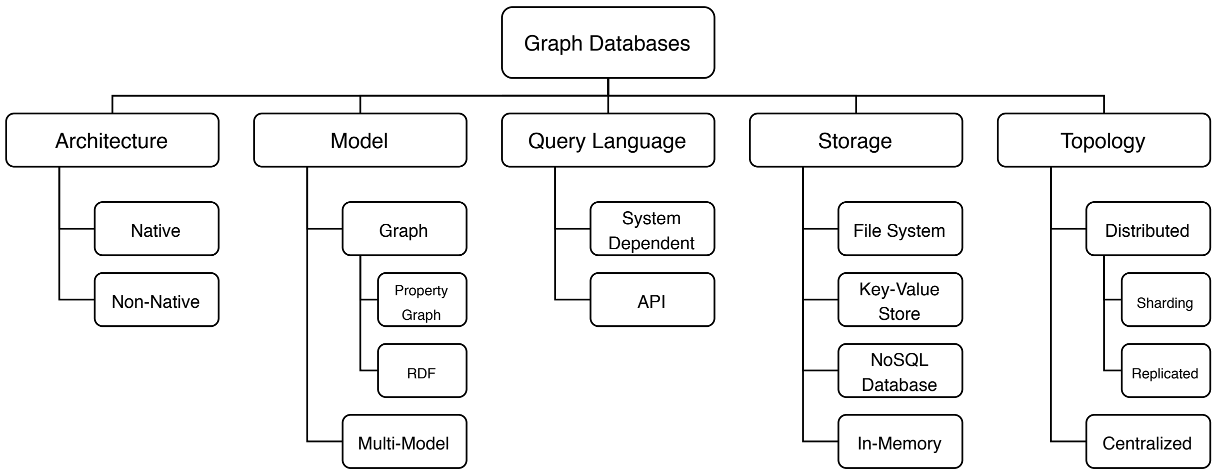


Figure 2.2: Graph Databases Taxonomy

traversal times. However, this may degrade the performance of queries that do not make use of graph traversal.

Non-Native architectures use an external data source, typically *NoSQL*, to store the graph data in a non-optimized representation, leading to worse performance in general. Their storage system is not optimized specifically for graph data but instead for other storage models. The data is typically translated from that storage model (i.e., columnar, relational, document) as a graph, which requires the database management system to perform costly transactions to and from the primary storage model.

Model: The model relates to how the graph is represented by the storage system. We identify two main distinctions: Graph models and Multi-model.

Graph models can only represent graphs, typically in an optimized format. Inside this model there are two main representations: Property Graph and Resource Description Framework (RDF). The property graph model represents attributed graphs (sometimes multi-graphs) where both the vertices and edges have user-defined attributes attached to them. Typically, the vertices and edges are also assigned labels, which function as the type of that object. In the RDF model the graph is represented with a series of triples *subject-predicate-object*. The *subject* is a vertex, the *predicate* is an edge and the *object* is either another vertex or a literal value. In the RDF model the vertices and the edges have no internal data, they are composed only by their identifier. In order to attach an attribute to a vertex, a new vertex containing a literal value must be created and connected through an edge to the original vertex.

Multi-models are capable of representing multiple storage models (Documents, SQL Tables, etc.) including graphs, sometimes keeping these models separate and without much performance overhead. Some implementations support multiple models while still maintaining optimized representations for each.

Query Language: The query language is the language used to perform queries on the graph data. Because there was no standard query language until recently with *GQL*[4], most of the existing graph database systems implemented their own custom query language. Thus we separate this feature into two different categories: System Dependent languages that are implemented specifically for a database system and API which describes database systems that only offer a public API, typically in the same programming language used to implement the database, to perform queries.

Storage: The storage component pertains to the location the graph data is persisted. The most relevant systems either store the graph data in File system directly, including distributed file systems such as HDFS, in a Key-Value Store where the vertices and edges are stored by mapping their identifier to their attributes, or in a NoSQL Database adapted to store graph data.

Topology: The topology of the database is divided into two main categories: Distributed and Centralized.

Distributed systems distribute the graph across multiple machines. These systems offer high availability and fault tolerance. Inside this category we further divide into two sub-categories: Sharding and Replicated. Sharding horizontally partitions the graph across multiple machines to evenly distribute the workload. This approach reduces the storage requirements of each individual machine, while still allowing for the parallel processing of the graph. Replicated stores a copy of the entire graph in each machine, typically to tolerate faults or maintain availability. Although this approach leads to overall higher storage requirements, it has the benefit of offering much higher availability of the entire graph.

Centralized systems store the entire graph in a single machine. In some cases, where specialized hardware is available, these types of systems may have similar or even better performance than distributed systems.

2.2.1 Relevant Systems

In this section we present the most relevant graph database systems. Table 2.2 shows the most relevant Graph Database Systems identified in our research, classified according to the previously described taxonomy, highlighting their distinctive features.

Neo4J: A *native* graph database [21] platform used to store, query, analyse and manage highly connected data in *property graphs*, providing its own query language (*Cypher*). The graph is persisted to *disk* using 3 files: *nodestore.db* to store nodes, *relationship.db* to store relationships (edges) and *property.db* to store the properties (attributes) of nodes and relationships. Data is stored on disk as linked lists of fixed-size records. Properties are stored as a linked list of property records, each holding a key and value and pointing to the next property. Each node and relationship references its first property record. The nodes also reference the first relationship in its relationship chain. Each relationship refer-

[4]Graph Query Language Standard `https://www.gqlstandards.org/`

System/Work	Architecture	Model	Query Language	Storage	Topology
Neo4J [21]	Native	Property Graph	System Dependent	File System	Sharding*
DGraph	Native	RDF	System Dependent	Key-Value Store	Sharding
TigerGraph [36]	Native	Property Graph	System Dependent	In-Memory	Sharding
DEX [37]	Native	Property Graph	API	File System	Centralized
GraphDB	Native	RDF	System Dependent	File System	Replicated*
JanusGraph	Non-Native	Property Graph	System Dependent	NoSQL Database	Sharding
NebulaGraph	Non-Native	Property Graph	System Dependent	Key-Value Store	Sharding
OrientDB	Non-Native	Property Graph	System Dependent	File System	Sharding

* only supported in the enterprise edition

Table 2.2: Graph Database System Classification.

ences its start and end nodes. It also references the previous and next relationship records for the start and end nodes, respectively.

DGraph: A distributed database[5] with a *native* graph backend using the *GraphQL* query language. Data is stored in a key-value database named Badger that uses an Log Structured Merge (LSM)-tree based design to store data in the RDF model. The subject is a node, the predicate is a relationship, and the object can be another node or a primitive data type representing the connection from a node to an attribute value.

TigerGraph: A native parallel graph database [36] with an engine that computes queries and analytics in massively parallel processing fashion for significant scale-up and scale-out performance. The graph is stored *in-memory*, with the user being able to specify how much of the available memory can be used to

[5]DGraph `https://dgraph.io/`

store the graph. If the graph does not fit into memory the rest will spill into *disk*. Data values are stored in encoded formats that effectively compress the data. The data does not need to be decompressed when used internally, only for displaying the information to the user. The system also allows for distributed computing supporting two classical programming paradigms: *vertex-centric* and *edge-centric*.

DEX: A graph database system [37] capable of efficiently performing out-of-core data management with the use of bitmap structures. It represents graphs using the *property graph* model while also supporting multi-graphs, allowing for multiple edges between the same two vertices. The implementation of the system is based on assigning each object (vertex or edge) an unique identifier (*oid*) and using two data structures: *bitmaps* and *maps*. A bitmap or bit-vector is a collection of presence bits that denotes which objects are selected or related to other objects. The *maps* map each object identifier to a specific value. These two structures together create a more complex one called *link*; this structure allows to efficiently retrieve a value given an identifier and to retrieve all identifiers associated with a given value. The graphs are stored out-of-core with each storage component being divided into multiple pages of 64KB each.

GraphDB: A semantic graph database[6] (also called RDF triplestores) built on the RDF4J framework, designed for storing and querying of RDF data. The graph data is persisted to the *filesystem* to a pre-configured directory, consisting of two main indices: the Predicate-Object-Subject (POS) index and the Predicate-Subject-Object (PSO) index. Both of these indexes are used to handle RDF data. The system also supports indexing the context identifier of statements, to speed searches, and indexing literal values allowing for faster look-ups of numeric and date/time object values. The query language used by the system is the *SPARQL*, the standardized query language for RDF graphs.

JanusGraph: A *non-native* scalable graph database[7] optimized for storing and querying graphs containing hundreds of billions of vertices and edges distributed across a multi-machine cluster (*sharding*). The storage backend implementation is provided by third-parties, supporting backends such as *Apache Cassandra* and *Google Cloud Bigtable*. The system supports the property graph model through *Apache TinkerPop*, as well as the *Gremlin* query language.

Nebula Graph: A horizontally scalable distributed open-source graph database[8] using a directed *property graph* as its data model. The storage engine is based on RocksDB, an embeddable persistent key-value store. The vertices are stored in the key-value store using a key composed of the vertex identifier and the vertex tag (or label) and its encoded attributes as the value. Edges are split into in-edge and out-edge and stored as separate key-value entries in the store, and each edge is stored in a different

[6]GraphDB `https://graphdb.ontotext.com/`
[7]JanusGraph `https://janusgraph.org/`
[8]NebulaGraph `https://nebula-graph.io/`

partition.

OrientDB: A *non-native* multi-model NoSQL database management system[9] capable of handling Graph, Document, Key/Value and Object models. In the case of graphs the model used is the *property graph*. The system can be configured to be *sharded* by using a Multi-Master architecture, thus partitioning the graph data across multiple workers. In this type of distributed architecture each server can read and write to the database. Optionally, servers can function as *replicas* and only be able to read from the database. The graphs are stored on *disk* with a caching mechanism to reduce I/O requests.

2.3 Optimized Graph Representations

In this section we present the state-of-the-art work for optimized graph representations [38] and a taxonomy to study, analyse, and classify them. We classify graph representations using five features: **Approach**, **Attributes**, **Edges**, **Graph Type** and **Dynamicity**. Figure 2.3 presents a taxonomy of the state-of-the-art work in optimized graph representations.

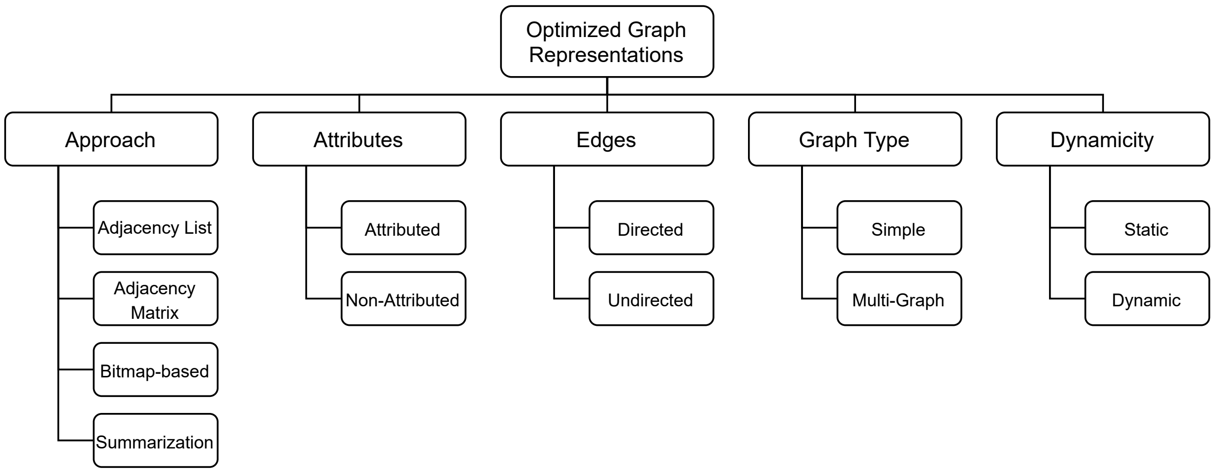


Figure 2.3: Optimized Graph Representations Taxonomy

Approach: The approach relates to the underlying method that the optimized representation is based on, we divide this feature into four categories: Adjacency List, Adjacency Matrix, Bitmap-based and Summarization.

Adjacency Lists keep an array of vertices where each entry stores a pointer to a linked list of edges. The pointer at index *u* in the node list points to a linked list where each element *v* is an outgoing edge(*u*, *v*). Adjacency lists support fast inserts but slow search because the edges are unsorted.

Adjacency Matrices are $n \times n$ matrices of n vertices, where each entry is a single bit and the entry [i, j] corresponds to the edge(i, j), or to a non-existent edge if the entry has a 0 value. It is very efficient at

[9]OrientDB https://www.orientdb.org/

storing dense graphs, however, it wastes space when the graph is very sparse, requiring $O(n^2)$ space.

Bitmap-based approaches rely on values mapped to bitmaps where each bit corresponds to a vertex or an edge, depending on the context. In these representations complex operations can be performed efficiently by using basic binary operators between bitmaps.

Summarization approaches have the goal of providing a smaller graph description and leaving the actual compression has secondary result. The actual technique used various from work to work but typically consists in forming groups of vertices, called *super-vertices* or *cliques*, that are connected together through *super-edges* and together represent a new graph.

Attributes: This feature describes whether the graph representation supports user data attached to vertices and/or edges. They are also denominated as *properties* in some works. We divide this feature into two categories: Attributed and Non-Attributed.

Attributed graphs allow for attributes in their vertices and/or edges. Some solutions also focus on further compressing the attributes of the graph. This type of graphs can also be categorized as property graphs.

Non-Attributed graphs do not support attributes in their vertices and/or edges. Most optimized representations do not focus on the compression of attributes, and only address on how to compress vertices and edges. However, attributes can still be attached to the optimized structure by using other complementary data structures.

Edges: This feature relates to the direction of the edges of the graph. We divide this feature into two categories: Directed and Undirected.

Directed graphs define a direction in the edge between two vertices. Directed graphs can always support undirected edges by simply creating two directed edges in both directions. Although this would mean that they would have twice the number of edges that an undirected graph representation would need.

Undirected graphs do not have any direction attached to their edges. This means that any edge is bidirectional. Because of this, these types of graphs usually have smaller space requirements than directed graphs, since a single edge in a undirected graph would require two edges in a directed graph.

Graph Type: The graph type relates to the structure of the graph and is divided into two categories: Simple and Multi-Graph.

Simple graphs contain at most one edge between any two vertices. If we have an attributed graph representation, a multigraph can be supported by representing the multiple edges between two vertices as a single edge, where each multi-edge corresponds to an attribute of that edge.

Multi-Graphs allow for multiple edges between any two vertices. Typically, this type of graph representation takes up more space than simple graph representations.

Dynamicity: This feature establishes if the representation allows the graph to be updated. It is divided into two categories: (Dynamic) or (Static).

Dynamic representations typically are not as space efficient as their static counterparts since more information is necessary to obtain efficient structures that can be updated without having to perform a full reconstruction of the representation, as it happens with static representations.

Static representations are more common than dynamic representations and require the entire representation to be re-constructed when any of its contents need to be changed.

2.3.1 Relevant Research Works

In this section we present the most relevant Optimized Graph Representations. Table 2.3 shows the most relevant Optimized Graph Representations identified in our research, classified according to the previously described taxonomy, highlighting their distinctive features.

Work	Approach	Attributes	Edges	Graph Type	Dynamicity
k^2-tree [22]	Adj. Matrix	Non-Attributed	D	Simple	Static
Attk^2-tree [39]	Adj. Matrix	Attributed	D	Multi-Graph	Static*
dk^2-tree [40]	Adj. Matrix	Non-Attributed	D	Simple	Dynamic
Dynamic Trie [41]	Adj. Matrix	Non-Attributed	D	Simple	Dynamic
DEX [23]	Bitmap-based	Attributed	D	Multi-Graph	Dynamic
CSR [24]	Adjacency List	Non-Attributed	D	Simple	Static
PCSR [24]	Adjacency List	Non-Attributed	D	Simple	Dynamic
GraphZIP [42]	Summarization	Non-Attributed	U	Multi-Graph	Static
GSSC [43]	Summarization	Attributed	U	Multi-Graph	Static

* the work describes both a static and dynamic version

Table 2.3: Optimized Graph Representation Classification.

k^2-tree: A compact graph representation that takes advantage of sparse adjacency matrices (Figure 2.4). Proposed by Brisaboa et al. [22] the tree represents the structure of the graph adjacency matrix, where each node in the tree is represented by a single bit: 1 for internal nodes and 0 for leaf nodes, except in the last level where all nodes are leaves and represent the bit values in the adjacency matrix.

The matrix is divided into k^2 submatrices of the same size following an MX-Quadtree strategy, each of these submatrices representing a child node of the root node in the tree. Each submatrix has a value of 1 if it contains at least one 1 bit cell or a value of 0 otherwise, representing a leaf node. The adjacency matrix must be a squared matrix $n \times n$ where n is a power of k, otherwise the matrix is extended to the right and bottom with 0s, making it of width $n' = k^{\lceil \log_k n \rceil}$.

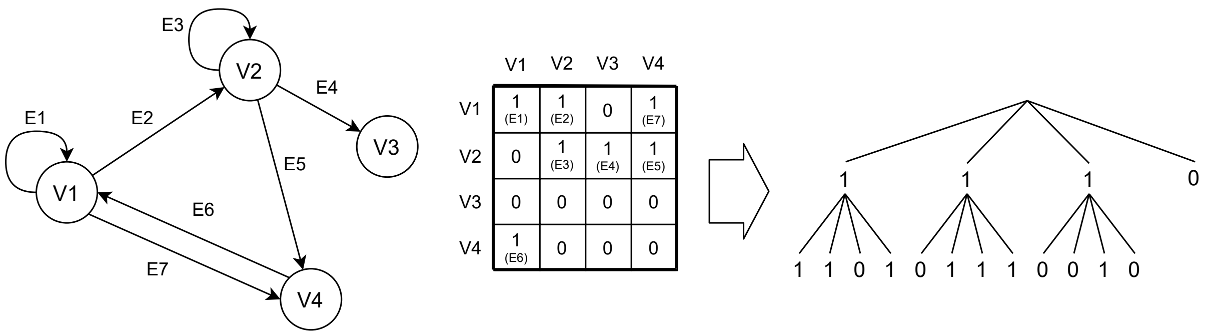


Figure 2.4: Graph and corresponding adjacency matrix and k^2-tree representation.

A key operation in this representation is the $rank(T, i)$ operation, which gives the number of bits with value 1 in sequence $T[1, i]$. Another useful operation is $child_i(x)$ (for $0 \leq i < k^2$) that gives the position of the i-th child of node x. To obtain the direct(reverse) neighbors of a vertex we need to find all bits with value 1 in $row_{i*}(column_{*j})$, where $i(j)$ is the identifier of the vertex. This can be obtained by performing a top-down traversal tree traversal that chooses k out of k^2 of a single node and collects all leaf nodes with a value of 1, which can be implemented using the $rank(T, i)$ and the $child_i(x)$ operations.

Attk^2-tree: A structure proposed by Álvarez-García, Sandra, et al. [39], the Attributed k^2-tree (AttK^2-tree) is a representation based on the k^2-tree and is capable of efficiently storing and processing attributed multi-graphs. The graphs also support labels, where each node and edge can have at most one label that represents the type of attributes that a component may have. The structure represents attributed graphs using three components.

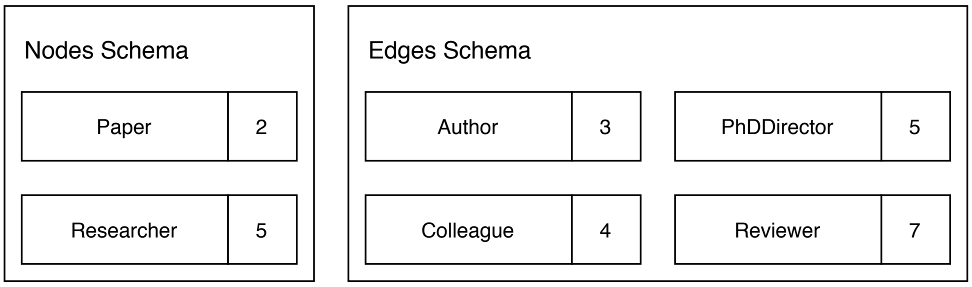
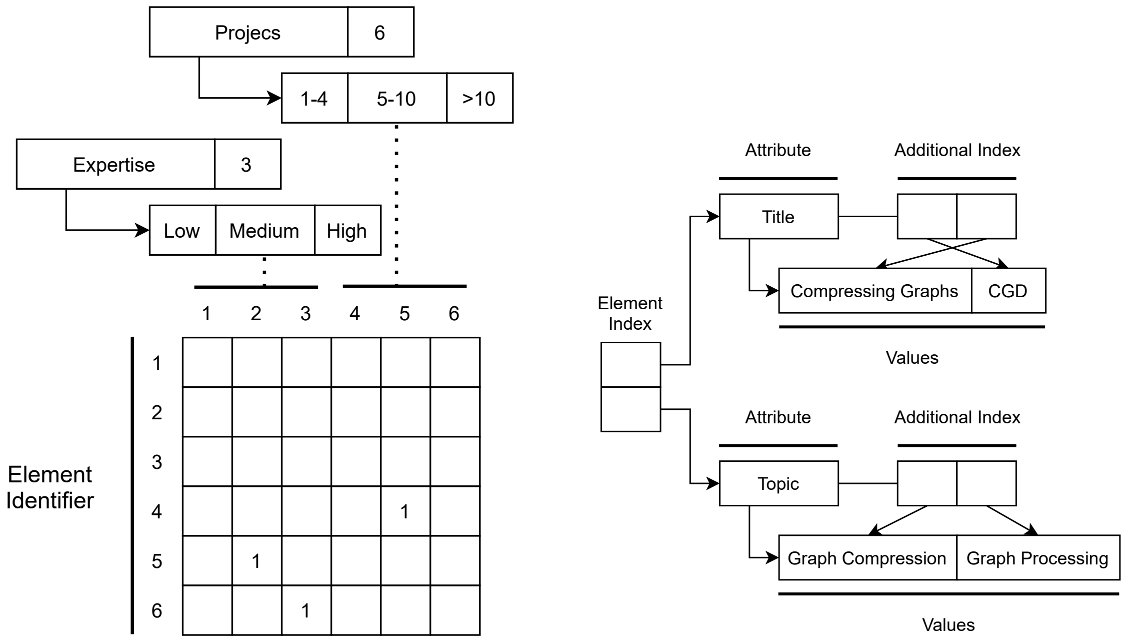


Figure 2.5: Schema component of the AttK^2-tree

The first component is the *Schema* component (Figure 2.5). The schema keeps track of all valid node/edge labels. The labels are ordered lexicographically and the identifiers are assigned sequentially to the node/edges of each label.



(a) Dense attributes **(b)** Sparse attributes

Figure 2.6: Data components of the AttK^2-tree

The second component is the *Data* component that stores the attributes of both nodes and edges, according to their given type (Figure 2.6). The attributes are represented in two different ways depending on the frequency distribution of its values. For attributes for which their values are used by many nodes and edges, they are referred to as dense attributes and are represented using k^2-trees (Figure 2.6a). For attributes for which the nodes (edges) usually take different values, they are referred to as sparse attributes. These attributes are stored as a list indexed by the element identifier (Figure 2.6b).

