PK-Graph: Partitioned K^2-Tree Graph - Extended Abstract

Bruno Alexandre Coimbra Morais
bruno.c.morais@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisbon, Portugal

November 2021

Abstract

Graphs are becoming increasingly larger, having millions of vertices and billions (or even trillions) of edges in some cases. As a result, it is 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 leading to an impact on 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 the 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 representation, graph databases, graph processing systems, optimization, compression

1 Introduction

Graphs [6] are now more relevant than ever, being used in social networks [19, 16], biology [23], the web [8, 17], cryptocurrency [3], and many more fields (e.g., managing community clouds [7]). 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 [4, 9]. 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 the access time and, ideally, even improving it.

Current solutions store graphs in a uncompressed format [22, 20, 12, 18, 11, 10, 14, 13]. 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 [2, 15, 21, 5]. All while maintaining the same performance, or even better, when accessing the graph.

It also may be relevant to modify the graph, such as adding or removing edges/vertices, without having to reconstruct the entire graph. For example, some popular graph algorithms (i.e, `PageRank`) require the attributes stored in the vertices/edges to be mutated.

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.

Current solutions are centered in partitioning graphs based on edges, to better distribute work among computing nodes [22, 14]. 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.

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 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 when adding new elements.

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 representa-

tion. The solution should achieve similar performance to the uncompressed version, ideally even improving it. The solution should also allow for the graph to be fully dynamic, by being possible to mutate attributes and add new vertices and edges.

This paper is structured as follows: Section 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 Section 3 we present the architecture of our solution, in Section 4 we describe the evaluation methodology and the results obtained for our implementation, and in Section 5 we conclude our thoughts on the topic.

2 Related Work

In this section, we present important work on the topic of graph representation and graph processing. We start by presenting relevant **Graph Processing Systems**, focusing on how they handle graph storage and processing, **Graph Databases** and state-of-the-art **Optimized Graph Representations** and processing.

2.1 Graph Processing Systems

Graph processing systems are focused on iterating an input graph and applying it transformations in order to generate a new graph. Our survey focuses on the storage components of these systems, since most try to store the graph in main memory if possible, only using secondary storage when the graph is too large, typically storing it in a type of serialized format.

These systems do not typically require fine grained access to the vertices and/or edges of the graph, instead they iterate all, or some subset of the graph components.

Some systems use a **Shared** architecture to 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.

Another approach is **Distributed** architectures, that partition the graph throughout a cluster of processors, where each processor stores only a fraction of the total graph.

Their programming model may also vary, typically either using a **General Purpose** model not specifically made to handle graphs, **Vertex-Centric** in which a a user defined function is executed in the context of each vertex or **Graph-Centric** for models focused on performing computations in the context of a sub-graph.

The scheme used to partition a graph through a cluster of computing nodes is typically either a **Vertex-Cut**, which divides the graph by its vertices and leads vertices to be distributed across multiple partitions and edges to be assigned to a unique partition, or a **Edge-Cut** which similarly divides the graph by its edges and leads 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.

In terms of dynamism, these systems typically allow for applying changes to the graph (i.e, adding vertices, removing edges, etc), but only by transforming an existing graph. This operation does not necessarily mean that the graph is completely reconstructed. Some systems avoid this by rebuilding only the affected partitions.

An example of a relevant graph processing system is the **GraphX** system. Spark's *API* for graphs and graph-parallel computation [22]. Data storage is handled by Spark's *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.

2.2 Graph Databases

Graph database systems are similar to typical databases, but have specialized formats to efficiently store graphs. These systems also focus on fine grained access to the vertices and edges of a graph, allowing for complex queries to be made and do not necessarily need to traverse the entire graph for each query.

As such, the storage of the graph is made to be very space efficient but also to allow for very low latency when performing queries.

The architecture of these systems is typically either **Native** when the system has an exclusive preference to store graph workloads across its entire stack. This leads to much better performance when handling graphs, compared with more general databases. Or **Non-Native** architectures when a external data source, typically `NoSQL`, is used to store the graph data in a non-optimized representation, leading to worse performance in general. 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.

The storage used by the database system pertains to the location 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.

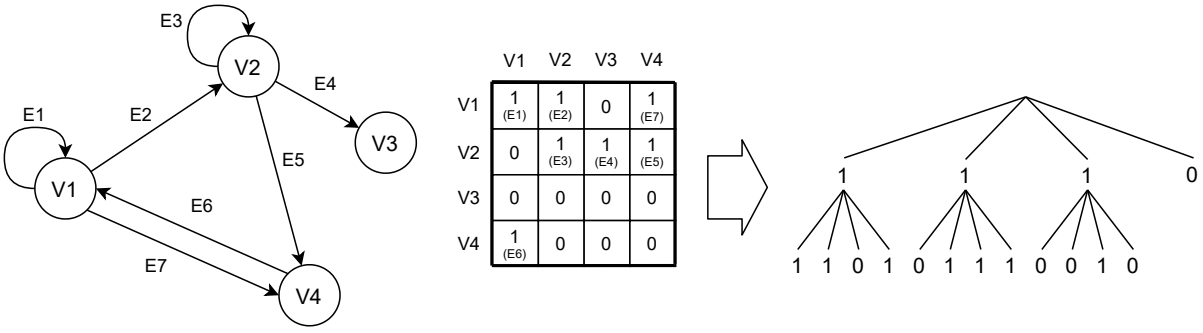


Figure 1: Graph and its adjacency matrix and corresponding k²-tree.

The database can also be *Distributed* when the graph is stored across multiple machines, or *Centralized* when the entire graph is stored in a single machine. In some cases, where specialized hardware is available, centralized systems may have similar or even better performance than distributed ones.

An example of a relevant graph database is **Neo4J**, a *native* graph database [10] platform used to store, query, analyze and manage highly connected data in *property graphs*, providing its own query language (*Cypher*). 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.

2.3 Optimized Graph Representations

Optimized graph representations relate to representations of graphs that are typically compressed to reduce the memory requirements of the entire graph. These representations typically make use of compression and/or summarization techniques to either reduce the memory need to store the graph or reduce the number of existing vertices and/or edges of the graph.

Some representations support attributed graphs (property graphs) and some can also compress the attributes alongside the rest of the graph. The direction of the edges is also relevant for the representation, since some implementations support **Undirected** edges and/or **Directed** edges, which require twice the number of edges.

The type of graph can also support at most a single edge between two vertices (**Simple**) or multiple edges between the same two vertices (**Multi-Graph**).

Most optimized graph representations do not allow for mutable graphs, since these typically require more memory usage than their static counterparts.

An example of a relevant optimized graph representation is the **k²-tree** (Figure 1), a compact graph representation that takes advantage of sparse adjacency matrices. Proposed by Brisaboa et al. [2] 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.

3 PK-Graph

Our solution extends the `GraphX` processing system and make use of a k²-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. 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 RDD` (Resilient Distributed Datasets) 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.

3.1 Overview

Figure 2 shows a 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 **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`.

All graph operations are 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 shows