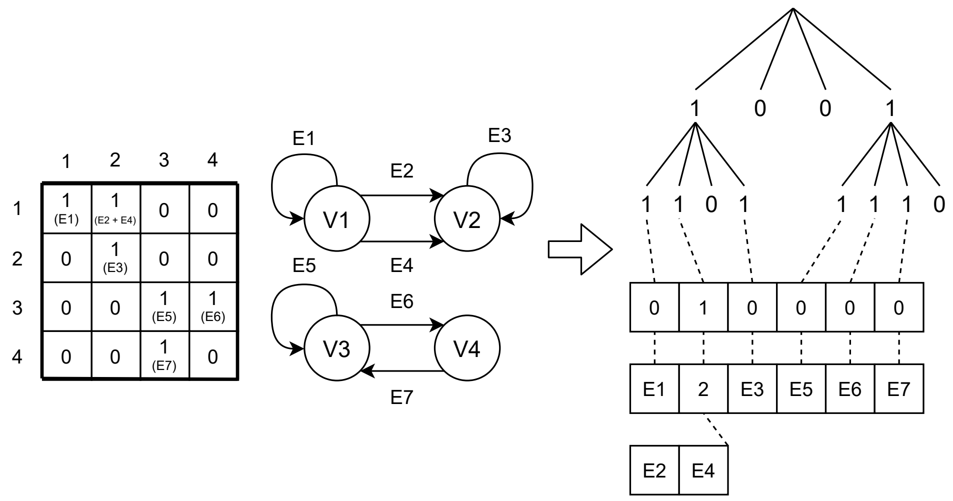


Figure 2.7: Relations component of the AttK^2-tree

The third and final component is the *Relations* component where the different edges that connect the nodes are stored using a k^2-tree (Figure 2.7). To support multiple edges between two nodes a slightly modified version of the k^2-tree is presented, called a *multi-edge k^2-tree*. The described structure is static and requires the whole graph to be rebuilt when changes occur, however, the paper also describes a dynamic version of the AttK^2-tree.

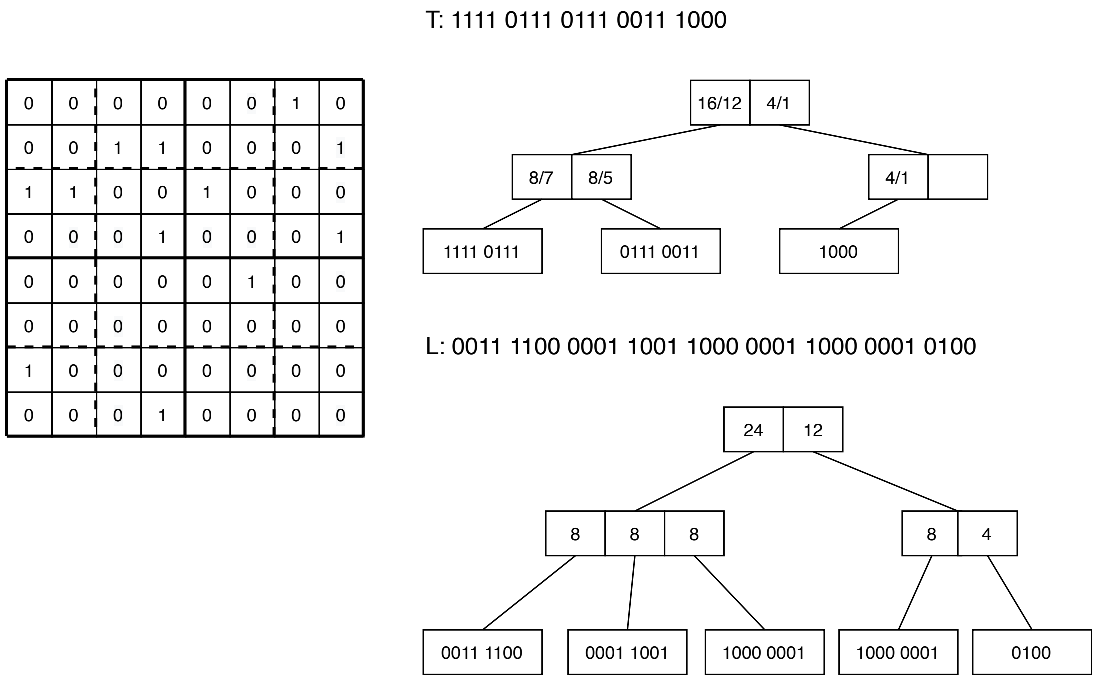
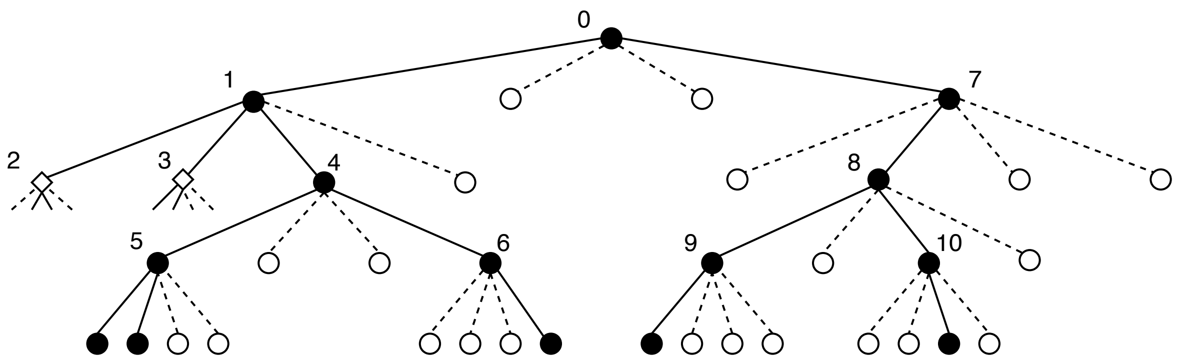


Figure 2.8: dk^2-tree representation and the corresponding T_{tree} and L_{tree}.

dk^2-tree: Proposed by N.R. Brisaboa et al. [40], this structure is a dynamic variation of the k^2-tree that allows to add and remove vertices and edges (Figure 2.8). The static version of the k^2-tree is implemented using two bitmaps, *T* and *L*, in this dynamic version, named dk^2-tree, these bitmaps are replaced with practical implementations of dynamic bitmaps using two trees, T_{tree} and L_{tree}.

The leaves of T_{tree} and L_{tree} contain the bits in *T* and *L*, while the internal nodes provide access to arbitrary positions and act as a dynamic rank structure. Each internal node contains a set of entries that allow access to the leaves for query and update operations.

Each entry in T_{tree} is of the form (*b, o, P*) where *b* and *o* are counters and *P* is a pointer to the corresponding child node the entry is referring to. If *P* points to a leaf node, the *b* counter will have the number of bits stored in the leaf node and the *o* counter will have the number of bits with value 1. If *P* points to a internal node then *b* and *o* will contain the sum of all the *b*- and *o*-counters in the child node. Internal nodes in the L_{tree} are similar with the exception that the *o*-counter does not exist since there is no need to perform rank operations on the bits of the last level.



	0	1	2	3	4	5	6	7	8	9	10
T_B:	1001	1110	0110	1100	1001	1100	0001	0100	1010	1000	0010

Figure 2.9: Example of the Dynamic Trie representation.

The nodes in both of these trees may be partially empty, having a minimum and maximum capacity for the number of entries in each node and containing any number of bits, in the leaf nodes between those two parameters. When the contents of the tree change, the nodes may be split or merged in order to keep the tree completely balanced.

All basic operations performed in a $K^2 - tree$, based on $rank$ and $child$, are also supported by the dynamic version. The relations between existing elements can be created or deleted by removing or adding branches to the conceptual tree representation.

Dynamic Trie: Another dynamic approach that also uses k^2-trees (in fact it works for k^d-trees) was proposed by Arroyuelo, Diego, et al. [41]. This approach has a different approach on the k^d-tree representation, by regarding the tree has a *trie* (prefix tree) on the *Morton* codes [44] of the cells in the adjacency matrix (Figure 2.9).

The *Morton* codes are used to represent the position of each cell in the matrix, so given a cell at position (*r*, *c*) the *Morton* code can be obtained by interlacing the bits of the binary representations of *r* and *c*, for example, given cell (*1*, *1*) the resulting *Morton* code would be *0011*. The trie is then composed of strings of length $log_k n$ over an alphabet of size k^d.

This representation is implemented by dividing the trie into multiple blocks, and each block having child blocks, thus forming a tree of blocks. At any given time, a block can store at most N nodes, if new nodes need to be added then the size of the block is increased.

The main operation needed for traversing the tree structure is the $child(x, i)$ operation, which is similar to the static version of the k^2-tree and yields the child of node x by symbol $0 \geq i < k^2$.

DEX: Martínez-Bazan, Norbert, et al. [23] describe the internals of the DEX graph database that uses an approach different from the ones using k^2-trees. This approach makes use of bitmaps to represent labelled and directed attributed dynamic multi-graphs (Figure 2.10). The values are represented using *value sets* that are implemented using two maps: one maps each vertex (edge) to a value, and the other maps each value to a bitmap where each position corresponds to the identifier of a vertex (edge), if the position *i* has a bit value of 1 then the vertex (edge) with identifier *i* also has the same value that the bitmap corresponds to.

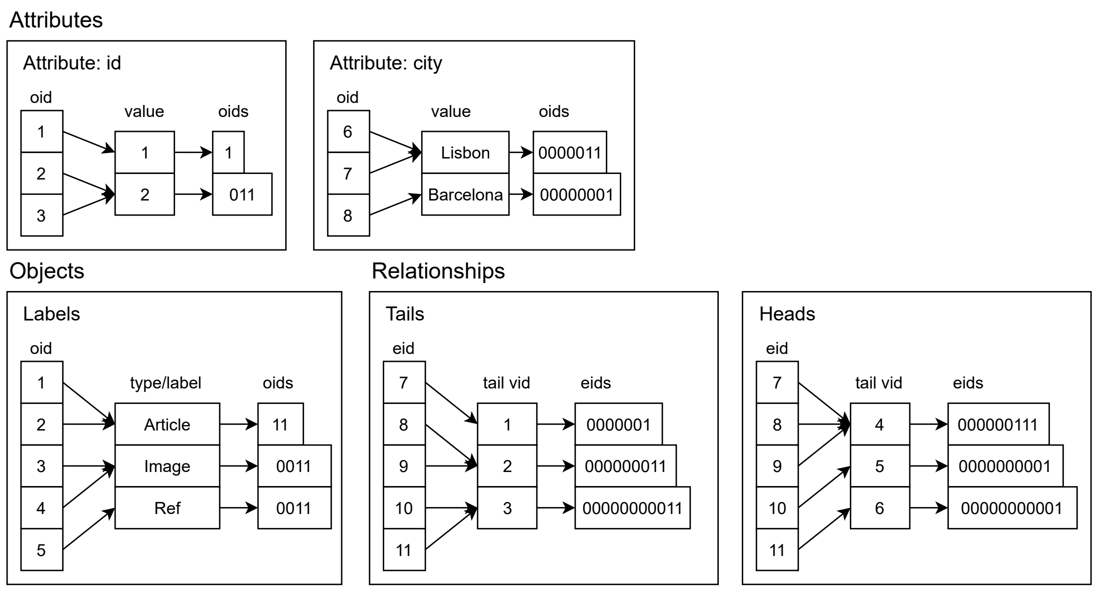


Figure 2.10: Diagram of the internal representation used by the DEX graph database.

The structure used to represent the graph is divided into three groups: **Objects**, where the labels/-types of both vertices and edges are stored; **Relationships**, where the edges are stored in two different *value sets*, one mapping the edge identifiers to the tail vertex identifiers and the other mapping the edge identifiers to the head vertex identifiers; **Attributes** where, for each attribute, a *value set* maps the vertex (edge) identifier to the corresponding attribute.

All operations are implemented by performing logical operations with the stored bitmaps. Since each bitmap encodes the identifier of the object with the relevant value, by performing a union operation, it is possible to obtain all objects with that value, and by performing an intersection between 2 bitmaps it is possible to obtain only objects that share the same value.

Compressed Sparse Row (CSR): CSR is a format used for storing sparse graphs and matrices based on adjacency lists [24]. It efficiently packs all the entries together in arrays, allowing for quick traversal of the data structure (Figure 2.11). This structure uses three arrays to store a graph: a vertex array, where each entry contains the starting index in the edge array where the edges for that vertex are stored in sorted order by destination; an edge array, where the destination vertices of each edge are stored; and a value array only used for weighted graphs. This format only supports static graphs, needing to be

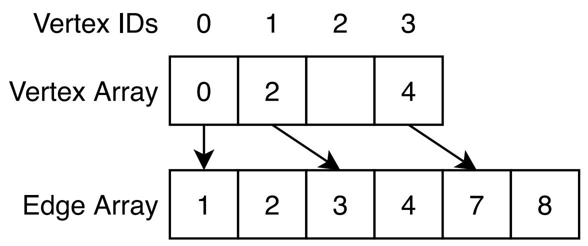


Figure 2.11: Example of the compressed sparse row representation.

completely rebuilt when the graph changes.

To insert a new edge, we must first find the entry in the vertex array corresponding to the new edge source vertex identifier and update the offset stored in each element. In the edge array, all elements after the newly inserted elements are moved one position over. The new elements in the edge array will be inserted in sorted order.

Packed Compressed Sparse Row (PCSR): A dynamic variation of the Compressed Sparse Rows format, Packed Compressed Sparse Row (PCSR) was also proposed by Wheatman, Brian, and Helen Xu [24]. This format efficiently packs all the entries together using Packed Memory Arrays (PMAs) that maintain edges in sorted order and leave spaces between elements to support fast inserts and deletes (Figure 2.12).

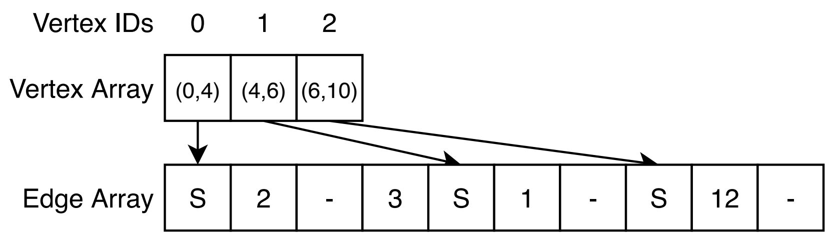


Figure 2.12: Example of the packed compressed sparse row representation.

The PMA keeps an implicit tree with $n/logn$ leaves with each leaf having $logn$ slots for entries in the array, where n is the size of the array. The PCSR uses the same vertex and edge lists as the CSR, with the exception that the edge list is implemented with a PMA instead of an array. Each element in the vertex array stores the start and end pointers to the edge array for its corresponding range. Each nonempty entry in the edge array contains the destination vertex identifier and the edge value.

To add a new vertex, a new entry is added to the end of the vertex array and a sentinel is added to the end of the edge array. The new vertex keeps a pointer to this sentinel. To add a new edge, we start by finding the corresponding vertex in the vertex array, then performing a binary search over the relevant section of the edge array and adding the new edge in sorted order. If a rebalanced is triggered, the sentinel entry is checked and in the case that it was moved, the pointers in the vertex array are updated.

GraphZIP: A clique-based graph compression method that improves the performance of graph algorithms and reduces the space required to store the graph in both memory and disk (Figure 2.13). A clique is a set of vertices any two of which are adjacent.

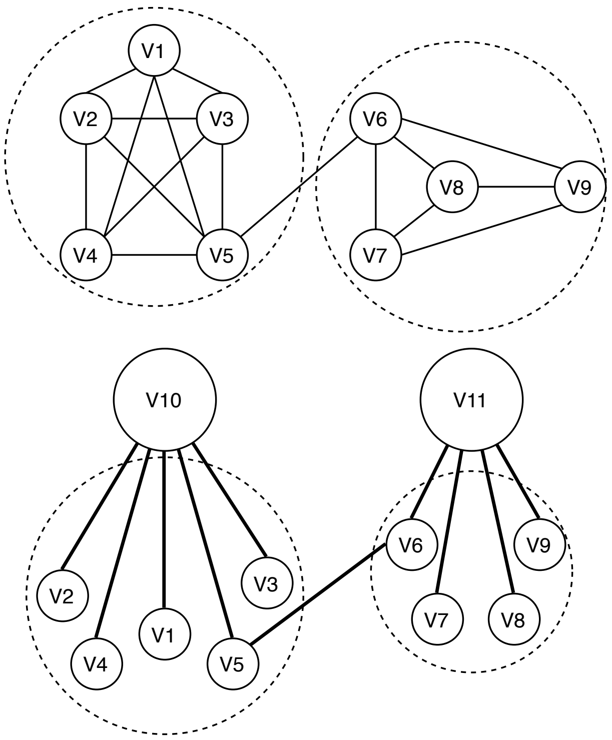


Figure 2.13: Example of the GraphZIP compression method.

This method, proposed by Ryan A. Rossi and Rong Zhou [42], works by using a generalization of the CSR representation that introduces a new type of vertex called *clique vertex* that represents a clique in the graph. This solution works by representing multiple vertices as a single vertex, thus the larger the clique the better the compression.

To obtain the set of cliques an iterative algorithm is used that finds the largest clique at each step. The algorithm can use heuristics, resulting in faster processing but worse compression, or exact clique methods that obtain better compression. The algorithm is executed until all vertices in the graph have been processed or a given threshold has been reached. After the cliques have been computed, a graph encoding method is applied to map the cliques to conceptual vertices.

Graph Summarization based on both Structure and Concepts (GSSC): A compressed representation for attributed simple graphs, proposed by Ashrafi Payaman and Kangavari M. [43], that summarizes the graph based on its structure and attributes. For this reason, a conceptual edge is introduced that connects to vertices with similar attribute values. A new graph is generated from the original graph with these conceptual edges added, as well as weights to each edge. The summarization uses a top-down approach by removing edges with a weight less than a given threshold and partitioning the graph into multiple sub-graphs (Figure 2.14).

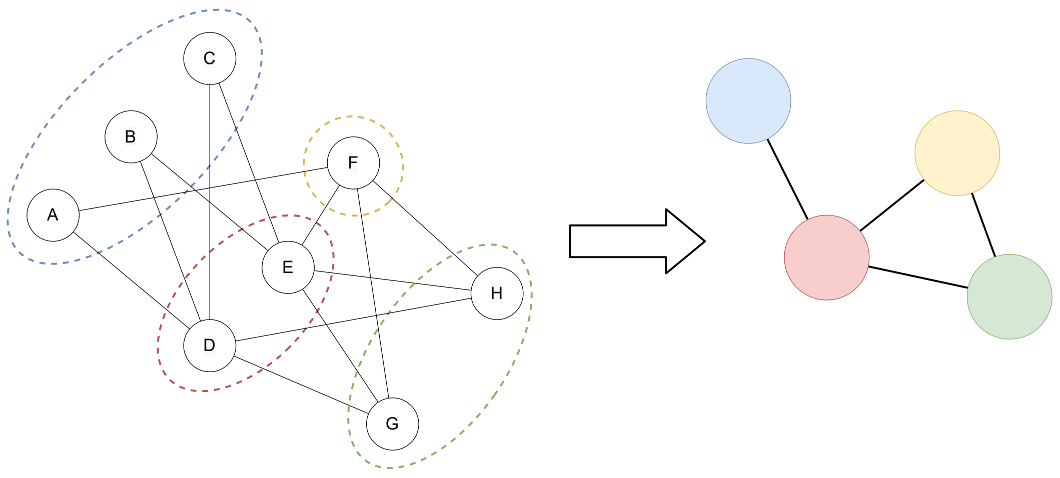


Figure 2.14: Example of the *GSSC* compression method.

Final Remarks

Some more recent work in compressed graph representations includes *g-Sum* by ur Rehman, Saif, et al. [45], a graph summarization approach for large social networks that minimizes the Reconstruction Error (RE) of the representation, allowing for a more accurate summarization and improving its usefulness.

Another recent work, by Jihoon Ko, Yunbum Kook, and Kijung Shin. [46], focuses on incremental lossless graph summarization. This work provides a novel approach in the efficient and lossless summarization of fully dynamic graphs. However, this representation is not suitable for distributed processing systems like *Spark* since the graph would need to be partitioned throughout various executors. Furthermore, the summarization is not intended to allow for the iteration of all edges/vertices of the graph, instead it focuses on the processing of changes to the underlying graph.

2.4 Analysis and Discussion

For the graph database systems described in Section 2.2 we analysed in detail two systems: *Neo4J* and *JanusGraph*.

Neo4J is currently the most popular graph database system, which has the advantage of a much more active development community than other systems. It also offers a much simpler storage system by storing the graph in the file system instead of distributing the graph (although the enterprise edition supports sharding). However, extending this system would require implementing a compressed representation both in disk (binary representation) and in cache (in-memory representation). A compressed representation for a graph database system would also not be very relevant since these systems typically store a large majority of the graph in secondary storage.

Another relevant graph database system is JanusGraph, given that is also a very popular graph database system capable of distributing the graph across various machines. However, the storage backend contains several implementations (*Apache Cassandra*, *Google Cloud Bigtable*, etc) making it much more difficult to extend the system to use a compressed representation.

We decided to extend a distributed graph processing system for the development of our solution. We opted on extending the GraphX system since it is a relevant graph processing system that presents an interesting challenge on the aspect of graph dynamicity. This is due to the underlying dependency on Spark RDDs that are inherently immutable.

For the optimized graph representation, we opted for the k^2-tree representation seeing as it is a relevant compressed data structure that offers great compression for real-world graphs [22] by exploiting sparse matrices, common in web graphs and some social networks.

Another relevant optimized graph representation is the *Attk^2-tree* that supports attributed multigraphs by using k^2-trees. But because of the space overhead introduced by supporting multi-edges between vertices, we opted to use the k^2-tree to represent simple attributed graphs. Simple attributed graphs can still represent multi-edges by making use of the attributes of an edge.

3

Architecture

Contents

3.1 Overview 30

3.2 Graph 32

3.3 Vertices 34

3.4 Edges 36

3.5 Dynamism 40

3.6 Partitioning 40

3.7 Discussion 43

Our solution will extend the GraphX processing system and make use of a k^2-tree implementation to allow for a compressed representation of attributed graphs in main memory. The GraphX system provides an abstraction over a graph, containing a view of vertices, a view of edges and a view of edge triplets, that correspond to the union of an edge with its corresponding source and destination vertices. All views are partitioned according to the user. GraphX implements this abstraction by replicating the vertices in the edge partitions, thus efficiently performing a join between an edge and its corresponding vertices. It is important to note that this abstraction is static and does not allow for new vertices or edges to be added. It is possible to update the attributes of either vertices or edges, but because the underlying Spark RDDs are immutable it presents a challenge to update the graph. Our solution will provide the same three views while maintaining a compressed fully dynamic representation of the graph, capable of adding new edges or vertices as well as updating their attributes.

In this chapter we will present the architecture of our extension to the *GraphX* platform, by providing the specification details of our implemented solution, leaving a more detailed look of our implementation to Chapter 4. In Section 3.1 we give a general overview of our architecture and how it integrates with the *GraphX* system. In Section 3.2 we explain the base graph interface that our architecture integrates with. In Section 3.3 and Section 3.4 we give the specification of the data structures used to store the vertices and edges of the graph, respectively. In Section 3.5 we present the dynamic interface of the graph. In Section 3.6 we highlight some strategies to partition the graph. Finally in Section 3.7 we give our final thoughts on the architecture of our system.

3.1 Overview

Figure 3.1 shows a Unified Modeling Language (UML) diagram of the architecture overview of our system and how it integrates with the *GraphX* platform. The diagram shows in blue the main classes of the *GraphX* implementation and in green the main classes of our system.

The *GraphX* platform offers a very thin abstraction layer composed by the *Graph*, *VertexRDD* and *EdgeRDD* classes, that together provide an interface to represent a distributed graph with a set of operations to iterate and transform it.

The *Graph* class provides an interface for all basic graph operations, primitives used to implement graph algorithms and access to the underlying vertex and edge RDDs.

The *VertexRDD* class provides a vertex specific interface to access an RDD of vertices. Each vertex is represented by a tuple containing its identifier, provided by the user, and an optional generic attribute.