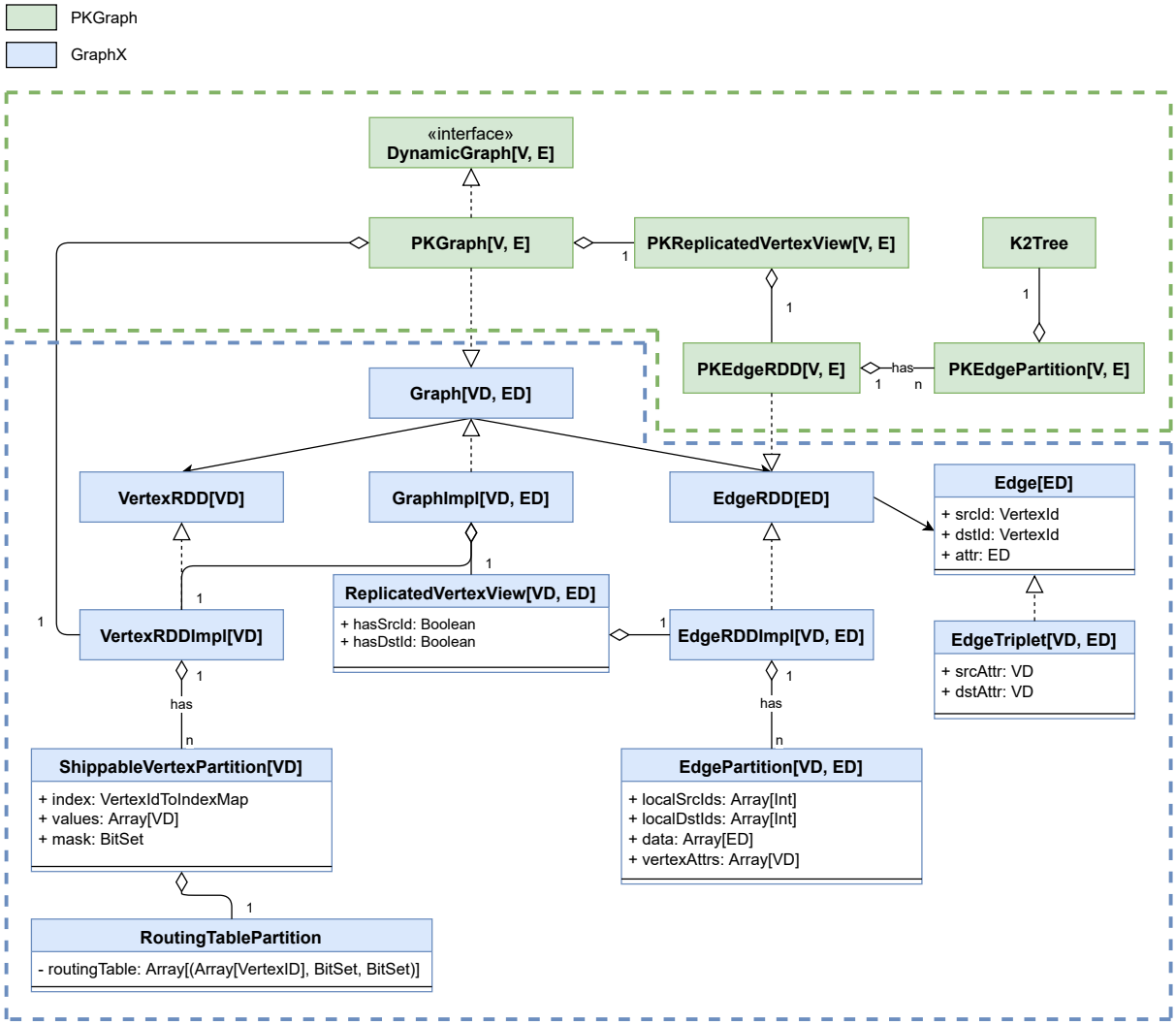


Figure 2: Architecture overview of our system.

an example of how a graph operation can be distributed throughout a cluster.

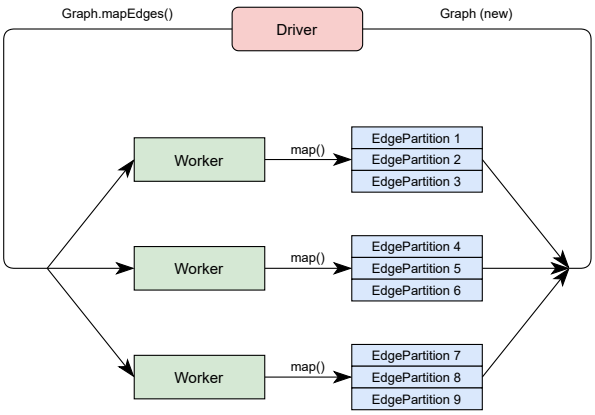


Figure 3: Distributed graph work in a cluster

3.2 Vertices

The `VertexRDD` class provides a interface for vertex specific `RDDs`, containing operations to iterate and transform the underlying vertices of the graph. The **VertexRDDImpl** contains the default `GraphX` implementation of the **VertexRDD** class.

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 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 4b).

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 **mask** bitset keeps track of all active vertices in the partition. The vertex operations of a partition are only applied to the active vertices.

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)
    i ← partition.mask.nextSetBit()
    while i >= 0 do
        vertexId ← partition.index[i]
        attr ← partition.values[i]
        output Vertex(vertexId, attr)
        i ← partition.mask.nextSetBit()
```

3.3 Edges

The **EdgeRDD** class provides a interface for edge specific `RDDs`, containing operations to iterate and transform the underlying edges of the graph.

Our solution extends this abstraction, by the **PKEdgeRDD** class, and provides a specific implementation of the edge partitions (**PKEdgePartition**) using the compressed data structure k²-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 k²-tree compressed data structure as proposed by Brisaboa et al. [2].

Figure 4a shows the interface of one of our edge partitions.

Every operation in the edge partition creates a new instance 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 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** 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.

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 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 dynamic operations (**addEdges** and **removeEdges**) 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 k²-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.

Algorithm 3.2 Algorithm to iterate the edges of a given partition