The *EdgeRDD* class, in a similar fashion to the *VertexRDD* class, provides an edge specific interface to access an RDD of edges. Each edge, represented by the *Edge* data class, is a simple data structure containing the identifier of both the source and destination vertices and an optional generic attribute. In addition to the edges, the *Graph* abstraction also exposes a joint view between an edge and its corre-

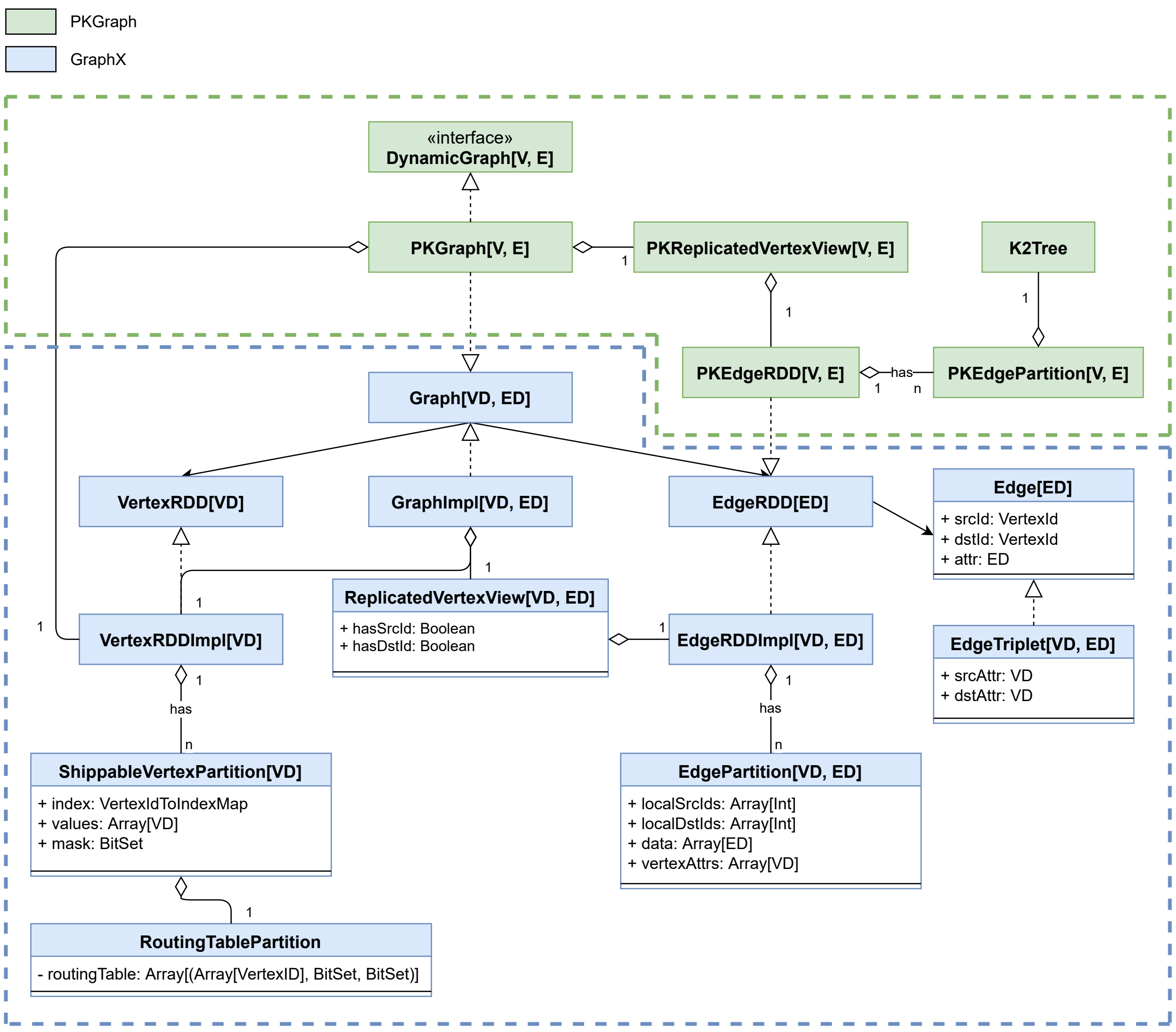


Figure 3.1: Architecture overview of our system.

sponding vertices, by use of the data class *EdgeTriplet*. This triplet view contains the same information as a simple edge, with the addition that the vertex attributes of both the source and destination vertices are also stored.

Both of the RDD abstractions (*VertexRDD* and *EdgeRDD*) are implemented in *GraphX* by using an underlying RDD containing their respective partition types.

In the case of vertices, the *ShippableVertexPartition* class provides an implementation of a vertex partition containing a mapping of each vertex identifier to its corresponding attribute. This partition is also prepared as to be joined together with an edge partition, by being ”shipped” to an *EdgeRDD* for caching. The information necessary to perform these joins, such as to which edge partition to send each vertex to, is stored in the *RoutingTablePartition* class. The vertices are cached in the edge partitions to provide an efficient mechanism to expose the triplet views of a graph.

In the case of edges, these are stored in the *EdgePartition* class, alongside some metadata to keep track of the source and destination identifiers of each edge and their attributes. This class is encased by a small wrapper, implemented by the *ReplicatedVertexView* class, that simply deals with the vertex shipping operations.

The main focus of our system lies in the implementation of an edge partition using the k^2-tree compressed data structure, as proposed by Brisaboa et al. [22] and reusing *GraphX*'s implementation of a *VertexRDD* for the vertices. In the following sections we will explain in more detail the various components of our architecture.

3.2 Graph

As described earlier, the *Graph* class provides the base abstraction for graphs in the *GraphX* system. Our solution extends this base abstraction for usage with our own edge partitions. The key interaction between *PKGraph* and the *Graph* class is detailed in Figure 3.2.

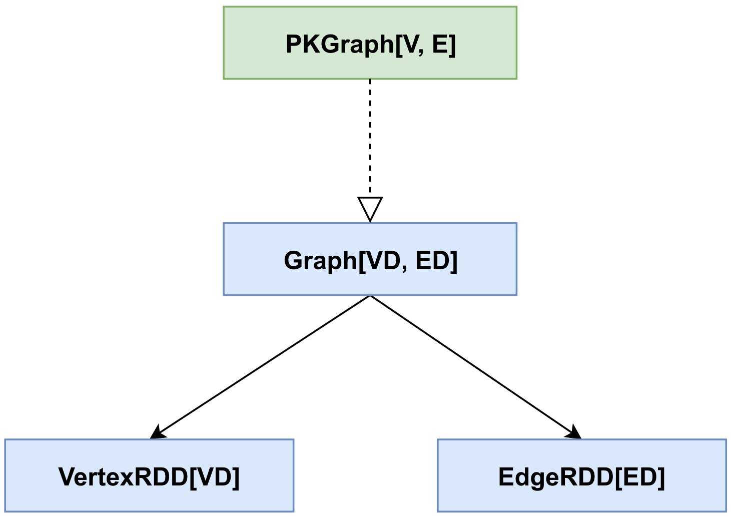


Figure 3.2: Graph component in our architecture

In more detail, we show in Figure 3.3 the interface of the *PKGraph* class, inherited by the *Graph* class, to handle these elements. As stated in the previous section, a graph contains an RDD of both vertices and edges, as well as a joint triplet view that combines the edges with their respective vertex attributes.

The **mapVertices**, **mapEdges** and **mapTriplets** operations apply a user function to each vertex, edge or edge triplet that allows for the creation of a new graph with modified user attributes. All identifiers remain unchanged after these transformations. The **reverse** operation reverses all edges in the graph, by switching the source vertices with the destination vertices. So an edge from A to B would become an edge from B to A.

The **subgraph** operation applies a filter to both the edges and vertices and returns a sub graph of

```scala
class PKGraph[V, E] extends Graph[V, E] {
    val vertices: VertexRDD[V]
    val edges: EdgeRDD[E]
    val triplets: RDD[EdgeTriplet[V, E]]

    def mapVertices[V2](map: (VertexId, V) => V2)

    def mapEdges[E2](map: Edge[E] => E2): Graph[V, E2]

    def mapTriplets[E2](map: EdgeTriplet[V, E] => E2): Graph[V, E2]

    def reverse: Graph[V, E]

    def subgraph(
        epred: EdgeTriplet[V, E] => Boolean,
        vpred: (VertexId, V) => Boolean
    ): Graph[V, E]

    def mask[V2, E2](other: Graph[V2, E2]): Graph[V, E]

    def groupEdges(merge: (E, E) => E): Graph[V, E]

    def partitionBy(
        partitionStrategy: PartitionStrategy,
        numPartitions: Int
    ): Graph[V, E]

    def aggregateMessages[A](
        sendMsg: EdgeContext[V, E, A] => Unit,
        mergeMsg: (A, A) => A,
        tripletFields: TripletFields
    ): VertexRDD[A]
}
```

Figure 3.3: Interface of the PKGraph class

the original graph, containing only vertices and edges that satisfied the predicate. The **mask** operation combines two graphs and keeps only the vertices and edges that exist in both. It is assumed that the provided graph instance is the same implementation has the graph this operation is being applied to.

The **groupEdges** groups any edges between the same two vertices. Because our solution uses a standard k^2-tree, we cannot represent multigraphs, so this operation does nothing. An alternative to using multi-graphs would be to represent each additional edge using the attributes of a single edge.

All these operations are then executed in a lazy and distributed fashion, by propagating them throughout a cluster of computing nodes and aggregating the result in the driver program. Figure 3.4 shows an example of how a graph operation can be distributed throughout a cluster.

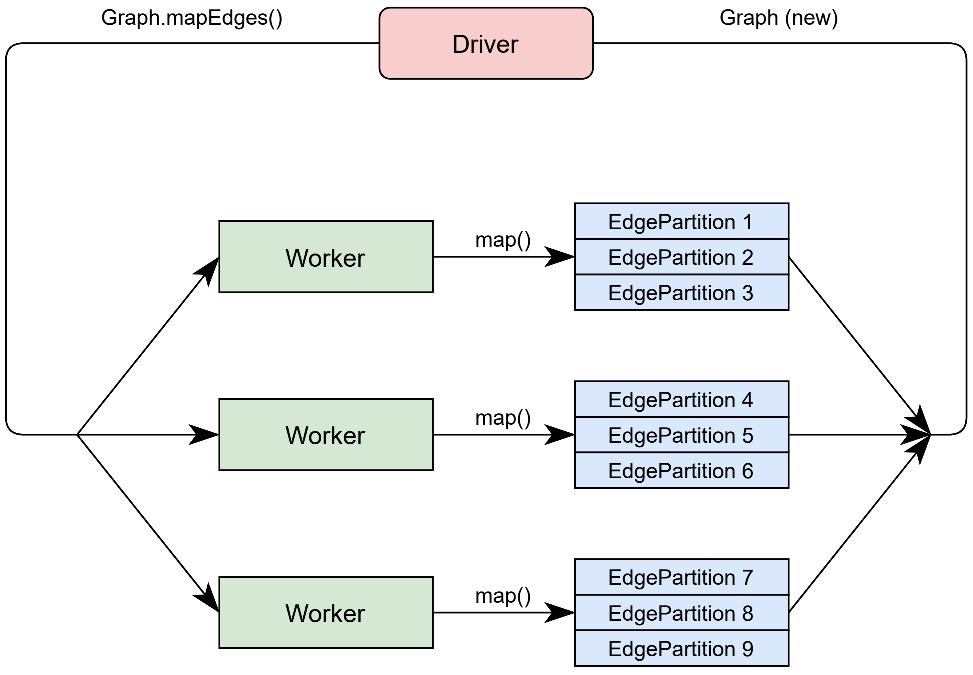


Figure 3.4: Distributed graph work in a cluster

3.3 Vertices

The *VertexRDD* class provides an interface for vertex specific RDDs, containing operations to iterate and transform the underlying vertices of the graph (Figure 3.5).

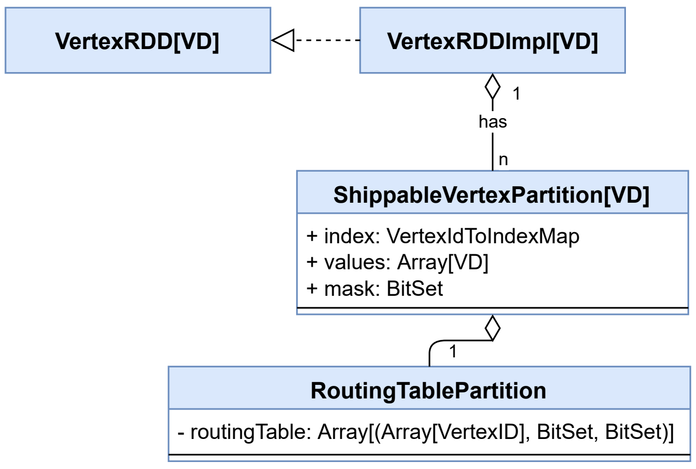

VertexRDD[VD]
VertexRDDImpl[VD]
1
has
n
ShippableVertexPartition[VD]
+ index: VertexIdToIndexMap
+ values: Array[VD]
+ mask: BitSet
1
RoutingTablePartition
- routingTable: Array[(Array[VertexID], BitSet, BitSet)]

Figure 3.5: Vertex components in our architecture

Our solution is focused mainly on compressing the edges of the graph by use of a k^2-tree, therefore the approach for storing the vertices remains unchanged from the *GraphX* system. The *VertexRDDImpl* class, that contains the implementation of a *VertexRDD*, is reused for our solution with some slight changes in its construction, that we will explain in more detail in Chapter 4, to accommodate for our own specific implementation for the edge partitions.

```scala
class ShippableVertexPartition[VD] {
    // Hash set of vertex identifiers
    val index: VertexIdToIndexMap

    // Vertex Attributes
    val values: Array[VD]

    // Mask of active vertices
    val mask: BitSet

    // Routing information of each vertex to its corresponding edge partition
    val routingTable: RoutingTablePartition

    def shipVertexAttributes(
        shipSrc: Boolean,
        shipDst: Boolean
    ): Iterator[(PartitionID, VertexAttributeBlock[VD])]

    def shipVertexIds(): Iterator[(PartitionID, Array[VertexId])]
}
```

Figure 3.6: Interface of the ShippableVertexPartition class

The vertex partitions, where the actual vertices are stored, are implemented by the *ShippableVertexPartition* that keeps them in a format ready to be ”shipped” to their corresponding edge partitions (see Figure 3.6). Each vertex partition keeps track of the routing information for each of its vertices, to later be used to determine to which edge partition to ship them to. The **shipVertexAttributes** function returns an iterator of all vertices keyed by their corresponding edge partition identifier. In a similar fashion, the **shipVertexIds** function returns an iterator of all vertex identifiers keyed by their edge partition. For both operations, only active vertices can be shipped, which are kept in the *mask* bitset. To access the vertices of a partition, we iterate all set bits in the *mask* and retrieve the corresponding vertex identifier and attribute (see Algorithm 3.1).

Algorithm 3.1 Algorithm to iterate the vertices of a given partition

```
procedure ITERATE_VERTICES(partition: ShippableVertexPartition)
    $i \leftarrow partition.mask.nextSetBit()$
    while $i >= 0$ do
        $vertexId \leftarrow partition.index[i]$
        $attr \leftarrow partition.values[i]$
        output $Vertex(vertexId, attr)$
        $i \leftarrow partition.mask.nextSetBit()$
```

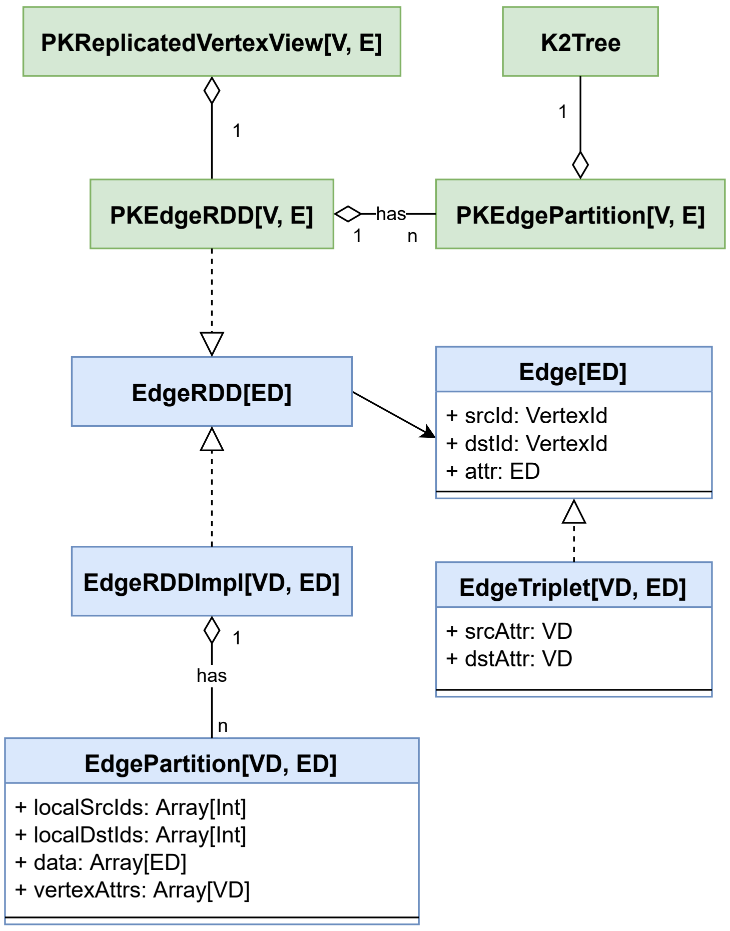


Figure 3.7: Edge components in our architecture

3.4 Edges

The *EdgeRDD* class provides an interface for edge specific RDDs, containing operations to iterate and transform the underlying edges of the graph (Figure 3.7).

Our solution extends this abstraction, by the *PKEdgeRDD* class, and provides a specific implementation of the edge partitions (*PKEdgePartition*) using the compressed data structure k^2-tree to store the edges of the graph (*K2Tree*).

The edge partitions are stored in the *PKEdgePartition* class which provides operations to iterate and transform the underlying edges. The actual edges are stored in the *K2Tree* class, which implements the k^2-tree compressed data structure as proposed by Brisaboa et al. [22].

Figure 3.8 shows the interface of one of our edge partitions. Every operation creates a new partition with copies of the previous data and any modifications applied, since this is the expected behavior when changing the elements of an RDD.

The **updateVertices** operation receives an iterator referencing cached vertices in the partition that should be updated with new attributes. The **reverse** operation reverses all edges in the partition, by

```
class PKEdgePartition[V, E] {
    def updateVertices(iter: Iterator[(VertexId, V)]): PKEdgePartition[V, E]

    def reverse: PKEdgePartition[V, E]

    def map[E2](f: Edge[E] => E2): PKEdgePartition[V, E2]

    def filter(
        epred: EdgeTriplet[V, E] => Boolean,
        vpred: (VertexId, V) => Boolean
    ): PKEdgePartition[V, E]

    def innerJoin[E2, E3](
        other: PKEdgePartition[_, E2]
    )(f: (VertexId, VertexId, E, E2) => E3): PKEdgePartition[V, E3]

    def aggregateMessages[A](
        sendMsg: EdgeContext[V, E, A] => Unit,
        mergeMsg: (A, A) => A,
        tripletFields: TripletFields,
        activeness: EdgeActiveness
    ): Iterator[(VertexId, A)]

    //
    // Dynamic Operations
    //

    def addEdges(edges: Iterator[Edge[E]]): PKEdgePartition[V, E]

    def removeEdges(edges: Iterator[(VertexId, VertexId)]): PKEdgePartition[V, E]
}
```

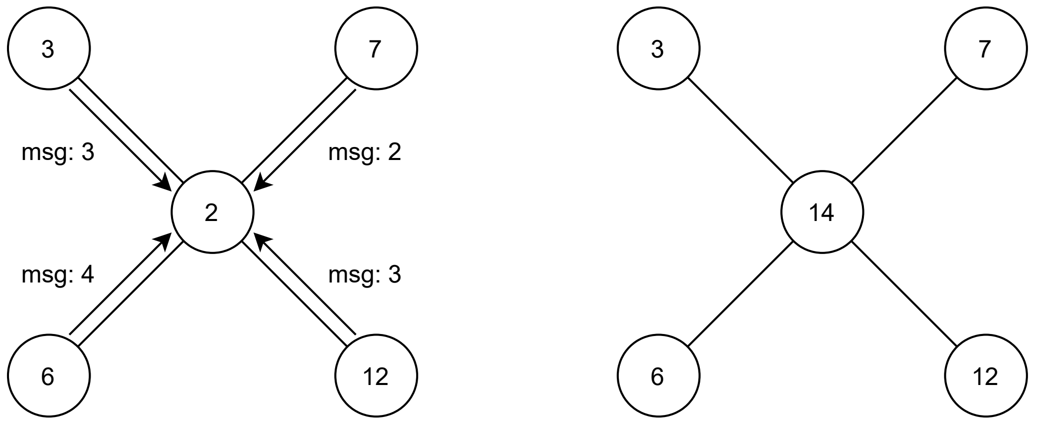
Figure 3.8: Interface of the PKEdgePartition class

switching the source vertices with the destination vertices. This operation is directly used by the graph abstraction to perform its own **reverse** operation.

The **map** operation applies a user function to all edges stored in the partition. The **filter** operation filters both the vertices of an edge and the actual edge according to the user defined predicates. The **innerJoin** operation performs an inner join between two edge partitions. The **aggregateMessages** operation is the primitive used to implement all popular graph algorithms. It implements a **Pregel** [29] like messaging system to exchange messages between the vertices of a graph. Each vertex is capable of ”sending” a message through an edge to another vertex. These messages are then aggregated and merged at each vertex and collected after all messages have been sent.

Figure 3.9 shows an example of exchanging messages between vertices in a graph. In this example, the attributes are merged by simply adding the integer values together.

The *GraphX* computing model also has the ability to only ”activate” some vertices, meaning that only the active vertices would be able to receive messages. Which vertices remain active are stored in each



(a) Messages are sent to the source vertex of each edge **(b)** Messages are aggregated at the central vertex

Figure 3.9: Example of vertices exchanging messages.

edge partition and the non-active vertices are skipped when aggregating messages. The activeness requirements can then be specified as a parameter of the **aggregateMessages** function.

The **addEdges** and **removeEdges** functions are dynamic operations that can add or remove edges from the partition. Although they are dynamic operations, the edge partition does not need to be mutable, since a new instance of the *PKEdgePartition* class is returned as a result of these operations.

As stated previously, the edge partition uses a k^2-tree compress data structure to store the edges of the graph. This data structure is capable of representing the edges of a graph in a very space-efficient format. Our architecture only requires that the implementation of this structure provides a method to access and iterate its edges.

This requires iterating the k^2-tree in a depth-first fashion and calculating the line and column in the adjacency matrix of each edge.

The k^2-tree iterator works by traversing all child nodes of each node, starting from a virtual node at the top of the tree. To find the position of the next node in the bitset the algorithm makes use of the *rank* operation, which counts the number of bits with a value of 1 up to a given position. This process is repeated until a leaf node is reached by making various recursive calls that continuously move down in the adjacency matrix into increasingly smaller quadrants. At the same time, we keep track of the current line and column in the adjacency matrix, returning them when we reach a leaf node.

Each line and column corresponds to a local vertex identifier, which then needs to be efficiently mapped to global identifiers, as well as determining for each edge its corresponding attribute. Algorithm 3.2 shows an example in pseudo code of a possible implementation to access the edges of an edge partition by iterating its corresponding k^2-tree.

In a similar fashion to the *GraphX* system, our solution also uses a simple wrapper over an edge RDD, provided by the *PKReplicatedVertexView*, that handles the shipping of vertices to the underlying edge partitions. Figure 3.10 shows the interface of this class. This class stores the underlying

Algorithm 3.2 Algorithm to iterate the edges of a given partition

```
procedure ITERATE_EDGES(partition: EdgePartition)
    $iterator \leftarrow tree\_iterator(k^h, 0, 0, -1)$        ▷ $k^h$ is the size of the global adjacency matrix
    $i \leftarrow 0$
    while $iterator.hasNext()$ do
        $(localSrcId, localDstId) \leftarrow iterator.next()$
        $srcId \leftarrow partition.local2Global[localSrcId]$
        $dstId \leftarrow partitino.local2Global[localDstId]$
        $attr \leftarrow partition.edgeAttrs[i]$
        output $Edge(srcId, dstId, attr)$
        $i \leftarrow i + 1$

procedure TREE_ITERATOR(size, line, col, pos)
    if $x \geq |T|$ then        ▷ leaf node
        if $L[pos - |T|] = 1$ then output (line, col)
    else        ▷ internal node
        if pos = -1 or T[pos] = 1 then
            $y \leftarrow rank(T, pos) \cdot k^2$        ▷ k$^2$-tree $rank$ operation to find child node
            for $i = 0..k^2 - 1$ do
                $tree\_iterator(size/k, line \cdot (size/k) + i/k, col \cdot (size/k) + i \bmod k, y + i)$
```

```scala
class PKReplicatedVertexView[V, E] {
    var edges: PKEdgeRDD[V, E]
    var hasSrcId: Boolean
    var hasDstId: Boolean

    def reverse(): PKReplicatedVertexView[V, E]

    def upgrade(vertices: VertexRDD[V], includeSrc: Boolean, includeDst: Boolean)

    def updateVertices(updates: VertexRDD[V]): PKReplicatedVertexView[V, E]
}
```

Figure 3.10: Interface of the PKReplicatedVertexView class

PKEdgeRDD instance and keeps track of whether the view includes the attributes of both the source and destination vertices or if these are only partially shipped, since in some cases these may be unnecessary.

The **reverse** function simply reverses all edge partitions in the underlying RDD. The **upgrade** function prepares the given vertices to be ”shipped” to their corresponding edge partitions, which can be deduced by using the routing table stored in the *VertexRDD* class. The **updateVertices** function updates the attributes of the vertices cached in the underlying edge partitions.

3.5 Dynamism

The *DynamicGraph* interface exposes various functions to both add and remove vertices and edges from a graph. However, since the underlying *Spark* RDDs are immutable, some partitions of the graph will need to be rebuilt, or at the very least a new copy of them will need to be made. This does not mean that the entire graph will need to necessarily be rebuilt, only the partitions which we are transforming. Thus, adding or removing both vertices and edges requires determining the partitions affected, and only transforming these. Figure 3.11 shows the interface of the *DynamicGraph* class.

```
class DynamicGraph[V, E] {
    def addVertices(other: RDD[(VertexId, V)]): Graph[V, E]

    def addEdges(other: RDD[Edge[E]]): Graph[V, E]

    def removeVertices(other: RDD[VertexId]): Graph[V, E]

    def removeEdges(other: RDD[(VertexId, VertexId)]): Graph[V, E]
}
```

Figure 3.11: Interface of the DynamicGraph class

The **addVertices** and **addEdges** functions add new vertices and edges, respectively, to the graph, returning a new graph instance in the process.

The **removeVertices** and **removeEdges** functions remove the given vertices and edges from the graph, also returning a new graph instance in the process. Both of these functions work very similarly to applying a filter over the graph, with the slight optimization that only either the vertices or the edges of a graph are affected, instead of always having to filter both.

3.6 Partitioning

Because *GraphX* processes the graph data in a distributed fashion, our solution also needs to address the problem of how to partition the graph to allow for spatial and computational efficiency.

The input graph is represented by two RDDs provided by the user, one representing the vertices and another representing the edges (similar to the GraphX implementation). For the case of edges, our solution interprets them as an edge adjacency matrix that is partitioned using a 2D partitioning scheme [47] that splits the adjacency matrix into several submatrices of equal size, each assigned to a unique partition (see Fig. 3.12). In case the number of partitions is not a perfect square the last column will have a different number of rows than the others.

One problem with this distribution is that it leads to poor work balance since, given a sparse adjacency matrix, some partitions will have many more edges than others. To overcome this, we shuffle the vertex

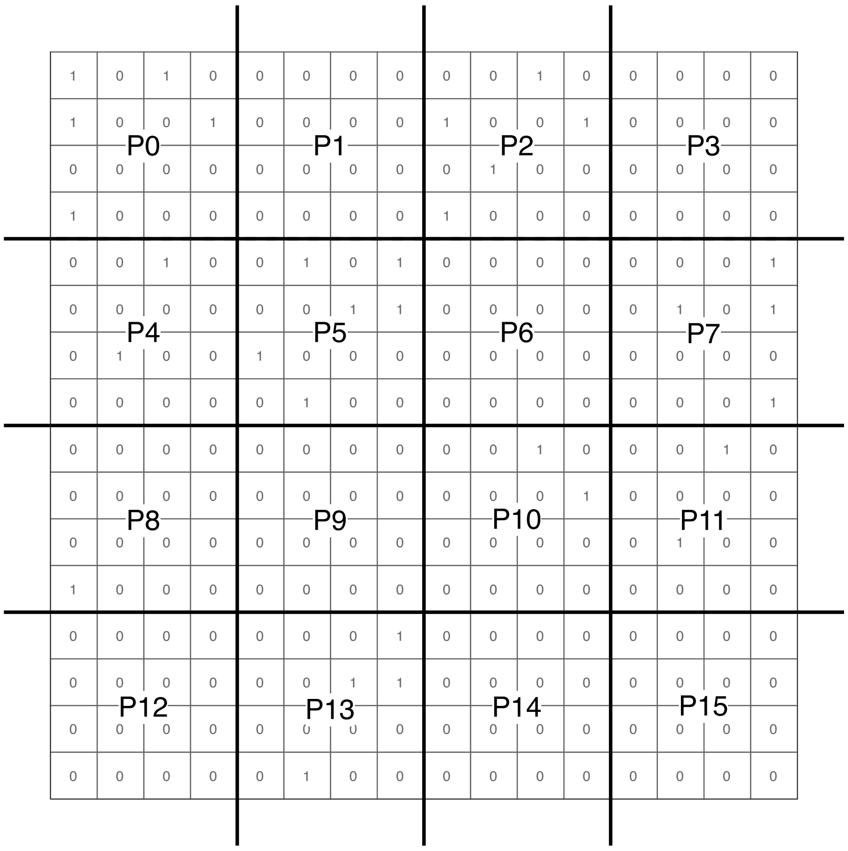


Figure 3.12: Adjacency matrix partitioning scheme

locations in order to evenly distribute them through all partitions.

Like *GraphX*'s implementation, our solution also replicates the vertices in the edge partitions to provide an efficient way to join the edges with their respective vertices. Using this distribution we guarantee that any vertex is replicated at most $2 \times \sqrt{|P|}$, where $|P|$ is the number of partitions to partition the adjacency matrix by, since any vertex is represented by a line and a corresponding column in the matrix, and every line and column intersect at most $\sqrt{|P|}$ partitions.

The described partitioning scheme is applied by default, with no configuration required to the edges. It is also possible for the programmer to specify a different partitioning scheme by using the already existing API provided by *Spark*. For the vertices, we would default to the partitioning scheme supplied by the user or, if no scheme was provided, default to a uniform partitioning strategy such as the one based on the hash of each vertex.

In cases where the graph becomes unbalanced, the user can repartition the underlying vertex and edge RDDs to either increase or decrease the number of partitions, using *Spark*'s **re-partition** function. When increasing the number of partitions these are shuffled, which incurs a significant overhead due to network communication between workers. However, when decreasing the number of partitions it is possible to avoid a shuffling phase by using *Spark*'s **coalesce** function.

The *PartitionStrategy* class provides an interface to implement all existing partition strategies in the *GraphX* platform (see Figure 3.13).

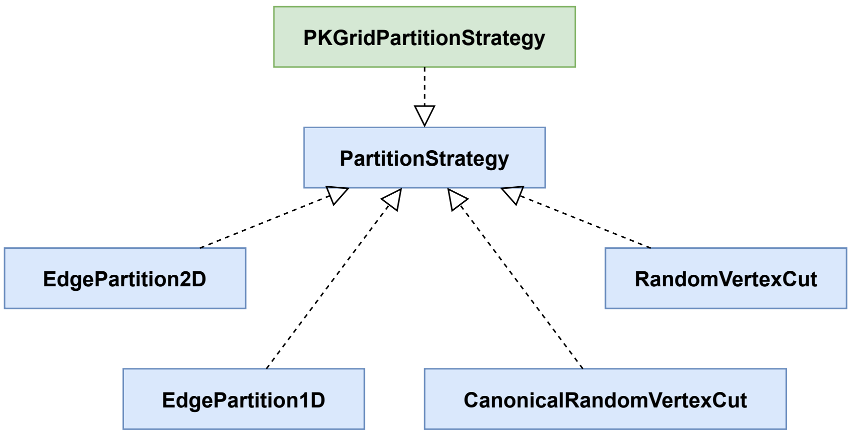


Figure 3.13: Overview of the existing partition strategies

The *PartitionStrategy* class exposes a simple interface to implement a partition strategy to use to partition a graph (see Figure 3.14).

```
class PartitionStrategy {
    def getPartition(
        src: VertexId,
        dst: VertexId,
        numParts: PartitionID
    ): PartitionID
}
```

Figure 3.14: Interface of the PartitionStrategy class

The **getPartition** function returns the partition identifier for an edge with the given source and destination vertex identifiers.

The *GraphX* platform already offers several partition strategies, such as: **EdgePartition2D**, this is the strategy described earlier and implements a strategy that divides the adjacency matrix of the graph into several blocks, as well as shuffling the vertices of the graph to provide a more balanced work distribution; **EdgePartition1D**, groups together edges with the same source vertex; **RandomVertexCut**, distributes the edges based on the hash code of both the source and destination vertex identifiers; **CanonicalRandomVertexCut**, is the same strategy as the **RandomVertexCut** but the direction of the edge is also taken into account when performing the hash.

Our solution also introduces a new partition strategy, represented by the **PKGridPartitionStrategy** class. This strategy is very similar to the **EdgePartition2D** approach that already exists implemented in the *GraphX* platform. The main difference between the strategies is that the vertices won't be shuffled, as to not change the data locality of the edges, thus providing a more space efficient representation of the entire graph in some cases, at the cost of worse workload distribution in the cluster.

3.7 Discussion

Our solution is designed to improve upon *GraphX*'s implementation by using a k^2-tree to efficiently represent binary relations between two vertices, representing an edge. More specifically, *GraphX*'s implementation uses two arrays to store the local source and destination vertex identifiers and a hash map to keep track of all the direct neighbors of each vertex. Our solution replaces all this by a k^2-tree that can efficiently compute the direct and reverse neighbors of any local vertex. *GraphX* does not provide any mechanism to transform the graph by adding new elements while our solution implements a dynamic graph API that allows to add and remove vertices and edges.

4 Implementation

Contents

4.1 K^2-Tree 45

4.2 Graph 49

4.3 Partitions 51

4.4 Analysis 55

Our implementation uses the k^2-tree compressed data structure to store the edges of a graph in the *GraphX* platform. Our implementation was made using *Spark* version 3.1.1 and implemented in the *Scala* programming language version 2.12.10.

In this chapter we present our implementation of the architecture described in the previous chapter. We start by presenting the implementation of the chosen compressed representation in Section 4.1, introducing our implementation of the base graph abstraction in the *GraphX* platform in Section 4.2, explaining the implementation of both the vertex partitions and edge partitions in Section 4.3 and finally analyzing and discussing our implementation in Section 4.4.

4.1 K^2-Tree

As we will see in Section 4.3, the K^2-tree structure is used to store the edges of an edge partition in a space-efficient format. The basic components of a K^2-tree were already described in Sub-Section 2.3.1 when describing the structure as proposed by Brisaboa et al. [22], so in this section we will only focus on the details of our implementation.

The K^2-tree structure is implemented by the *K2Tree* class. Each instance stores some metadata about the structure (such as the *k*-value used and the size of the adjacency matrix the tree represents) and a single bitset to keep all internal and leaf bits, as well as two integers to keep track of the number of internal and leaf bits.

The tree is stored in a bitset by representing each node by a single bit, ordered by each level of the tree (from top to bottom), and ordered from the leftmost node to the rightmost node inside each level (Figure 4.1).

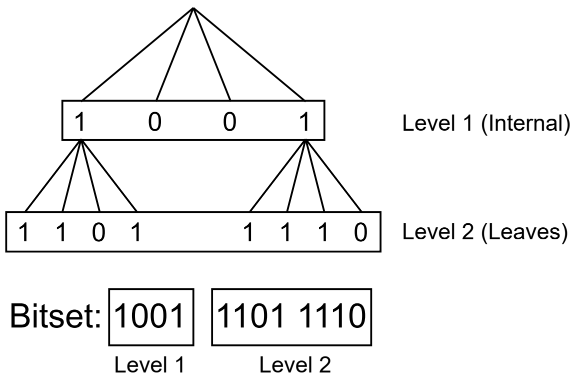


Figure 4.1: Diagram of how a K^2-tree is represented using a bitset.

Instead of separating internal bits and leaf bits into their own bitset structure, these are kept in the same *Bitset* which allows for a slightly more space efficient implementation and only requires the usage of an integer to keep track of where the internal bits end and the leaf bits begin.

4.1.1 Construction

The work of constructing the K^2-tree structure is delegated to the *K2TreeBuilder* class. The structure is built from a ordered list of edges, each edge being a simple tuple containing the line and column of the edge in the adjacency matrix.

The order that the edges are added to the tree must be the same as the order that they are traversed when iterating the tree. To calculate this order an index is assigned to each edge of the adjacency matrix based on the same strategy used to create a K^2-tree from an adjacency matrix. This strategy essentially consists in calculating the row-major index of a quadrant at each level of the tree and then combining them to create a *K2TreeIndex*. These quadrants are obtained by calculating the quadrant that a given edge belongs to at each level of the tree, which can be determined by simply consecutively dividing the current line and column by the *k* value of the tree. Figure 4.2 shows an example of calculating the index of the edge (2, 2) in a 8x8 matrix using a $k = 2$.

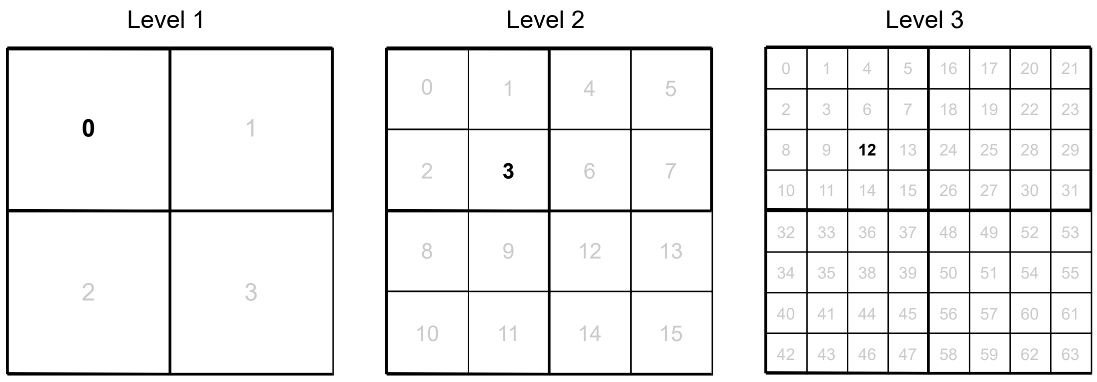
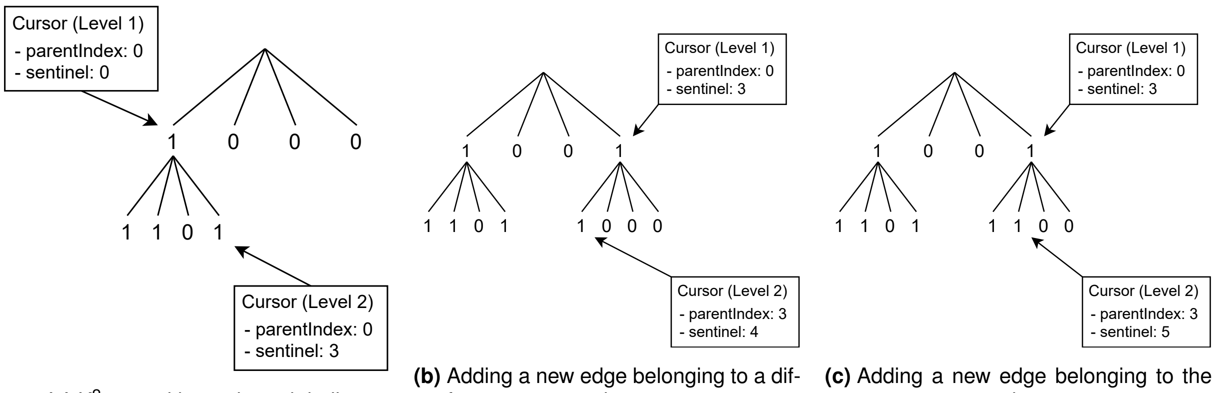


Figure 4.2: Example of calculating the *K2TreeIndex* for an edge.

At each level we calculate the row-major index of the quadrant and multiply it by size of the quadrant. By repeating this process for each level, we reach the final index of the edge. By building this index for each edge we can order the edges to build the tree. This process does not introduce significant overhead since the edges will need to be ordered regardless because of the attached user attributes, as we will see in Section 4.3.

The building algorithm used consists in keeping a cursor at each level of the tree (the height of the tree is calculated at the start by knowing the k value used and the size of the adjacency matrix). Each cursor keeps track of all bits of a given level, the index of the last set bit (**sentinel**) and the index of the its parent node (**parentIndex**), which is used to determine whether a new bit belongs to the same parent as the current node.

For each edge that is added to the builder we traverse the tree from the bottom to the virtual root, updating the cursors at each level as necessary. Because the edges are ordered when inserting, we only need to move the cursor from left to right and keep track of whether the new node we are placing in the tree belongs to the same existing parent node as the sentinel node of that level or whether a new parent node is needed. In case a new parent node is needed, we simply add a new sequence of k^2 bits and place a value of 1 in the correct bit.



(a) K^2-tree with one branch built **(b)** Adding a new edge belonging to a different parent node **(c)** Adding a new edge belonging to the same parent node

Figure 4.3: Example of building a K^2-tree.

Figure 4.3 shows the process of adding new edges to the builder. In Figure 4.3a we start by having a tree with only a single branch built with a cursor at each level pointing to the sentinel bit. In Figure 4.3b we add a new edge that belongs to a new parent node. We start from the bottom of the tree (level 2), check if the new bit belongs to the same parent node as currently defined in the cursor, in this case the bit belongs to the a different parent node so we add a new sequence of k^2 bits and update the sentinel bit. We then repeat this process up to the virtual root. Figure 4.3c highlights one optimization that building the tree from the bottom to the top allows for. In this case the new edge belongs to the same parent node, which means that the entire tree is already built down to that node and it is only necessary to flip the bit at the last level. In web graphs, there can be large clusters of vertices, this method can greatly increase the efficiency of building the K^2-tree when compared to building from the top to bottom.

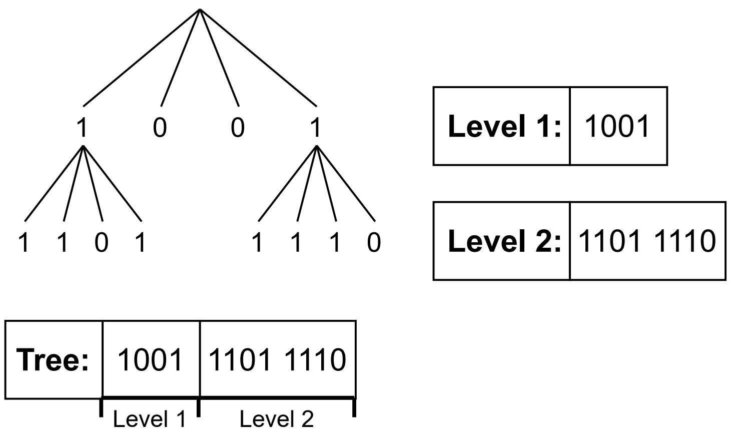


Figure 4.4: Transforming the builder to the final compressed K^2-tree.

After this process is repeated for all edges we are left with the bits for each level in their respective bitsets. To build the final K^2-tree we simply need to copy all bits of each level to a single bitset (Figure 4.4).

It is worth noting that the bitset at each level is allocated with a default initial size and may need to potentially grow as the number of bits added increases. We employed a simple approach that simply checks if the bit we are trying to change is within the capacity of the bitset and if not, double its capacity until it is sufficient.

This approach allows for an efficient performance when building a new K^2-tree while needing very little extra space when compared with the final compressed representation.

An alternative approach that was also considered consisted in creating an initial tree with all nodes represented, meaning each node (independently of whether it had a value of 1 or 0) had k^2 child nodes, and then adding each edge from top to bottom. The final compressed representation would be constructed by iterating all bits and removing sequences of k^2 bits which all had a value of 0.

This approach removes the need to order the edges, but as we will see in Section 4.3 this will still be necessary, and thus slightly improved the time needed to build a new tree but at the cost of much more space overhead when compared to the final compressed representation, making it impossible to use values greater than $k = 14$ due to the exponential growth of the number of bits.

Considering these penalties, our implementation uses the first approach described which requires much less space and still offers very efficient performance when building the tree.

4.1.2 Iteration

The algorithm used to iterate a K^2-Tree is the same as described in Section 3.4 but implemented with an iterative method instead of using recursive calls and using an auxiliary *rank* structure to reduce the number of *rank* calls needed.

The algorithm consists in iterating all the children of a node, from left to right, in a depth-first traversal until the leaf nodes are reached. To accomplish this, a similar approach to the tree construction is taken, by having a cursor at each level of the tree, but now each cursor only keeps the index of the current bit inside the respective level.

As stated in Section 2.3.1 when discussing the K^2-tree, the *rank* operation is crucial to iterate the edges of a tree and is typically used to determine the position of the next child node in the bitset. Our implementation reduces the number of *rank* calls by creating an auxiliary *rank* structure that performs only a single *rank* call for each level and stores them in a map as level offsets. This way, we only need to add the level offset of the current level to determine the position of the current node in the bitset. Then, by keeping track of the current line and column, and updating them as we iterate up and down the tree, we can determine the line and column of an edge when a leaf node is found.

Figure 4.5 shows an example of how the K^2-tree is iterated. The cursor at a given level only moves to the next node when all of its children nodes have been iterated.

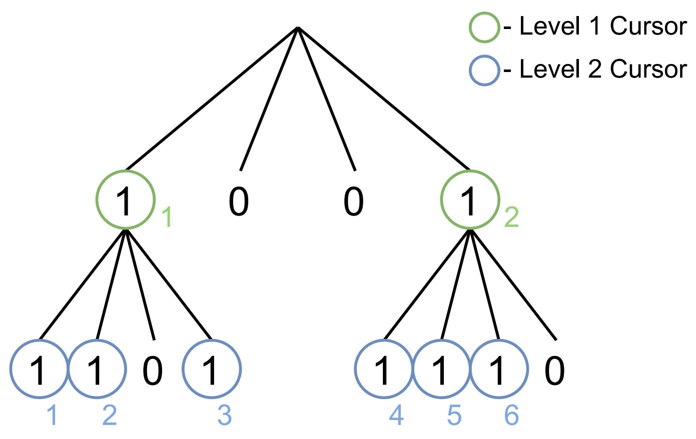


Figure 4.5: Example of iterating a K^2-tree.