```
procedure ITERATE_EDGES(partition)
    iterator ← tree_iterator(k^h, 0, 0, −1)   ▷ k^h is the size of the global adjacency matrix
    i ← 0
    while iterator.hasNext() do
        (localSrc, localDst) ← iterator.next()
        srcId ← partition.local2Global[localSrc]
        dstId ← partitino.local2Global[localDst]
        attr ← partition.edgeAttrs[i]
        output Edge(srcId, dstId, attr)
        i ← i + 1

procedure TREE_ITERATOR(size, line, col, pos)
    if x ≥ |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 ← rank(T, pos) · k²                     ▷ k²-tree rank operation to find child node
            for i = 0..k² − 1 do
                tree_iterator(size/k, line · (size/k) + i/k, col · (size/k) + i mod k, y + i)
```

This will require iterating the k²-tree in a depth-first fashion and calculating the line and column in the adjacency matrix of each edge. Each line and column will correspond to local vertex identifiers, which then will need 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 im-

```
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]
}
```

(a) Interface of the **PKEdgePartition** class.

```
class ShippableVertexPartition[VD] {
  // Hash set of vertex IDs.
  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 routing: RoutingTablePartition
}
```

(b) Interface of the **ShippableVertexPartition** class.

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

(c) Interface of the **PKReplicatedVertexView** class.

```
class DynamicGraph[V, E] {
  def addVertices(...): Graph[V, E]
  def addEdges(...): Graph[V, E]
  def removeVertices(...): Graph[V, E]
  def removeEdges(...): Graph[V, E]
}
```

(d) Interface of the **DynamicGraph** class.

Figure 4: Relevant class methods of PKGraph.

plementation 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 4c shows the interface of this class.

This class stores the underlying **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.

3.4 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 will require determining the partitions affected, and only transforming these. Figure 4d shows the 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.

All dynamic functions receive `RDD` instances as parameters to allow for these operations to be distributed throughout a computing cluster.

3.5 Partitioning

Because `GraphX` processes the graph data in a distributed fashion, our solution will also need 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 will interpret them as an edge adjacency matrix that will be partitioned using a 2D partitioning scheme [1] that splits the adjacency matrix into several submatrices of equal size, each assigned to a unique partition (see Fig. 5).

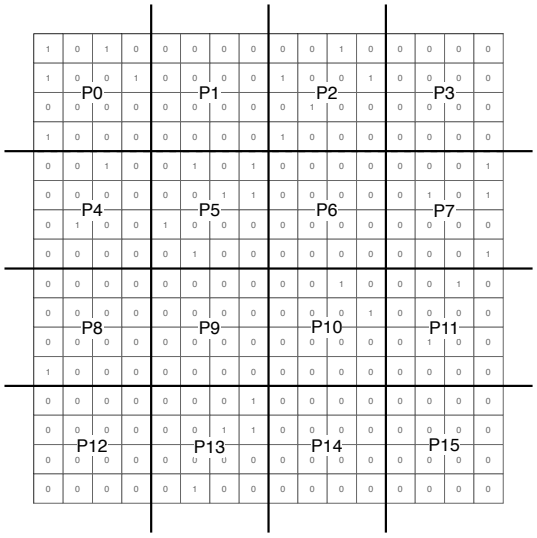


Figure 5: Adjacency matrix partitioning scheme

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 locations in order to evenly distribute them through all partitions.

Like `GraphX`'s implementation, our solution will also replicate 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 interface 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 **repartition** function. When increasing the number of partitions these will be shuffled, which will incur 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 `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 will also introduce 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.

3.6 Discussion

Our solution improves 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 will implement a dy-

namic interface that allows to add and remove vertices and edges.

4 Evaluation

To evaluate the implementation of our solution, we performed various benchmarks in a cluster of computing nodes, each node corresponding to a Spark worker that keeps part of the total graph in main memory.

We submitted 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 analyze the gains (penalties) our solution has in terms of storage compression and processing.

4.1 Setup

The cluster was prepared using the `AWS 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 6). 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.

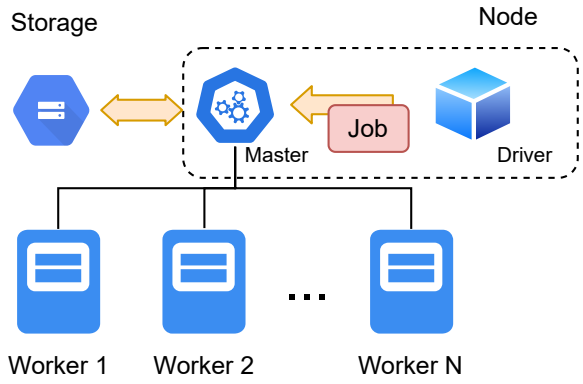


Figure 6: Overview of the cluster used to execute spark jobs.

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

4.2 Datasets

The datasets used in the evaluation of our implementation are from the Network Repository and the Stanford Large Network Dataset Collection. The datasets chosen for the benchmarks are the following:

- **Youtube Growth** (3M vertices, 12.2M edges)
- **EU (2005)** (863K vertices, 19M edges)
- **Indochina (2004)** (7M vertices, 194M edges)
- **UK (2002)** (18M vertices, 298M edges)

4.3 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 7 shows the results of the memory usage of the entire graph with varying datasets.

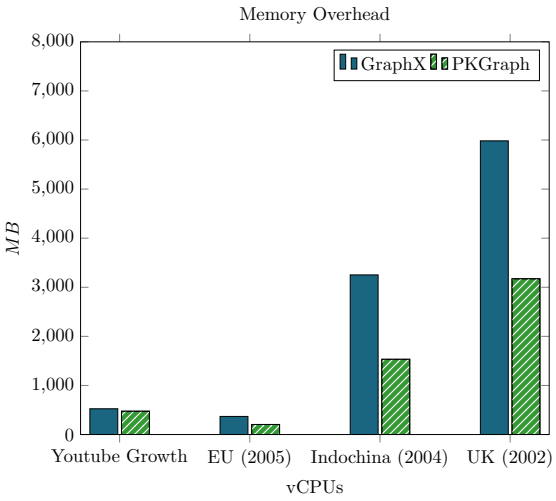


Figure 7: Results of the memory overhead for each dataset

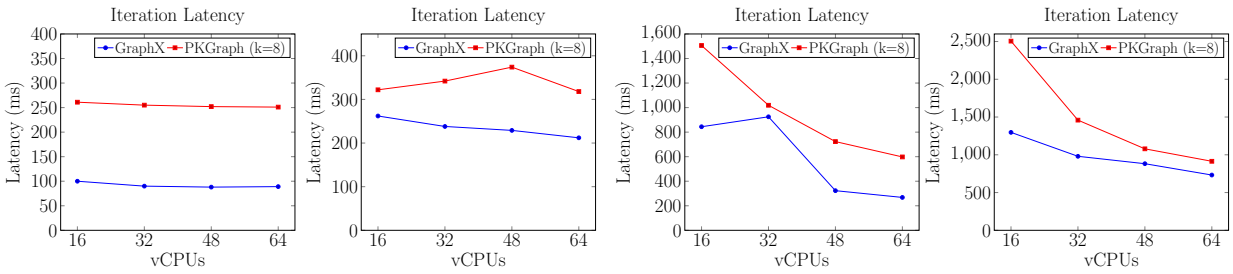
The results show, as did the micro-benchmark results, that our solution has significantly less memory overhead then the `GraphX` implementation. Although our previous tests showed a reduction between 60% to 70% when compared to the `GraphX` implementation, when testing the memory usage of the entire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

4.4 Iteration

This workload iterates all edges of the graph and applies a user function to each edge. The results obtained are showed in Figures 8a, 8b, 8c, 8d.

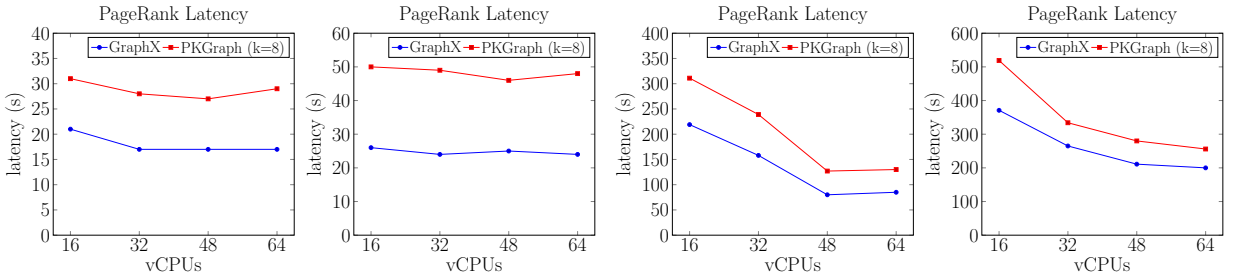
Just like the previous tests, as the number of processors increases, the iteration latency decreases. Due to the GraphX implementation being much more efficient at traversing all edges in an edge partition, it achieves a lower latency compared to our implementation, even using a more processing optimized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