Since not all child nodes of a node need to be iterated, only the nodes which have a value of 1, our implementation makes use of a special operation of the bitset, called *nextSetBit*. This operation guarantees a time complexity of $O(1)$ to find the next bit with a value of 1 by counting the number of trailing zeros of a word and making use of masks and bit shift operations. Our tests showed that using this operation over iterating all bits significantly increases the performance of iterating the K^2-tree.

4.2 Graph

The *GraphX* platform uses a property graph, meaning that user attributes can be attached to both the vertices and edges of a graph.

The base class abstraction is provided by the *Graph* class, which contains an interface as described in the previous chapter (see Figure 3.3).

Our implementation of the base graph abstraction is provided by the **PKGraph** class, which contains a similar implementation to *GraphX*'s version, but uses its own type of edge partitions, reusing the same implementation for the vertices as well.

The **mapVertices** and **mapEdges** operations transform their corresponding RDDs by mapping each element with a given user function.

The edge triplet view is obtained by creating a new RDD from joining both the vertex and edge RDDs, mapping for each edge their corresponding vertex attributes. The vertices are kept cached in the edge partitions, updating whenever necessary, and then access to create the resulting joint view.

The **reverse** operation reverses the edges of all edge partitions of the graph. As we will see in the following sections, this requires reconstructing each edge partition.

The **subgraph** operation applies a filter to the underlying vertex RDD, then updates the cached vertices in the affected edge partitions and finally applies a filter over the edges as well. The cached vertices need to be updated first, since the user function receives the edge triplet view.

The **mask** operation performs an inner join between the RDDs of each graph and creates a new graph with the new transformed RDDs.

As explained in the previous chapter, the **groupEdges** operation does nothing since our solution does not support multi-graphs.

4.2.1 Partitioning

The **partitionBy** operation partitions the entire graph according to a *PartitionStrategy*. By default the graph will be partitioned based on the *hash* value of its vertices and edges, but in some cases, it may provide better performance to partition the graph using a specific strategy.

Our solution is capable of using any kind of partitioning strategy, but as described in Chapter 3 a 2D partitioning scheme will be better suited for most web graphs. This strategy will lead to better load distribution and reduce the space requirements of each partition.

The implementation of this strategy already exists in the GraphX platform in the form of the *EdgePartition2D* class. Our solution implements a slightly altered version of this strategy that aims to better preserve the locality of the edges in the adjacency matrix by not shuffling the vertices of the matrix, provided by the **PKGridPartitionStrategy** class. This partitioning strategy is based on a strategy proposed by Brisaboa et al. [47] called *Grid Partitioning*. Our implementation divides the entire adjacency matrix of the graph into various blocks of equal size, each block corresponding to an edge partition, and can potentially lead to better compression of the edges at the cost of a slightly worse load distribution.

4.2.2 Dynamism

The dynamism offered by our implementation consists in rebuilding only the necessary partitions of the graph, more specifically, the ones that will have vertices or edges added/removed. The *DynamicGraph* interface, as described in Chapter 3, contains the methods to add or remove vertices and edges to an existing graph.

The **addVertices** operation adds new vertices to existing partitions. The given RDD can either be already partitioned, in which case the vertices are added to their respective partitions, or may need to be partitioned first, according to the graph vertex partitions, to determine the partition of each vertex.

The **addEdges** operation adds new edges to the existing edge partitions. Similar to the **addVertices** operation the given RDD can already be partitioned according to the graph edge partitions.

The **removeEdges** function filters out all edges with the specified identifiers. This operation can be slightly more efficient than simply applying a filter on the graph since only the edges of the graph are affected and not all partitions need to be traversed. Only the partitions which removed edges belong to need to be processed.

The **removeVertices** operation is similar to the **removeEdges** but removes vertices from the vertex partition and any edges referencing the vertices to be removed from the edge partitions. This means that, in some cases, if most vertices are being reference by existing edges it may be more efficient to remove the edges instead.

For all previously described graph functions, the underlying edge partitions handle the actual logic of transforming the edge partition, so these operations will be described in the more detail in the following section.

4.3 Partitions

The vertex partitions are handled by the *GraphX* implementation of a *VertexRDD*, provided by the *VertexRDDImpl* class. Our solution simply changes how the vertex partitions are built to support our implementation of the edge partitions by creating alternative functions to create a new *VertexRDD*. These functions are very similar to *GraphX*'s version used when building a new *VertexRDD*, with the difference that they use our implementation of an edge partition.

The edge partitions are handled by the *PKEdgeRDD* class, that uses a very similar implementation to that of *GraphX* but using a different implementation for the edge partitions. The main focus of our work was placed in the implementation of these edge partitions, in the *PKEdgePartition* class.

The edge partitions use our implementation of a K^2-tree to store the edges and some metadata about that specific partition. Figure 4.6 shows a snippet of the metadata kept in the *PKEdgePartition* class.

```
class PKEdgePartition[VD, ED](
    val vertexAttrs: Array[VD],
    val global2local: OpenHashMap[VertexID, Int],
    val tree: K2Tree,
    val srcOffset: Long,
    val dstOffset: Long,
    val activeSet: Option[VertexSet]
)
```

Figure 4.6: Metadata of the *PKEdgePartition* class

The **srcOffset** and **dstOffset** fields are used to keep track of the offset of the partition in regard to the global adjacency matrix. These offsets allow the edges to be mapped from global identifiers to local identifiers and vice versa, which in turn allows to create a smaller k^2-tree. Figure 4.7 highlights the difference between using and not using these offsets. If these offsets were not present the resulting partition would become much larger, since it would start at the beginning of the global adjacency matrix.

As an example, with an offset of 5 for both the source and destination vertices, the edge $(5, 6)$ would become the edge $(5 - 5, 6 - 5) = (0, 1)$. The resulting K^2-tree will only represent the section highlighted

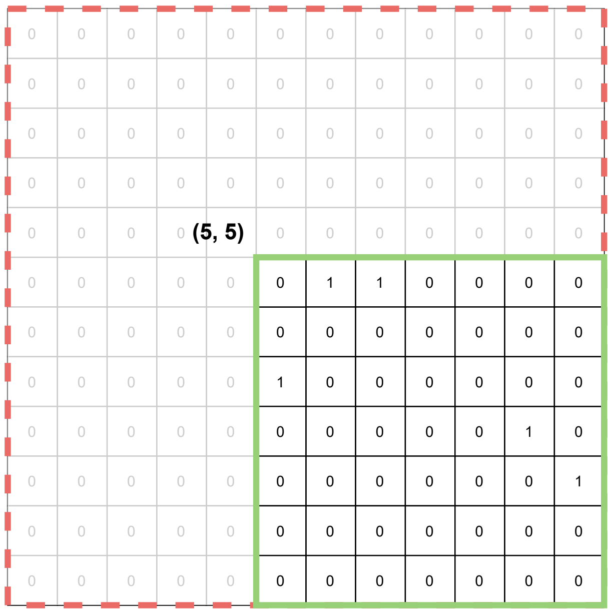


Figure 4.7: Example of a edge partition with an offset from the global adjacency matrix.

in green highlighted in Figure 4.7, leading to a more space efficient tree.

Besides storing the edges, the edge partition will also keep the cached vertices of those edges. The vertices are kept cached in the edge partitions as an efficient way to join the vertices and edges to create the *EdgeTriplet* view. The attributes of the vertices are kept in the **vertexAttrs** array and the **global2local** mapping maps the global identifier of the vertices to sequential local identifiers, this is the same approach used by the GraphX implementation of the edge partitions.

The **activeSet** keeps track of a set of vertices that are active in this partition. The active vertices are taken into account when aggregating messages between vertices in the *aggregateMessages* function. We will describe in more detail how this metadata is used in Section 4.3.2.

4.3.1 Construction

Creating a new edge partition consists in building a new k^2-tree, as described in Section 4.1, and also creating the necessary metadata alongside it.

The class responsible for building a new edge partition is called *PKEdgePartitionBuilder*, this class receives the edges that should be included in the partition, in no particular order, and stores them in a dynamic array. This dynamic array uses an optimized implementation, provided by the Spark platform, by the class *PrimitiveVector*. At the same time, the largest and smallest source and destination vertex identifiers are collected to later calculate the offsets of the partition, as well as the actual size of the partition. Figure 4.8 shows an example of building a new edge partition.

After all edges have been added to the builder these are sorted according to their *K2TreeIndex* and used to create a new k^2-tree. The edge attribute array is also constructed with the attributes ordered

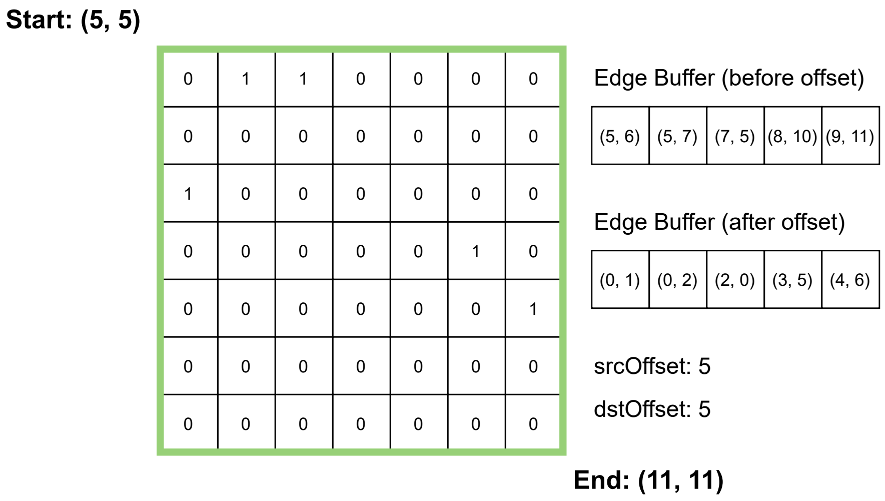


Figure 4.8: Example of building a new edge partition.

by this index. This way, the order that the edges are navigated in the k^2-tree corresponds to their order in the attribute array. The final step is creating a mapping from the global vertex identifiers to their respective local identifiers and building a new edge partition.

In some cases, after applying certain operations, the partition needs to be rebuilt with either modified edge attributes or removed edges. In these cases, to optimize the performance of building a new partition, a different approach is used and the partition is not entirely rebuilt, re-using some existing data and avoiding a full rebuild.

The class *PKExistingPartitionBuilder* is similar to the previous builder but keeps most of the metadata untouched, only needing to rebuild the k^2-tree and the edge attributes. It is assumed that the size of the adjacency matrix represented by the partition does not increase, as this is the case for all implemented operations. The edges are also already inserted in the correct order, meaning that there is no need to sort neither the edges nor their attributes, since this builder is used when iterating an already existing partition that already navigates the edges in the correct order.

4.3.2 Processing

In Chapter 3 we already described the interface of the *PKEdgePartition* class (see Figure 3.8), so in this section we will simply explain in more detail the implementation of each function of the interface.

Every operation creates a new partition by transforming the previous data, typically using a provided user function (e.g map, filter, etc), since this is the expected behavior in the *Spark* platform when transforming an RDD.

The **updateVertices** operation receives an iterator referencing cached vertices in the partition that should be updated with a new attribute. In our implementation we simply make a copy of the array of vertex attributes and iterate the new vertices to update their attributes.

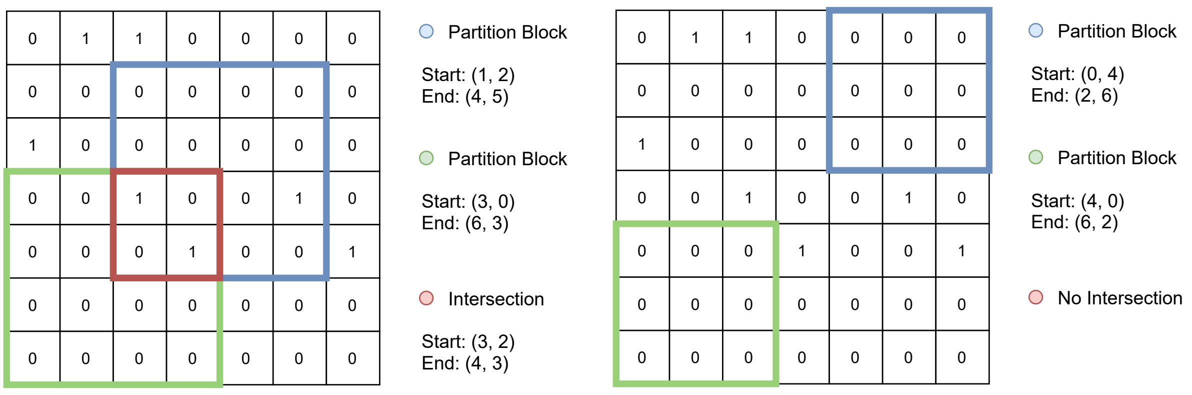
The **reverse** operation reverses all edges in the partition. Because the metadata may suffer some changes in this operation our implementation reconstructs the entire partition by iterating the existing edges and reversing their source and destination vertices.

The **map** operation applies a user function to all edges. In our implementation we navigate the k^2-tree and map the corresponding edge attribute.

The **filter** operation filters both the vertices of an edge and the actual edge according to the user-defined predicates. In this case, our implementation iterates the k^2-tree and, using an existing partition builder, build a new partition containing only the edges that satisfy the predicate.

The **innerJoin** operation performs an inner join between two edge partitions. The algorithm used consists in traversing both partitions at the same time, calculating which edge currently has the smallest index, and advancing its corresponding iterator until either the same edge is found or an edge with a larger index is found.

A small optimization was also implemented that can avoid performing an inner join at all in the case that the partitions have no possible shared edges. By interpreting the partitions as blocks in the global adjacency matrix and calculating their start and end position, we can determine if the partitions intersect, and in case they do not immediately return an empty partition. Figure 4.9 shows an example of detecting if two partitions have any intersection.



(a) Both partitions intersect **(b)** No intersection between both partitions

Figure 4.9: Example of testing the intersection between two partitions.

The **aggregateMessages** operation is the primitive to implement all popular graph algorithms. Our implementation consists on using an aggregating edge context that will store all aggregated messages at each vertex, and then scan all edges in the partition and send messages through all active edges. An edge is active according to its vertices and the *EdgeActiveness* parameter specified. This parameter can determine if both, neither, only the source vertex or only the destination vertex need to be active. A vertex is active if its present in the *activeSet* of the partition, which is supplied by the graph algorithm implementation when necessary. For example, the *Page Rank* algorithm only activates vertices that have received messages in the previous iteration.

Both the dynamic operations (**addEdges** and **removeEdges**) will cause the partition to be completely rebuilt due to the changes in the underlying k^2-tree.

In the case of adding new edges, the existing edges and the new edges are added back to a new partition builder, these are then sorted according to their index and a new partition is built.

In the case of removing existing edges, the edges to remove are placed in a set, so that the lookup for an edge is efficient, then the existing k^2-tree is navigated and all edges that do not exist in the set are added back to the partition. Because the edges are added back in the correct order, there is no need to sort them in this operation.

It is worth noting that all these operations are used when applying some transformation to an existing RDD, that is then used in the creation of a new graph instance, without having to reconstruct the entire graph again. Because transforming the graph typically involves various operations to the underlying RDDs, the graph is cached after being transformed. Meaning that after the partitions of the graph are constructed once, and if no modifications have happened since, none of the graph partitions will need to be rebuilt.

For some cases where this behaviour is undesirable, since all intermediate graphs resulting from these transformations will be cached in memory or secondary storage (user defined), it is also possible to remove the graph from cache by using the existing mechanisms in *Spark* RDDs (*unpersist* method).

4.4 Analysis

Our implementation differs from *GraphX* mainly in the implementation of the edge partitions. *GraphX* uses a simple approach to store the edges by keeping two arrays, one for the source vertices and another for the destination vertices. Inside each array the vertices are sorted according to their local identifier and its possible that there exists repeating vertices inside the arrays. Although this is not a very space efficient solution it allows for very efficient navigation of all edges in the partition, which is typically the case for most operations. To navigate the edges in the *GraphX* implementation, it is simply necessary to iterate both arrays at the same time, extracting the source vertex from one array and the destination vertex from the other. The edge attributes are also stored in an array sorted by the order that the edges are navigated.

Our solution replaces both of these arrays with a single k^2-tree and eliminates the need for most of the metadata required by the *GraphX* implementation, that as we will see in Chapter 5, offers a significantly more space efficient edge partition. However, iterating all edges will naturally be slightly less efficient when compared to the *GraphX* implementation, since our solution will require a depth-first traversal of the k^2-tree, instead of simply iterating an array.

Finally, the *GraphX* implementation is not dynamic and as such does not allow adding new vertices or edges. It can be possible to remove vertices and edges by obtaining a sub-graph of the total graph, but all vertices and edges will need to be removed, which for very large graphs can be less efficient than our approach.

4.4.1 Limitations

One limitation of our implementation is how the *aggregateMessages* primitive is implemented and the lack of optimizations from our solution when compared to what the *GraphX* implementation can perform. In the case of *GraphX*, their implementation offers two possible methods to perform this operation.

The first approach consists in scanning all edges in the partition and performing the exchange of messages for all active edges, this method is called *aggregateMessagesEdgeScan*.

The second approach consists in keeping a mapping of clusters of vertices in the metadata of the edge partition. This cluster mapping assigns a cluster identifier to each vertex, each cluster corresponding to the neighboring vertices of a given vertex. This approach can be effective for graphs with a sparse adjacency matrix or for graphs with a small amount of active vertices, since we can skip an entire cluster of vertices when we detect that the vertex assigned to that cluster is not active. It is worth noting that the *GraphX* implementation only keeps this clustering for the source vertices. Figure 4.10 shows an example of the clusters generated for a given edge partition, each source vertex (left) with neighboring vertices is grouped into a cluster (right).

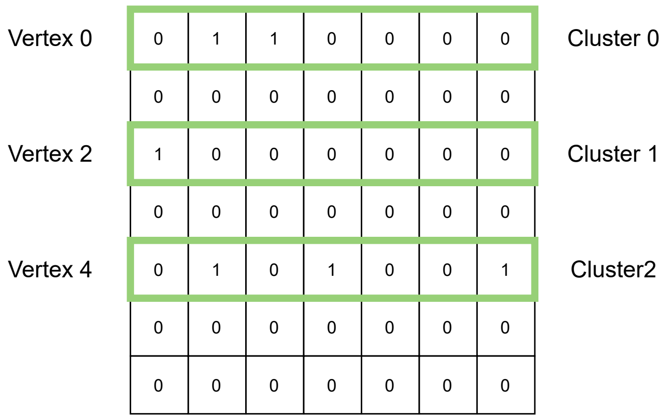
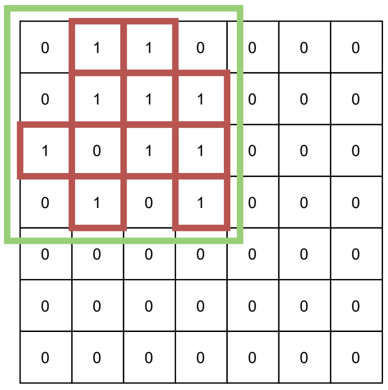


Figure 4.10: Example of building vertex clusters in a edge partition.

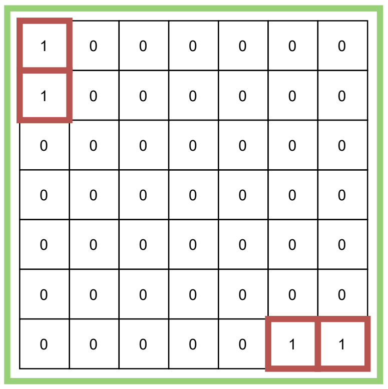
In the case of our implementation, we experimented with a similar approach, since this second approach corresponds almost directly with finding the direct neighbors of a vertex, with a possible advantage of also being able to find the reverse neighbors, all without any need for more metadata. However, our tests showed that this approach was actually less efficient, then simply iterating the entire partition, even in the case of only a small portion of the vertices being active. This can be explained by the fact that finding all direct or reverse neighbors of a vertex consists in having to iterate the entire height of the tree, effectively having similar performance to iterating the entire tree (specially for small values of *k*) for each active vertex. As such, our solution uses only a single approach of scanning all edges in the partition.

Another limitation lies in how the k^2-tree encodes the graph's adjacency matrix. As stated before, each graph as a corresponding adjacency matrix where lines represent source vertices and columns represent destination vertices. In the case of the *GraphX* platform, the line and column number map directly to the vertex identifier, as such, the total size of the adjacency matrix is given by $M = max - min$ where M is the size of the adjacency matrix, max is the largest vertex identifier and min is the smallest vertex identifier. As such, a k^2-tree representing an adjacency matrix of size M can store at most M^2. The resulting k^2-tree will then encode this entire adjacency matrix and efficiently compress larger empty areas.

For adjacency with good data locality, meaning that most edges are clustered, this leads to a k^2-tree with small heights and very good compression results (see Figure 4.11a) since the adjacency matrix can be shrunk to only contain the area with active edges (highlighted in green in Figure 4.11).



(a) All edges exist in a single cluster



(b) No edge clustering

Figure 4.11: Example of data locality of adjacency matrices.

However, for adjacency matrices with very poor data locality, meaning that there exist very little clusters of edges and as such the edges in the adjacency matrix are very far apart, the total number of edges that can be stored ends up being much larger than necessary (see Figure 4.11b) and thus leading to a k^2-tree with an unnecessarily large height, which is the main limiting factor in the processing performance of the structure. This is an inherent limitation of the k^2-tree data structure, since it mainly focuses on web graphs, where this corner case does not typically occur. The data locality of web graphs is also much better suited because of the common occurrence of edge clusters and sparse matrices.

To help alleviate this limitation a partition strategy such as the one described in Section 4.2 could be used. This strategy would lead to partitions with overall better data locality at the cost of a slightly worse distributed workload.

5 Evaluation

Contents

5.1 Metrics 59
5.2 Microbenchmarks 60
5.3 Macrobenchmarks 66
5.4 Optimization 74
5.5 Analysis 77

In this chapter, we detail the metrics and testing environments used to evaluate our implementation and how it fulfills the established requirements. We start by explaining the metrics we will consider in our evaluation in Section 5.1, then in Section 5.2 we explain how the micro-benchmarks were performed and the results obtained in Section 5.3 we present the macro-benchmarks and their results and in Section 5.4 we showcase some possible optimizations and how they compare with the standard implementation.

5.1 Metrics

In this section, we present the evaluation metrics and the requirements of our system.

Our goal is to reduce the graph storage requirements as much as possible without any loss of information, allowing for the graph to be kept in main memory, although distributed throughout a cluster of computing nodes. The impact on the processing time of the graph should be minimal or even non-existing, being acceptable if the compression gains are significant. To study the performance of our implementation, we will analyse the impact on:

Latency	The total time the system takes to execute graph processing jobs.
Resource Usage	To assess the operation of the system and its potential impact on resource utilization introduced by its internal operations.
CPU Overhead	Percentage of the Central Processing Unit (CPU) being used by tasks in the cluster as well as assessing resource waste and cost.
Memory Overhead	Assess the memory overhead introduced by storing the data structures described in our solution.
Throughput	Amount of data the system can process per unit of time.
Cost-Benefit Efficiency	The relation between resource overhead (savings) and performance improvements (penalties).
Scalability	How the system behaves as the volume of data and resources increase.

5.2 Microbenchmarks

In this section we present the micro-benchmarks performed on our implementation and the results obtained.

The micro-benchmarks performed focus on analyzing the latency and memory overhead of the main differentiating aspects of our implementation compared to the GraphX implementation.

More specifically, we focus on comparing the performance of the edge partitions of both implementations, focusing on the memory usage of the internal data structures and the latency of performing transformations on these partitions. These tests are executed outside of the *Spark*/*GraphX* system.

The evaluation compares three different versions of our implementation, each having a different k value to determine its impact on latency and memory usage, with the GraphX implementation. These benchmarks were performed on a machine running Ubuntu 21.04 x64 bit operating system, with a AMD® Ryzen 5 2600 six-core processor with 12 threads per core and 16GB of available RAM.

To perform the micro-benchmarks, we make use of the *Scalameter* library that handles the necessary preparations to execute the tests and collect metrics, such as creating a new Java Virtual Machine instance for each test, handling the garbage collector to avoid collecting inaccurate results and taking multiple samples of each test and obtaining their average value. Each test is invoked in a new Java Virtual Machine containing 2GB of available memory.

We start by first analyzing the memory usage of each implementation, then move to analyzing the latency in building, iterating and performing the aggregate messages operation on an edge partition, and finally analyzing the performance of dynamic operations on our implementation. For each test, we present a bar graph detailing the results obtained as well as its standard deviation.

5.2.1 Memory

These tests will examine the memory overhead of the data structures in various edge partition implementations. Figure 5.1 shows the results of the memory overhead of the edge partition as the number of edges increases.

As the results shows our implementation takes up 33% of the original size of an edge partition (corresponding to a reduction of 66% in memory usage) depending on the number of edges and the sparsity of the corresponding adjacency matrix. For partitions of web graphs, the results could achieve even better compression and thus reduce even more the memory usage.

Besides the number of edges in a partition, the locality of these edges may also impact the memory usage of the edge partition as explained in Section 4.4.

Because our implementation makes use of a k^2-tree, the size of the adjacency matrix corresponding to an edge partition will determine the overall height of the tree. The larger the adjacency matrix the bigger the tree height, leading to more memory overhead and worse performance navigating the tree.

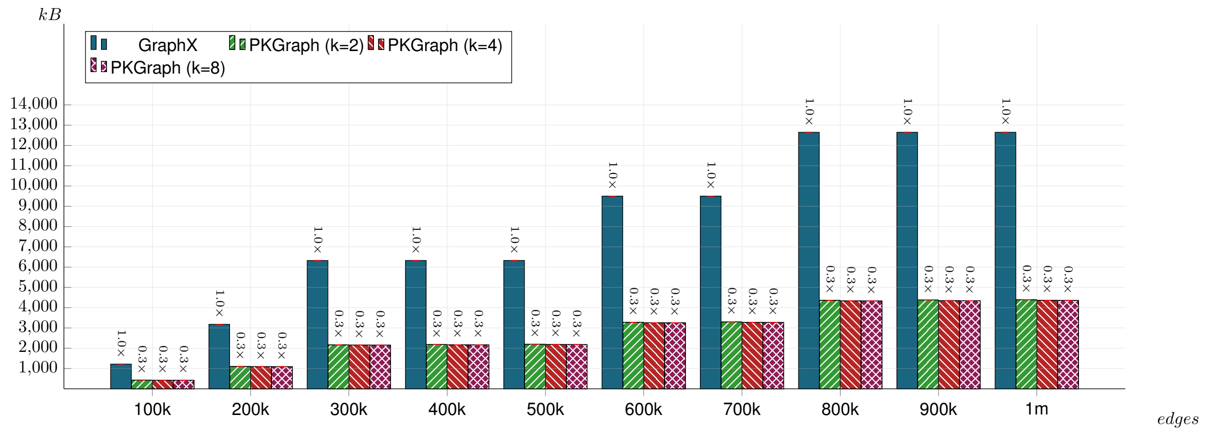


Figure 5.1: Edge partition memory overhead in relation to the number of edges

Figure 5.2 shows the results of the memory usage of a partition storing 1M edges with bad locality.

The memory overhead is given in relation to the density of the adjacency matrix, which always contains a fixed size of 1M edges. A density of 10% corresponds to an adjacency matrix with only 10% of its maximum capacity of edges being used, these edges are then randomly distributed throughout the matrix leading to very poor data locality.

In these tests, the size M of the adjacency matrix is scaled such that the density D of an adjacency matrix is such that $M \times D = 1.000.000$, guaranteeing that the partition is always storing 1M edges but with varying levels of density to simulate partitions that are increasingly more or less sparse. As such, each edge partition takes up roughly 4MB in memory.

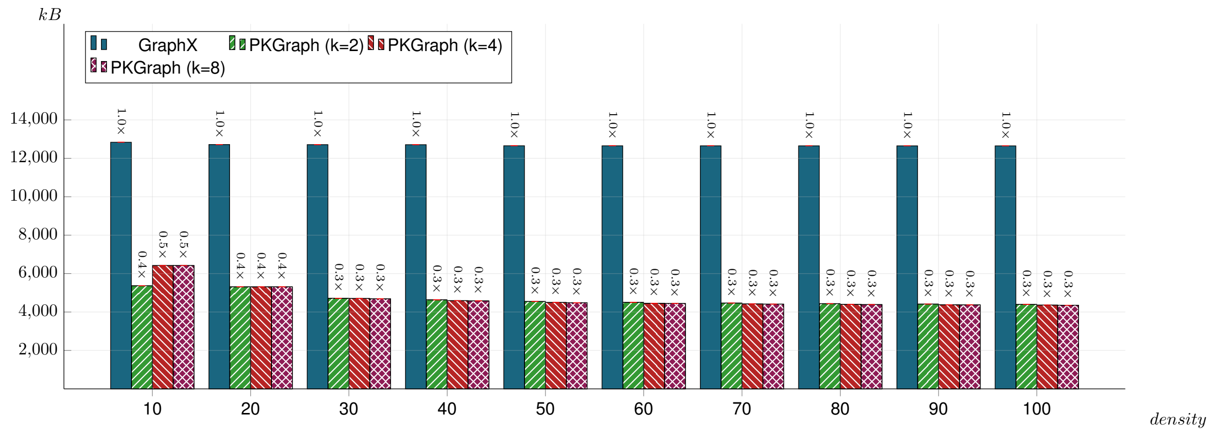


Figure 5.2: Edge partition memory overhead with poor data locality

As the results show, the data locality of the edges can slightly affect the compression of the partition, having an impact of roughly 1.5MB when comparing good to very poor data locality. Moreover, as the density of the adjacency matrix increases and the better the data locality becomes reaching its maximum value when the entire adjacency matrix is being used, the less the memory overhead of the edge partition.

In the case of the GraphX implementation, the density of the matrix is not as relevant since this solution does not make use of any compression, so as expected the memory usage is always the same since the same number of edges are being stored.

We also conclude that our solution works best for sparse graph partitions, meaning that the edges are mostly clustered in a few areas of the adjacency matrix, which is expected as this increases compression potential for the matrix.

5.2.2 Build

These tests compare the latency in building an edge partition from a list of edges. This is used when building the graph for the first time, after which slightly more optimized building techniques are used to build a new edge partition from an existing one. The results are show in Figure 5.3.

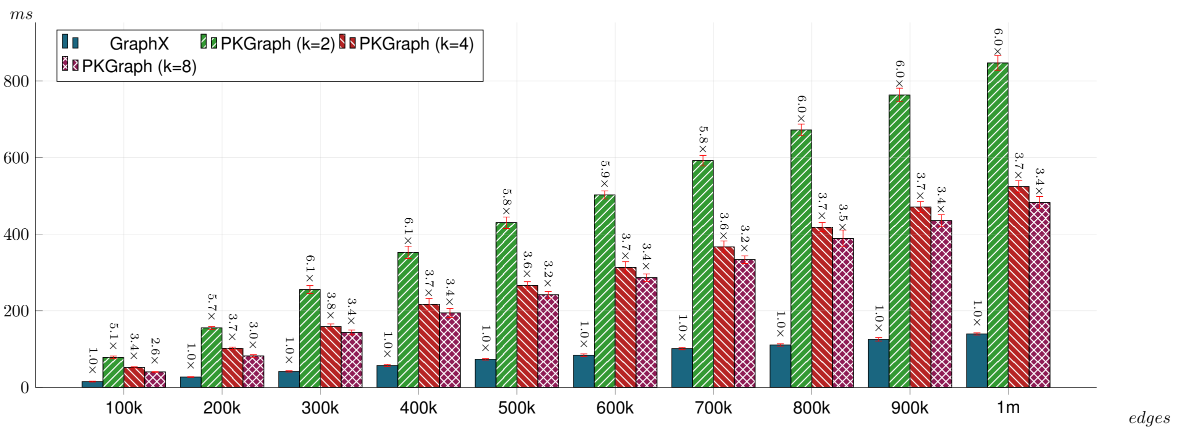


Figure 5.3: Edge partition build latency compared with partition size

The results show how the number of edges in a partition affect the performance of building a new edge partition. As expected, the more edges are in the partition the higher the latency in building that partition.

The results also show that the higher the k value used in the k^2-tree the better the performance in building the edge partition. This is due to the fact that higher values of k result in k^2-trees with smaller overall heights.

However, in all cases the GraphX implementation achieves a better performance than our implementation. This is explained by the time required to build a k^2-tree, which requires navigating the entire height of the tree (worst case). This is in comparison with the GraphX implementation which simply places the vertices of each edge into an array, separating them by source and destination vertices. Again, we note that this operation is seldom executed (just once) for each graph during the whole time it is being processed, updated dynamically, and reprocessed possibly several times.

5.2.3 Iterator

These tests will compare the latency in iterating the entire edge partition. This is the most basic type of operation that exists in the edge partition, being used by most all operations. Figure 5.4 shows the results of the iteration latency as the number of edges increases.

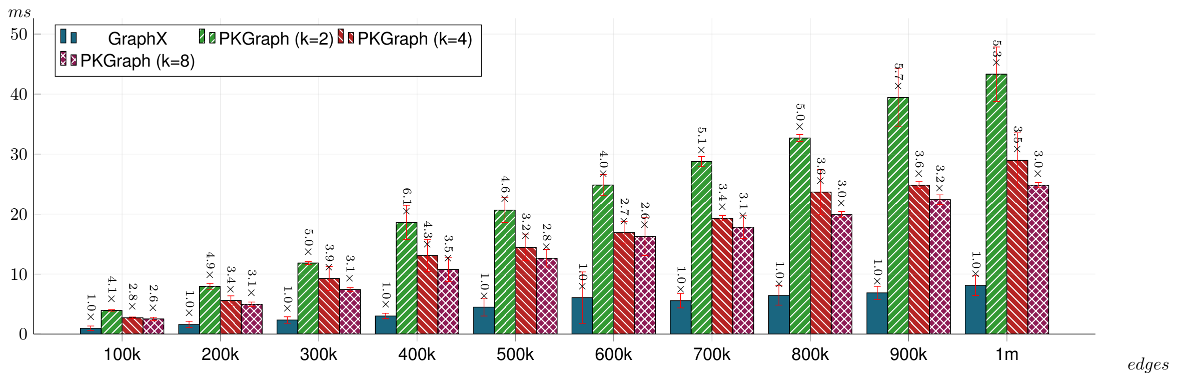


Figure 5.4: Edge partition iteration latency compared with number of edges

The results obtained are very similar to the results observed in the build operation. As expected, the higher the k value of the k^2-tree the better the iterator performance, due to the smaller height of the tree.

The GraphX implementation outperforms our implementation in iteration since the data is stored in an uncompressed format that requires only traversing an array of source vertices and an array of destination vertices to iterate all edges of the partition. This is in comparison to our implementation which requires a depth-first tree traversal. As observed in the previous tests, the height of the k^2-tree continues to be the main factor in the iteration performance, seeing as k^2-trees with smaller values of k have a worse performance.

5.2.4 Aggregate Messages

As stated in Section 4.4, this operation is the primitive used in the implementation of all graph algorithms in the GraphX platform. Our implementation offers only one implementation of this operation, which consists in iterating all edges of the partition, which the GraphX platform designates as edge scan. The GraphX platform also offers the edge scan approach and an additional approach designated as source index, which effectively keeps a mapping of the direct neighbors of each vertex, allowing for a small optimization in some algorithms.

Figure 5.5 shows the results of the latency of the aggregate messages operation as the number of edges increases.

Since this operation consists in essentially iterating the entire edge partition the results obtained are very similar to the results obtained in the iterator tests, and as such, the GraphX implementation outperforms our implementation.

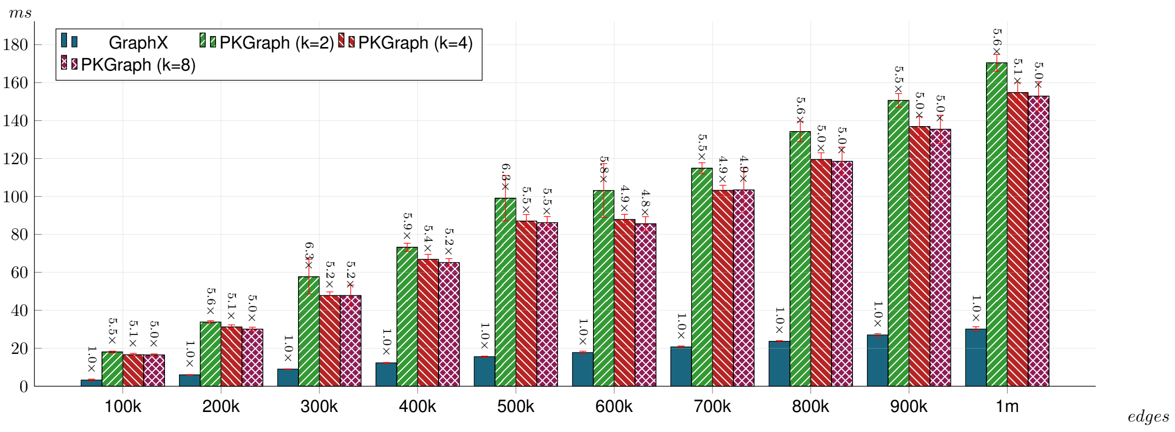


Figure 5.5: Edge partition aggregate messages latency compared with number of edges

As explained before, the GraphX implementation has an optimized method to aggregate messages in an edge partition, which consists in the source index approach. Figure 5.6 shows the results of aggregating messages in an edge partition containing 1M edges. We consider an implementation of a similar optimization for our solution has future work.

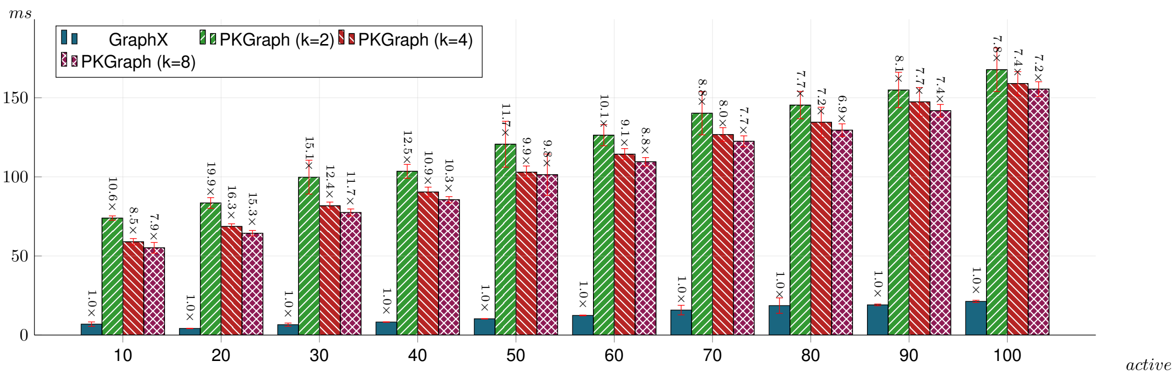


Figure 5.6: Edge partition aggregate messages with source index with varying percentage of active vertices

In these tests, the number of edges in the partition is fixed but the percentage of active vertices varies. This highlights the efficiency of this approach when only a small percentage of the vertices of an edge partition are active.

It is worth noting that not all algorithms can use this optimization, since some algorithms always have all vertices active or in some cases may not send messages to the source vertices.

The results show that the GraphX implementation can achieve even better performance using this approach for edge partitions with a low percentage of active vertices. This optimized approach is typically used whenever the active percentage of vertices in the partition is below 80%. Our implementation achieves a slightly better performance has the number of active vertices decrease, since fewer exchanges of messages between vertices occur.

5.2.5 Dynamism

Since the GraphX implementation does not have any operations to dynamically add or remove edges from an edge partition, these tests will only show the results of our implementation with different values of k.

In fact, we can consider that updating a graph in the original GraphX implementation would imply writing the graph to disk, updating it offline, and reload it from disk in order to re-execute some graph algorithm which becomes painfully slow and resource consuming, hence being one of the the motivations for this work.

For all dynamic tests the edge partition has a fixed size of 1M edges and only the number of edges being added or removed varies. Since our implementation requires rebuilding the entire partition when adding or removing edges, another important factor in the performance of these operations would be the size of the edge partition that is being altered. As showcased before, the more edges the higher the latency in building the partition.

Figure 5.7 shows the results of adding new edges to an existing edge partition and figure 5.8 shows the results of removing edges from an edge partition.

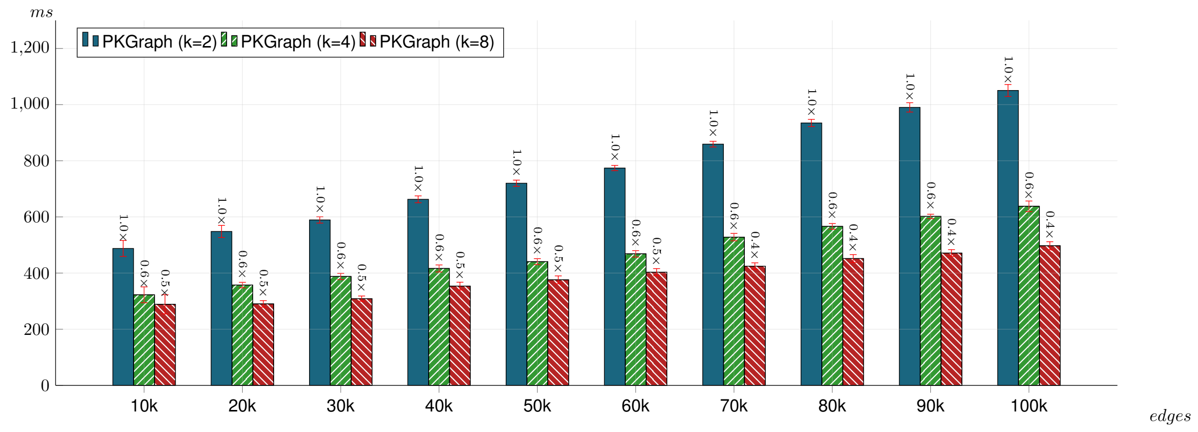


Figure 5.7: Results of adding new edges to an existing edge partition

In the case of adding new edges, as the number of edges to add increase so does the latency values. This result is expected since more edges result in more operations to add new nodes to the k^2-tree being constructed. The results are also consistent with the results obtained in the *Build* tests, since we are recreating the entire k^2-tree again with some new edges added.

In the case of removing existing edges, we are simply excluding existing edges from being added to the new partition. These removed edges are kept in a hash set and each edge is checked for potential removal before being added to the partition. This means that the more edges to remove, the less time it will take to build the new partition but also the more time it will take to build the hash set.

In both cases, the results show that the higher the k value is, the better the performance will be, due to having a smaller tree height.

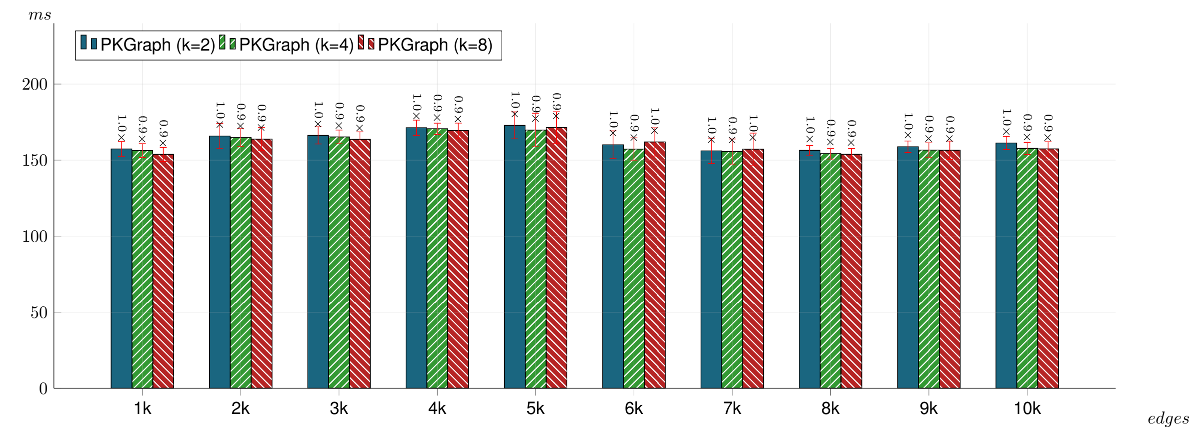


Figure 5.8: Results of removing existing edges from an existing edge partition

5.3 Macrobenchmarks

In this section we will present the macro-benchmarks performed on our implementation and the results obtained. For this evaluation, we will setup a cluster of computing nodes, each node corresponding to a Spark worker that keeps part of the total graph in main memory.

We will submit several graph processing jobs to the cluster, executing some basic graph operations and some of the more popular graph algorithms, using relevant graph datasets and analyse the gains (penalties) our solution has in terms of storage compression and processing.

5.3.1 Setup

The cluster was prepared using the Amazon Web Services (AWS) Elastic Map Reduce (EMR) service, that allows to easily setup a cluster of Spark workers. The cluster uses a single master node and various worker nodes (see Figure 5.9). The actual number of workers used will vary throughout each test. Each machine in the cluster has a 4 core processor with 16 GB of available main memory.

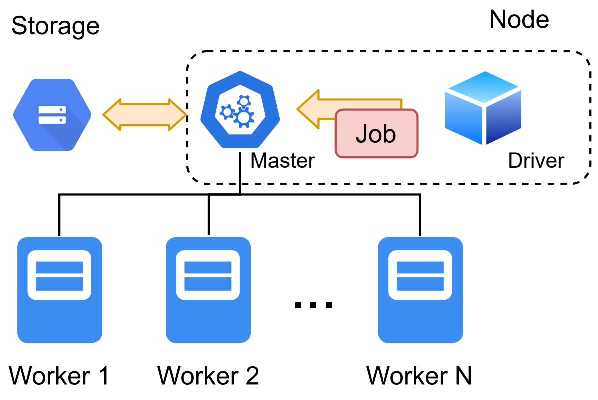


Figure 5.9: Overview of the cluster used to execute spark jobs.

The Spark jobs are submitted from a driver program in a remote machine and the datasets are retrieved from AWS Simple Storage Service (S3) buckets to be used in the jobs executed in the cluster.

5.3.2 Datasets

The datasets used in the evaluation of our implementation are from the Network Repository [48] and the Stanford Large Network Dataset Collection [49]. Table 5.1 presents the datasets used.

The datasets chosen are grouped into two types: i) **Web** graphs describing the references from one web page to another. Typically these types of graphs contain a very high clustering coefficient, making them ideal for compression using our solution; ii) **Social network** graphs to benchmark the use cases when a web graph is not used. In contrast to web graphs, some networks may have a very low clustering coefficient.

Graph Name	Type	$\|V\|^1$	$\|E\|^2$	${k_{avg}}^3$	${d_{avg}}^4$
Youtube Growth	social network	3.2M	12.2M	0.07	7
EU (2005)	web	863K	19M	0.71	43
Indochina (2004)	web	7M	194M	-	26
UK (2002)	web	18M	298M	-	16

1 - number of vertices; 2 - number of edges; 3 - average local clustering coefficient; 4 - average degree

Table 5.1: Table describing the datasets used in our benchmarks.

To evaluate our solution we used various types of workloads, ranging from common graph operations like constructing the graph and iterating it, to popular graph algorithms such as *PageRank* and *Triangle Count*, all of which already contain implementations in the GraphX system. In the following sections, we will present the various workloads used to evaluate our implementation, comparing it to the GraphX implementation, and the results obtained. For all workloads, we will use $k = 8$ for the k^2-tree, since, as we observed in Section 5.2, this value offers the best trade-off between performance and memory usage.

5.3.3 Memory Overhead

Our benchmarks show that the memory overhead of the data structure of the graph remains the same independent of the number of processors. This is due to the fact that the number of partitions used, chosen by Spark based on the size of the file where the dataset was read from, remains the same. Figure 5.10 shows the results of the memory usage of the entire graph with varying datasets.

The results show, as did the micro-benchmark results, that our solution has significantly less memory overhead then the GraphX implementation. Although our previous tests showed a reduction between 60% to 70% when compared to the GraphX implementation, when testing the memory usage of the

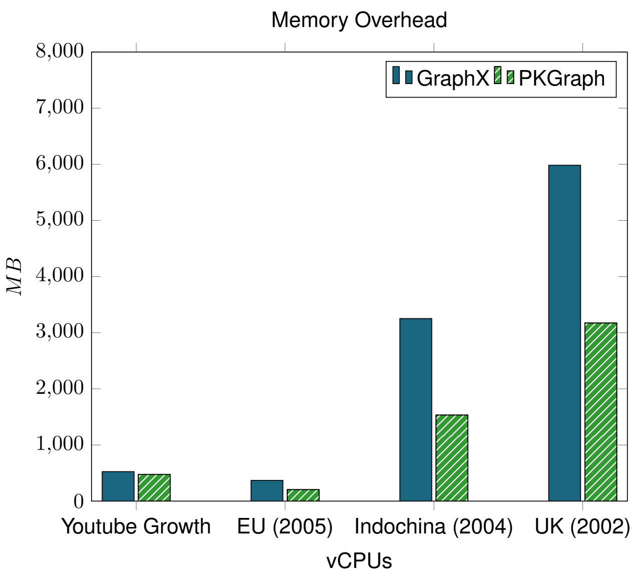


Figure 5.10: Results of the memory overhead for each dataset

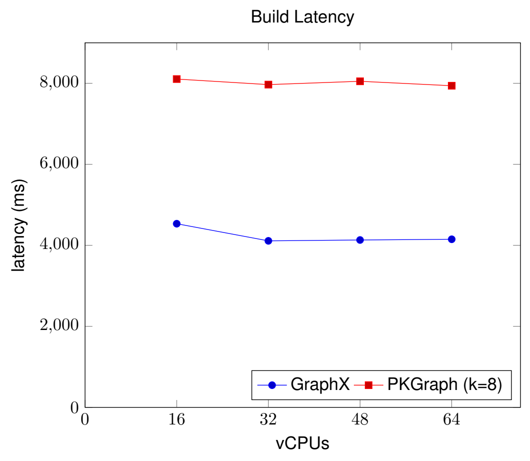
entire graph the reduction now is between 30% to 50%, in part due to the partitioning of the graph and the nature of the graph. The best performance is obtained in web graphs, since these have much higher edge clustering when compared to other types of graphs. Furthermore, the number of processors has no significant impact on the size in memory of the graph.

5.3.4 Build

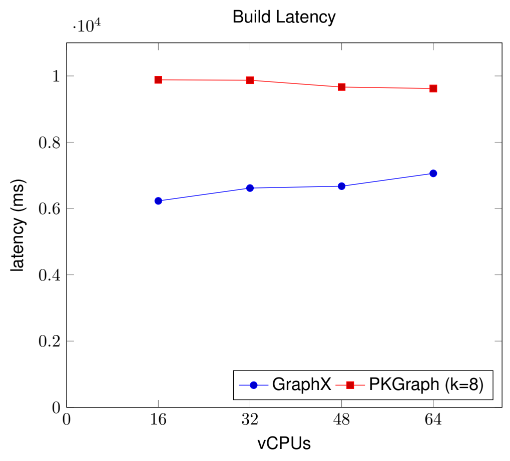
This workload constructs the graph from a given dataset. Each dataset is stored in a single text file and each line corresponds to an edge of the graph. Each worker node in the cluster is responsible for building part of the entire graph, so we should observe that as the number of processors increases the build latency of the entire graph should decrease.

As the results show in Figure 5.11, the GraphX implementation achieves lower build latency's when compared to our implementation, which is the same results obtained on our micro-benchmarks in Section 5.2. For smaller datasets, such as the *Youtube Growth* dataset (Figure 5.11a), the build latency values are not affected as much by the increase in the number of processors, since for smaller datasets the time needed to divide the work through all the workers becomes increasingly larger. For larger datasets, such as *UK (2002)* (Figure 5.11d), the number of processors used has a much bigger effect on the latency.

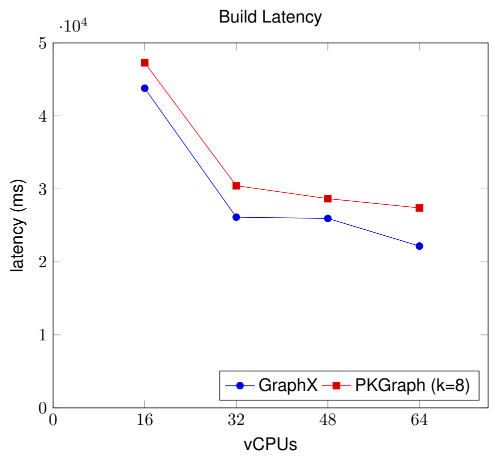
Despite these results, it should be noted that the graph is typically built once at the beginning and reused for every following operation. As such, the cost of this operation is suppressed, especially for long running sessions where the graph is reused multiple times.



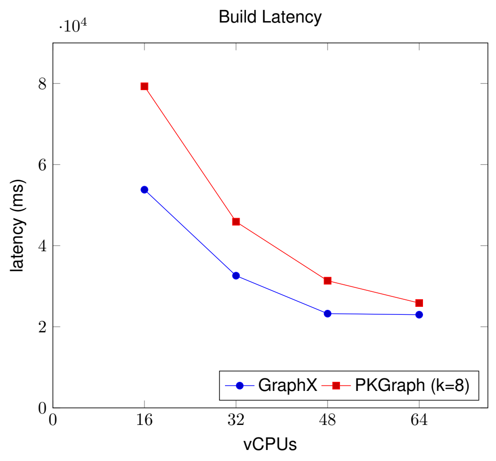
(a) Youtube Growth



(b) EU (2005)



(c) Indochina (2004)



(d) UK (2002)

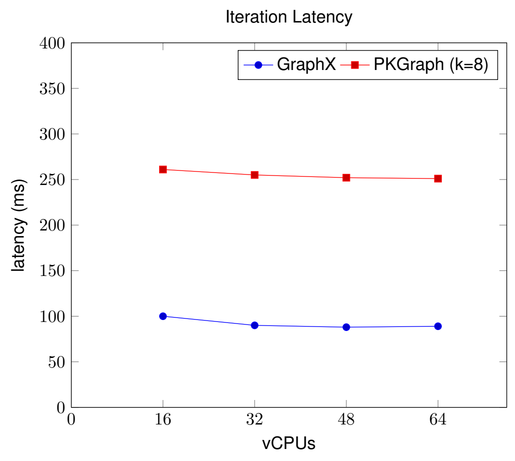
Figure 5.11: Results of the build latency for each dataset

5.3.5 Iteration

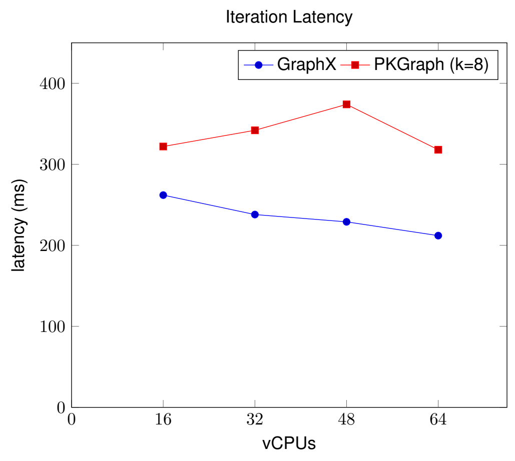
This workload iterates all edges of the graph and applies a user function to each edge. The results obtained are showed in Figure 5.12.

Just like the previous tests, as the number of processors increases, the iteration latency decreases. Due to the GraphX implementation being much more efficient at traversing all edges in an edge partition, it achieves a lower latency compared to our implementation, even using a more processing optimized k value.

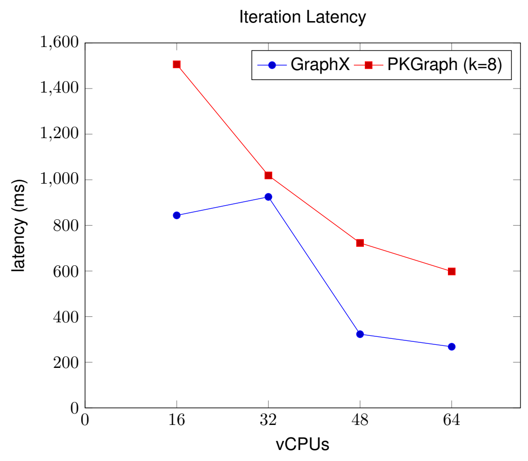
Overall, our implementation, in terms of iteration latency, is between 15% to 40% slower than the GraphX implementation (which uses 2x or more memory than our solution), depending on the type of graph, obtaining better results for web graphs when compared to social network graphs.



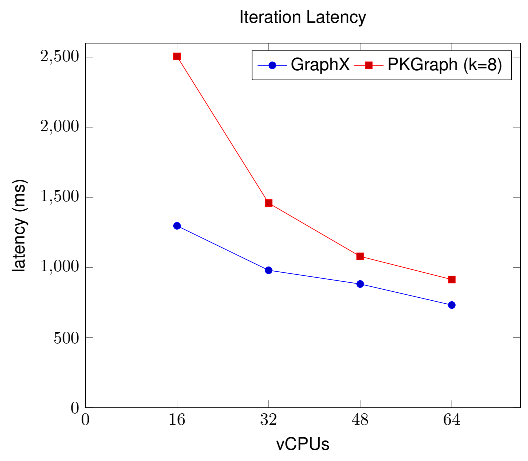
(a) Youtube Growth



(b) EU (2005)



(c) Indochina (2004)



(d) UK (2002)

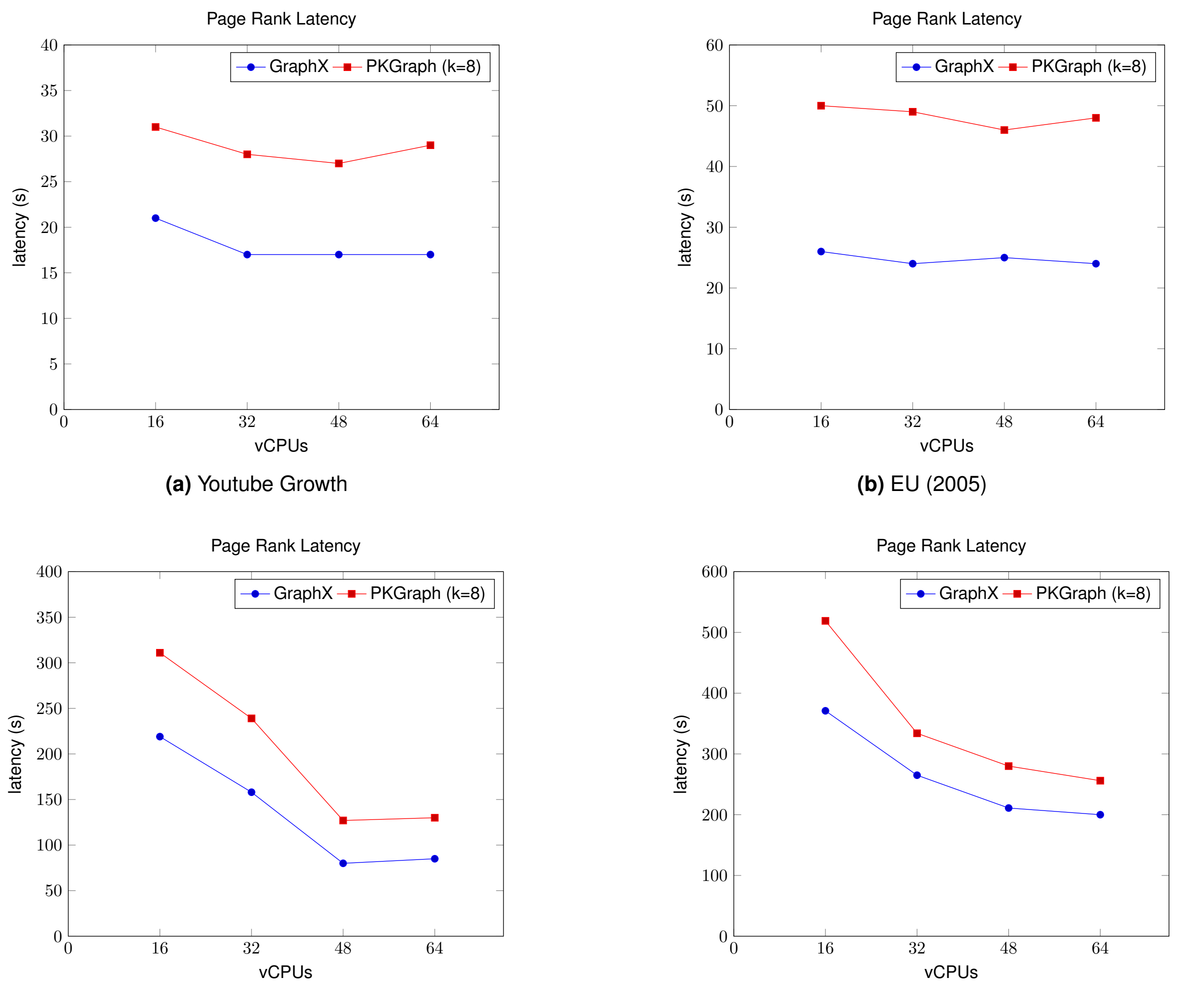
Figure 5.12: Results of the iteration latency for each dataset

5.3.6 PageRank

This workload executes the *PageRank* algorithm on the graph. This is a relevant graph algorithm used to measure the importance of web pages, used by Google Search to rank pages in their search engine results. Figure 5.13 shows the results obtained.

This workload is the longest running out of all workloads tested, since it requires running multiple iterations until the edge values no longer vary more than a certain tolerance. For these tests, a tolerance of 0.1 is used.

The results show that although our implementation has slightly higher latency than the GraphX implementation, it still achieves competitive results, having an increase in latency of around 20% while only having half of the original graph size. The results also show that, for large graphs, as the number of processors available increases the latency of the graph operation decreases, this can be explained by having more processors available for processing and incurring in less of an overhead in network



(a) Youtube Growth **(b)** EU (2005) **(c)** Indochina (2004) **(d)** UK (2002)

Figure 5.13: Results of the latency of the Page Rank algorithm for each dataset

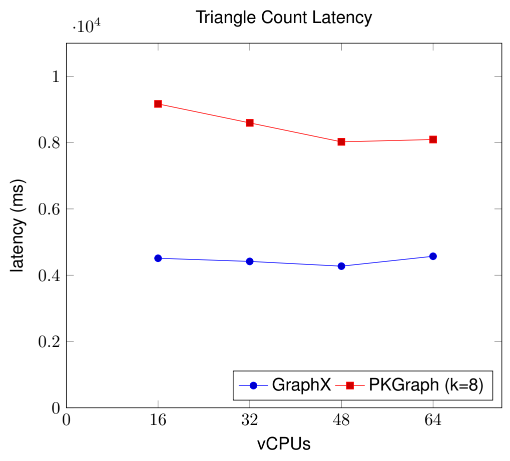
communication.

5.3.7 Triangle Count

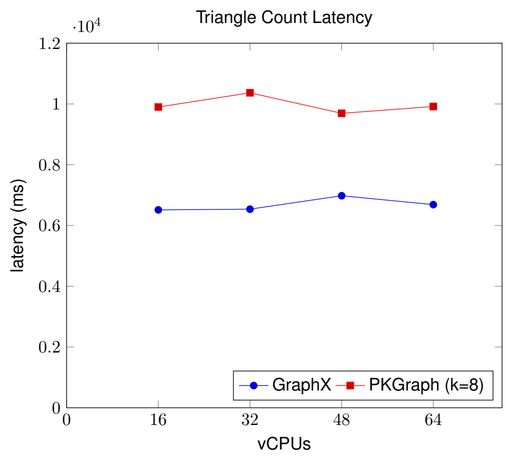
This workload executes the *Triangle Count* algorithm on the graph. This algorithm counts the number of triangles in a graph. A triangle is a set of three vertices, where each vertex is connected to all other vertices. This algorithm is typically used in social network analysis to detect communities and measure clustering coefficients. Figure 5.14 shows the results obtained.

This graph algorithm has much less latency than the *Page Rank* algorithm tested previously, and as such the difference between the two implementations becomes even smaller. Still, this graph algorithm shows similar results to the previous test, with the GraphX implementation still achieving a lower latency than our implementation.

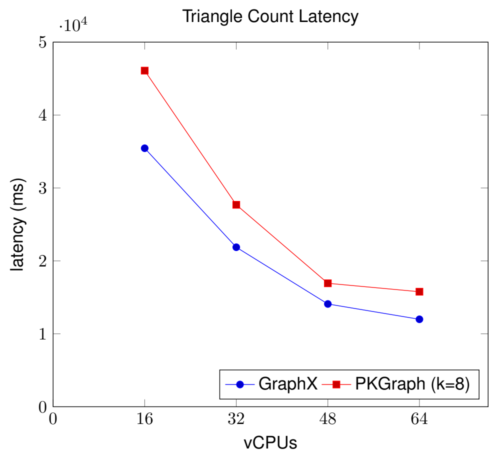
In both the *Triangle Count* and *Page Rank* algorithms the results shows that our solution's perfor-



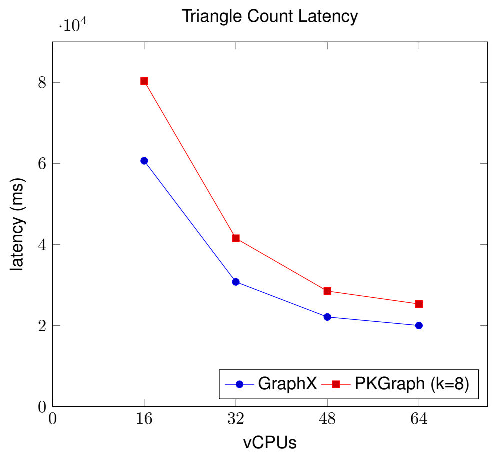
(a) Youtube Growth



(b) EU (2005)



(c) Indochina (2004)



(d) UK (2002)

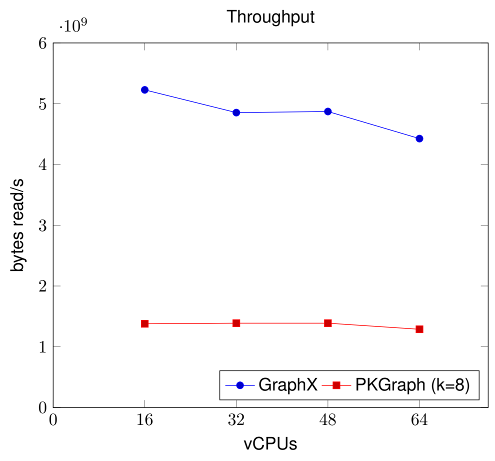
Figure 5.14: Results of the latency of the Triangle Count algorithm for each dataset

mance increases with the size of the graph. The performance penalty of our implementation decreases as the complexity of the algorithms executed or the graph size increases, demonstrating the scalability of our solution.

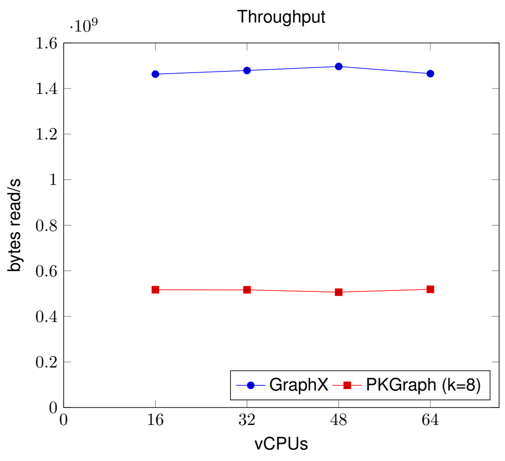
5.3.8 Throughput

To measure this metric, we analyzed the total number of bytes read and the total execution time, calculating the average number of bytes read per second for a predefined workload consisting in iterating all edges of a graph. Figure 5.15 shows the results obtained.

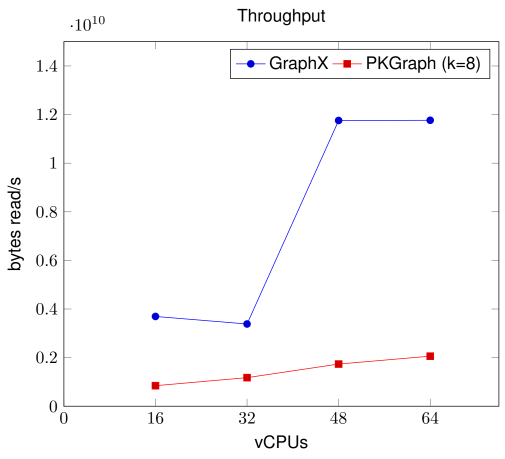
As expected, as the number of processors increases so does the average throughput. The GraphX implementation achieves higher throughput values when compared to our implementation, which matches with the lower latency values obtained in the previous tests.



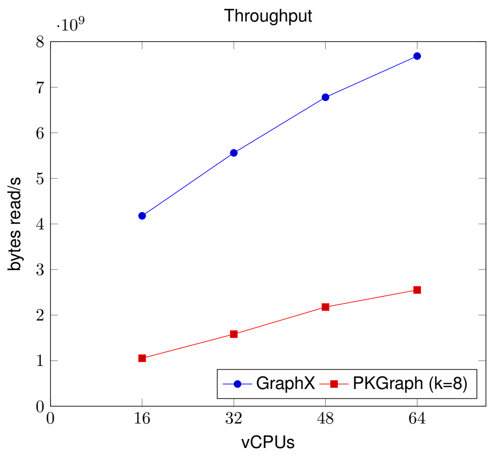
(a) Youtube Growth



(b) EU (2005)



(c) Indochina (2004)



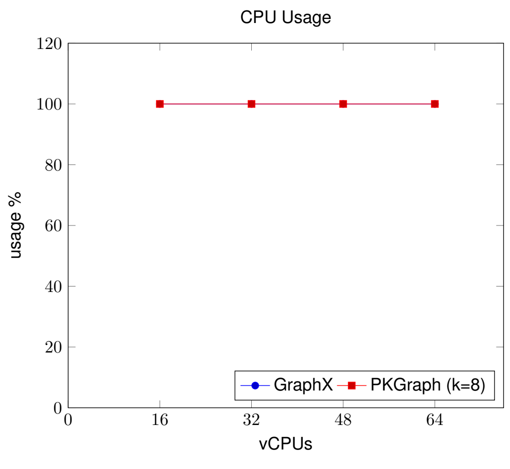
(d) UK (2002)

Figure 5.15: Results of the average throughput for each dataset

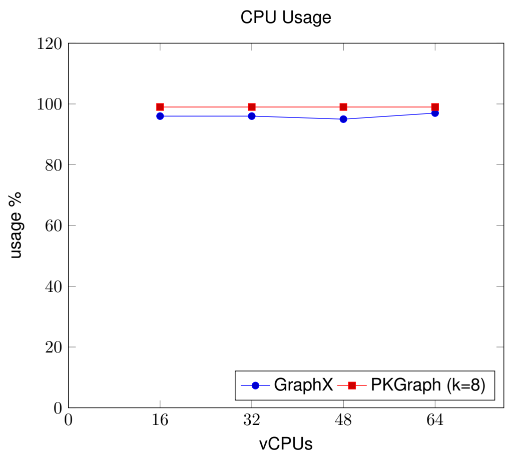
5.3.9 CPU Usage

For this metric we compared the total run time of the *Spark* executors with their total CPU time for each workload and calculated the overall average value. The CPU usage shows the percentage of the total runtime spent on the processor. The higher the CPU usage the less time was spent waiting for Input/Output (IO) operations to finish. Figure 5.16 shows the results obtained.

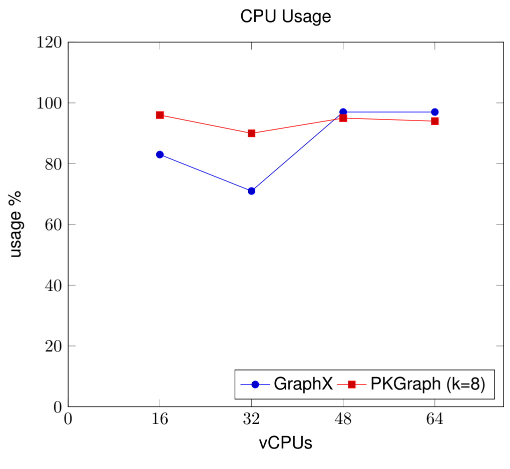
The results show that our implementation incurs in a higher CPU usage than the GraphX implementation due to the added complexity in processing the graph, seeing as the iteration algorithms used by our solution require more processing than the GraphX implementation.



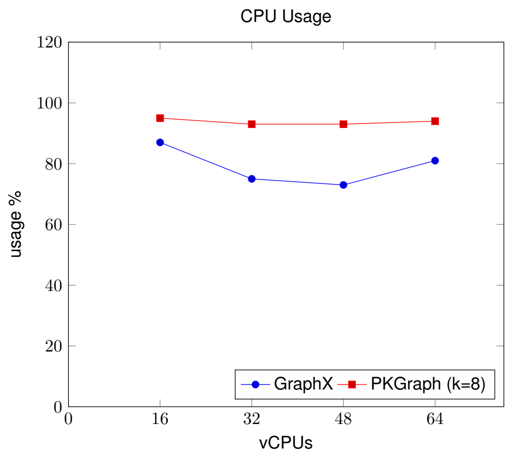
(a) Youtube Growth



(b) EU (2005)



(c) Indochina (2004)



(d) UK (2002)

Figure 5.16: Results of the average CPU usage for each dataset

5.4 Optimization

Although our implementation has significantly less memory usage than the GraphX implementation, it has a worse performance at iterating the edges of a partition, which as we saw is a basic operation that is then used to implement all other partition operations.

One possible optimization aims at improving the performance when iterating an edge partition while still having a low memory overhead. The latency overhead observed when iterating a k^2-tree is mostly due to the height of the tree. Iterating child nodes belonging to a parent node is very efficient by the use of the *nextSetBit* operation of a *BitSet*, which can find the next bit with a value of one in $O(1)$ time complexity. This means that trees with a smaller height will have much better performance than trees with large heights, hence why higher values of k, although having a higher memory overhead, achieve a better iterating performance.

This optimization focuses on changing how the k^2-tree is iterated by only iterating the leaf nodes, instead of traversing the entire height of tree. The information in the leaf nodes is not sufficient to determine the line and column of the corresponding edge, as such, the information about the parent node of each leaf node is also kept.

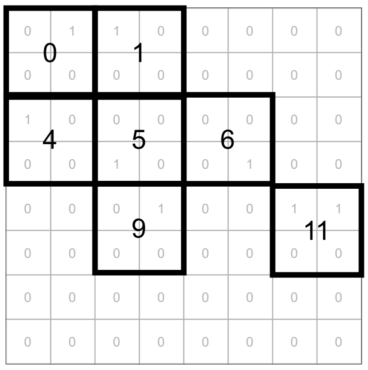


Figure 5.17: Example of assigning row major indices to a parent node

Each parent node is assigned an index, this index corresponds to the row major index of the quadrant the parent node represents. Since only the parent nodes of leaf nodes are kept, the quadrant will be a submatrix of the adjacency matrix with a size of k, holding k^2 cells. Figure 5.17 shows an example of assigning an index to each parent node in an adjacency matrix that is being represent by a k^2-tree with $k = 2$.

The indices of all parent nodes with leaf nodes are then kept in an array of 64 bit integers. Each index encodes the line and column of the parent node, which is then used as the line and column offset to apply to their respective child nodes. All that is left is to calculate the lines and columns of the leaf nodes inside their parent node's quadrant and add the offsets.

With this optimization, there is no need to store the entire k^2-tree, only the bits of the last level and the array containing the parent indices. The compression capabilities of the k^2-tree are also kept, although to a lesser extent, since each parent node will represent k^2 bits. This means that for sparse graphs this optimization should still achieve very good compression results. See Figure 5.18 for an example of bits and parent indices used to represent a standard k^2-tree.

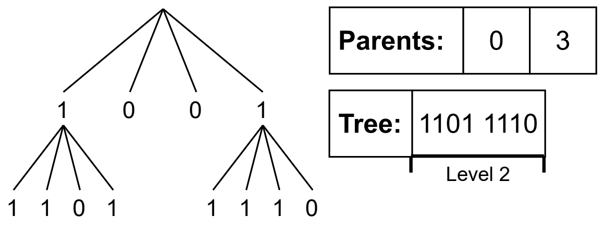


Figure 5.18: Example of the optimized iterator representation of a standard k^2-tree

However, since each parent index takes up 8 bytes, this approach requires slightly more memory overhead than the standard k^2-tree implementation, especially for very small graphs which would only require a few bytes.

Building this optimized representation is the same as building a standard k^2-tree with the exception that the parent index array also needs to be constructed as the parent node bits are inserted into to the tree and all levels of the k^2-tree except the last one are discarded.

The iteration then becomes simply iterating all bits of the last level and keeping track of their corresponding parent node index. Every time we iterate a sequence of k^2 bits we move to the next parent node (see Figure 5.19).

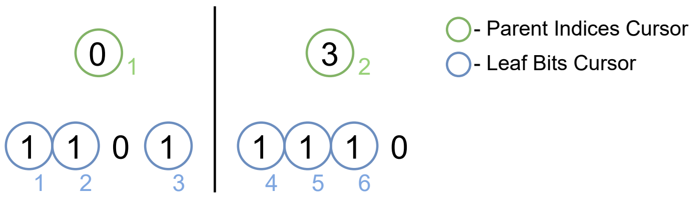


Figure 5.19: Example of iterating the edges of the optimized iterator representation.

The benchmark results show that this representation achieves significantly better performance in iterating edges than our standard implementation, with minimal memory overhead added to the edge partition.

Figure 5.20 shows the latency in iterating all edges of an edge partition with varying number of edges.

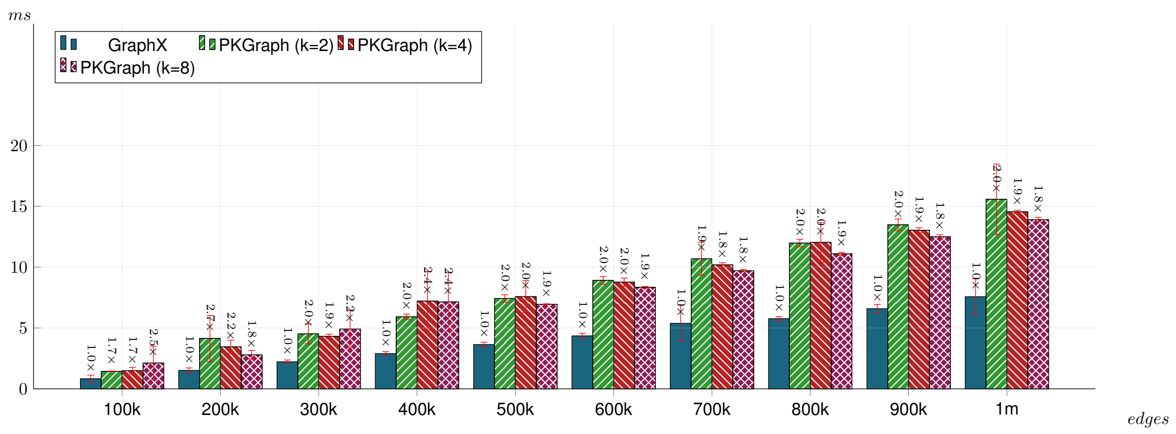


Figure 5.20: Edge partition iteration results using the optimized iterator

The results show a performance improvement of up to 60% when compared with the standard approach, depending on the k value used and the number of edges in the partition. However, it still has slightly more latency than the GraphX implementation. As observed with the standard implementation, the higher the k value is, the better the performance in iterating the partition.

Figure 5.21 shows the memory overhead of an edge partition using the optimized iterator representation with varying number of edges.

The results show only a slight increase in the memory overhead, in the order of 33% in comparison with the standard approach for $k = 2$ (meaning that this approach uses 66% of the original size) and a

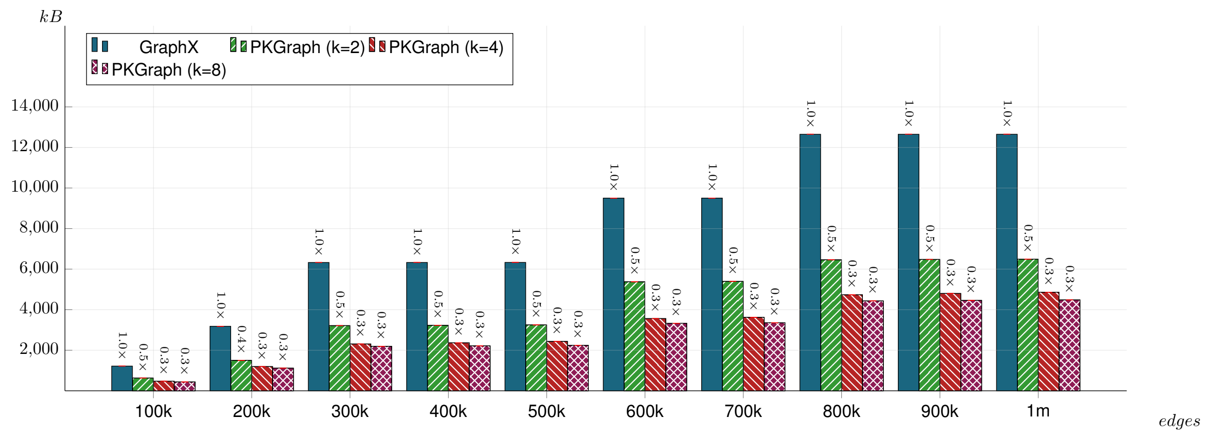


Figure 5.21: Edge partition memory overhead results using the optimized iterator

smaller increase for higher values of k. In all cases, the memory usage is still lower than the GraphX implementation. And again, the higher the k value is, the less the memory usage of the edge partition.

5.5 Analysis

In this chapter we provided a detailed evaluation of our implementation. Overall our solution provides a significant reduction in memory usage, between 40% and 50% depending on the k value used for the k^2-tree, the type of graph and the partitioning strategy employed. As we are using a k^2-tree as the compressed data structure, the more sparse the adjacency matrix of the graph is, the better the compression.

Our implementation also provides a competitive processing performance when compared to the *GraphX* implementation, specially considering that this current *GraphX* approach focuses mainly on having the best possible processing performance by keeping all edges in an array with no compression whatsoever.

Our results show a slowdown limited to 1.4 (meaning 40% slower) when compared to the GraphX implementation, which has a memory overhead 2x larger than our solution. Thus, the relative gains in memory savings outweigh the relative overhead in performance. This net favourable trade off could enable, when needed, employing more partitions (for higher processing power, e.g. in burst situations), without sacrificing memory savings completely (note that memory reserved for each partition is harder to reassign quickly, as opposed to CPU).

We also analyzed some possible optimizations to further improve the processing performance of our solution (60% improvement compared to our initial approach), but at the cost of more memory usage (33% more than our initial approach and 66% of the original size) and still not as efficient as the *GraphX* implementation.

6
Conclusion

Contents

6.1 System Limitations and Future Work 80

Graphs are now more relevant than ever, and its becoming increasingly more important to keep the entire graph in main memory to provide fast access to the underlying data.

Our work focused on reducing the memory usage of graphs while still maintaining a competitive processing performance. The main goal of our work was to design and develop an extension to the storage component of the *GraphX* distributed graph processing system so that the processed graph is made more space-efficient by using the k^2-tree lossless compressed representation, while also aiming to achieve similar performance to the uncompressed version.

To achieve this goal we presented a survey of the current state of the art on the storage components of graph databases and graph processing systems, as well as optimized graph representations with the goal of reducing the memory footprint of the graph while still maintaining fast access to uncompressed data.

Our solution consisted in implementing an extension to the *GraphX* graph processing system using the k^2-tree as the optimized graph representation in a distributed setting. We described the architecture of our solution in which we would make use of the compressed data structure to implement the edge partitions of the graph in the *Spark* ecosystem.

The architecture is built on the already existing graph abstraction implemented in the *GraphX* system. This abstraction is built on top of *Spark* RDDs that store the vertices and edges of a graph, using a number of partitions. Our work focused primarily in the memory usage of the edge partitions, while reusing the existing implementation in *GraphX* for the vertex partitions. To reduce the memory usage of the overall graph our solution made use of the k^2-tree compress data structure, capable of representing very efficiently sparse adjacency matrices which are very common in web graphs.

We then presented our implementation of a graph, called *PKGraph*, and described the details of the compressed data structure, the construction of the edge partitions and their processing, explaining some of the limitations of our approach and some possible optimizations that could be applied to enhance the performance of the implementation.

Finally, we performed a detailed evaluation of our implementation split into micro-benchmarks, which evaluated the performance of the edge partition in a single machine, and macro-benchmarks which were performed in a cluster of *Spark* workers and evaluated the performance of the overall graph, using various datasets to showcase the effectiveness of our solution in both web and non-web graphs, as well as how our solution scales as the size of the graph and number of available processors increase.

Our evaluation concluded that our solution offers a significant reduction in the memory usage of a graph, specially for web graphs, while maintaining a competitive processing performance when compared to the *GraphX* implementation.

6.1 System Limitations and Future Work

As explained in Section 4.4, our solution depends on good data locality in the adjacency matrix to provide an efficient k^2-tree compression, otherwise the resulting partition will be able to store much more edges than necessary, which results in a k^2-tree with a unnecessarily large height.

One possible solution would be to use an hybrid k^2-tree as proposed by Brisaboa et al. [22], in which the first level of the tree uses a different k value then the remaining levels. This approach leads to a tree with a smaller height at the cost of slightly more memory usage.

As noted in Chapter 5 the height of the tree is the main factor in the tree's iteration performance, so by reducing the height of the tree, this performance can be improved with a very slightly increase in the memory usage of the overall graph.

The vertex partitions could also be implemented with a similarly lossless data structure to further reduce the space overhead of the entire graph. Although the number of vertices are significantly less than that of the edges, the vertices are still replicated throughout the edge partitions as a caching mechanism. By optimizing the space overhead of the vertices, the overall size of the graph, as well as that of the edge partitions, could be significantly reduced.

Bibliography

[1] M. E. Coimbra, A. P. Francisco, and L. Veiga, "An analysis of the graph processing landscape," *J. Big Data*, vol. 8, no. 1, p. 55, 2021. [Online]. Available: https://doi.org/10.1186/s40537-021-00443-9

[2] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin, "Graphjet: real-time content recommendations at twitter," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1281–1292, 2016.

[3] M. Mohammadrezaei, M. E. Shiri, and A. M. Rahmani, "Identifying fake accounts on social networks based on graph analysis and classification algorithms," *Security and Communication Networks*, vol. 2018, 2018.

[4] S. Zhang, H. Chen, K. Liu, and Z. Sun, "Inferring protein function by domain context similarities in protein-protein interaction networks," *BMC bioinformatics*, vol. 10, no. 1, pp. 1–6, 2009.

[5] Y. Zhang, H. Lin, Z. Yang, J. Wang, and Y. Liu, "An uncertain model-based approach for identifying dynamic protein complexes in uncertain protein-protein interaction networks," *BMC genomics*, vol. 18, no. 7, p. 743, 2017.

[6] C. Cooper and A. Frieze, "A general model of web graphs," *Random Structures & Algorithms*, vol. 22, no. 3, pp. 311–335, 2003.

[7] S. Raghavan and H. Garcia-Molina, "Representing web graphs," in *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE, 2003, pp. 405–416.

[8] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhange, "Understanding ethereum via graph analysis," in *IEEE INFOCOM 2018-IEEE conference on computer communications*. IEEE, 2018, pp. 1484–1492.

[9] W. Chan and A. Olmsted, "Ethereum transaction graph analysis," in *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, 2017, pp. 498–500.

[10] M. E. Coimbra, M. Selimi, A. P. Francisco, F. Freitag, and L. Veiga, "Gelly-scheduling: distributed graph processing for service placement in community networks," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, H. M.

Haddad, R. L. Wainwright, and R. Chbeir, Eds. ACM, 2018, pp. 151–160. [Online]. Available: https://doi.org/10.1145/3167132.3167147

[11] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[12] M. Gabielkov and A. Legout, "The complete picture of the twitter social graph," in *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, 2012, pp. 19–20.

[13] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[14] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a {PC}," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 31–46.

[15] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First international workshop on graph data management experiences and systems*, 2013, pp. 1–6.

[16] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From" think like a vertex" to" think like a graph"," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.

[17] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, "Gbase: an efficient analysis platform for large graphs," *The VLDB Journal*, vol. 21, no. 5, pp. 637–650, 2012.

[18] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, 2013, pp. 1–12.

[19] M. Kaepke and O. Zukunft, "A comparative evaluation of big data frameworks for graph processing," in *2018 4th International Conference on Big Data Innovations and Applications (Innovate-Data)*. IEEE, 2018, pp. 30–37.

[20] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 527–543.

[21] J. Guia, V. G. Soares, and J. Bernardino, "Graph databases: Neo4j analysis." in *ICEIS (1)*, 2017, pp. 351–356.

[22] N. R. Brisaboa, S. Ladra, and G. Navarro, "k 2-trees for compact web graph representation," in *International symposium on string processing and information retrieval*. Springer, 2009, pp. 18–30.

[23] N. Martínez-Bazan, M. Á. Águila-Lorente, V. Muntés-Mulero, D. Dominguez-Sal, S. Gómez-Villamor, and J.-L. Larriba-Pey, “Efficient graph management based on bitmap indices,” in *Proceedings of the 16th International Database Engineering & Applications Sysmposium*, 2012, pp. 110–119.

[24] B. Wheatman and H. Xu, “Packed compressed sparse row: A dynamic graph representation,” in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[25] M. E. Coimbra, A. P. Francisco, L. M. S. Russo, G. de Bernardo, S. Ladra, and G. Navarro, “On dynamic succinct graph representations,” in *Data Compression Conference, DCC 2020, Snowbird, UT, USA, March 24-27, 2020*, A. Bilgin, M. W. Marcellin, J. Serra-Sagristà, and J. A. Storer, Eds. IEEE, 2020, pp. 213–222. [Online]. Available: https://doi.org/10.1109/DCC47342.2020.00029

[26] N. Doekemeijer and A. L. Varbanescu, “A survey of parallel graph processing frameworks,” *Delft University of Technology*, vol. 21, 2014.

[27] K. Ammar and T. Ozsu, “Experimental analysis of distributed graph systems,” *arXiv preprint arXiv:1806.08082*, 2018.

[28] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “Powergraph: Distributed graph-parallel computation on natural graphs,” in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 17–30.

[29] 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*, 2010, pp. 135–146.

[30] A. V. Gerbessiotis and L. G. Valiant, “Direct bulk-synchronous parallel algorithms,” *Journal of parallel and distributed computing*, vol. 22, no. 2, pp. 251–267, 1994.

[31] S. Oswal, A. Singh, and K. Kumari, “Deflate compression algorithm,” *International Journal of Engineering Research and General Science*, vol. 4, no. 1, pp. 430–436, 2016.

[32] Y. Zhang, H. Jiang, F. Wang, Y. Hua, D. Feng, Y. Cheng, Y. Hu, and R. Xiao, “Cic-pim: Trading spare computing power for memory space in graph processing,” *Journal of Parallel and Distributed Computing*, vol. 147, pp. 152–165, 2021.

[33] M. Mariappan, J. Che, and K. Vora, “Dzig: sparsity-aware incremental processing of streaming graphs,” in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 83–98.

[34] M. E. Coimbra, S. Esteves, A. P. Francisco, and L. Veiga, “Veilgraph: Streaming graph approximations,” 2019.

[35] R. kumar Kaliyar, “Graph databases: A survey,” in *International Conference on Computing, Communication & Automation*. IEEE, 2015, pp. 785–790.

[36] A. Deutsch, Y. Xu, M. Wu, and V. Lee, “Tigergraph: A native mpp graph database,” *arXiv preprint arXiv:1901.08248*, 2019.

[37] N. Martinez-Bazan, S. Gomez-Villamor, and F. Escale-Claveras, “Dex: A high-performance graph database management system,” in *2011 IEEE 27th International Conference on Data Engineering Workshops*. IEEE, 2011, pp. 124–127.

[38] M. Besta and T. Hoefler, “Survey and taxonomy of lossless graph compression and space-efficient graph representations,” *arXiv preprint arXiv:1806.01799*, 2018.

[39] S. Álvarez-García, B. Freire, S. Ladra, and O. Pedreira, “Compact and efficient representation of general graph databases,” *Knowledge and Information Systems*, vol. 60, no. 3, pp. 1479–1510, 2019.

[40] N. R. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, and G. Navarro, “Compressed representation of dynamic binary relations with applications,” *Information Systems*, vol. 69, pp. 106–123, 2017.

[41] D. Arroyuelo, G. de Bernardo, T. Gagie, and G. Navarro, “Faster dynamic compressed d-ary relations,” in *International Symposium on String Processing and Information Retrieval*. Springer, 2019, pp. 419–433.

[42] R. A. Rossi and R. Zhou, “Graphzip: a clique-based sparse graph compression method,” *Journal of Big Data*, vol. 5, no. 1, p. 10, 2018.

[43] N. Ashrafi Payaman and M. Kangavari, “Gssc: Graph summarization based on both structure and concepts,” *International Journal of Information and Communication Technology Research*, vol. 9, no. 1, pp. 33–44, 2017.

[44] G. M. Morton, “A computer oriented geodetic data base and a new technique in file sequencing,” 1966.

[45] S. ur Rehman, A. Nawaz, T. Ali, and N. Amin, “g-sum: A graph summarization approach for a single large social network,” 2021.

[46] J. Ko, Y. Kook, and K. Shin, “Incremental lossless graph summarization,” in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 317–327.

[47] S. Álvarez-García, N. R. Brisaboa, C. Gómez-Pantoja, and M. Marin, “Distributed query processing on compressed graphs using k2-trees,” in *International Symposium on String Processing and Information Retrieval*. Springer, 2013, pp. 298–310.

[48] R. A. Rossi and N. K. Ahmed, “The network data repository with interactive graph analytics and visualization,” in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[49] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” http://snap.stanford.edu/data, Jun. 2014.