Overall, our implementation, in terms of iteration latency, is between 15% to 40% slower than the GraphX implementation, depending on the type of



(a) `YouTube Growth`. **(b)** `EU (2005)`. **(c)** `Indochina (2004)`. **(d)** `UK (2002)`.

Figure 8: Iteration latency results and vCPU counts for the chosen datasets.



(a) `YouTube Growth`. **(b)** `EU (2005)`. **(c)** `Indochina (2004)`. **(d)** `UK (2002)`.

Figure 9: *PageRank* latency results and vCPU counts for the chosen datasets.

graph, obtaining better results for web graphs when compared to social network graphs.

4.5 PageRank

For the *PageRank* algorithm, we observe similar patterns to the basic iteration test, with PK-Graph's latency approaching that of `GraphX` with higher vCPU counts on the `Indochina (2004)` and `UK (2002)` datasets. The latency results for *PageRank* are depicted in Figures 9a, 9b, 9c and 9d. For larger graphs, as the number of available processors increases, the latency of the graph operation decreases.

4.6 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 application of compression techniques. The performance penalty of PK-Graph decreases in inverse relation with the complexity of the workload algorithm and the size of the dataset. Nonetheless, while requiring less memory (the resource harder to share across time between workloads), results show that PK-Graph incurs at times in a higher CPU usage than `GraphX`, due to the increased graph processing complexity over the compact data structure, as our iteration algorithms are more demanding on it.

5 Conclusion

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 exten-

sion 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.

To reduce the memory usage of the overall graph our solution made use of the k^2-tree compress data structure, capable of very efficiently representing sparse adjacency matrices which are very common in web graphs.

Finally, we performed a detailed evaluation which evaluated the performance of our implementation 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 the 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.

References

[1] 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*, pages 298–310. Springer, 2013.

[2] 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*, pages 18–30. Springer, 2009.

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

[4] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[5] M. E. Coimbra, A. P. Francisco, L. M. S. Russo, G. de Bernardo, S. Ladra, and G. Navarro. On dynamic succinct graph representations. In A. Bilgin, M. W. Marcellin, J. Serra-Sagristà, and J. A. Storer, editors, *Data Compression Conference, DCC 2020, Snowbird, UT, USA, March 24-27, 2020*, pages 213–222. IEEE, 2020.

[6] M. E. Coimbra, A. P. Francisco, and L. Veiga. An analysis of the graph processing landscape. *J. Big Data*, 8(1):55, 2021.

[7] 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 H. M. Haddad, R. L. Wainwright, and R. Chbeir, editors, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 151–160. ACM, 2018.

[8] C. Cooper and A. Frieze. A general model of web graphs. *Random Structures & Algorithms*, 22(3):311–335, 2003.

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

[10] J. Guia, V. G. Soares, and J. Bernardino. Graph databases: Neo4j analysis. In *ICEIS (1)*, pages 351–356, 2017.

[11] 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)*, pages 30–37. IEEE, 2018.

[12] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: an efficient analysis platform for large graphs. *The VLDB Journal*, 21(5):637–650, 2012.

[13] 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)*, pages 31–46, 2012.

[14] 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.

[15] 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*, pages 110–119. 2012.

[16] 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*, 2018, 2018.

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

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

[19] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin. Graphjet: real-time content recommendations at twitter. *Proceedings of the VLDB Endowment*, 9(13):1281–1292, 2016.

[20] 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*, 7(3):193–204, 2013.

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

[22] 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*, pages 1–6, 2013.

[23] S. Zhang, H. Chen, K. Liu, and Z. Sun. Inferring protein function by domain context similarities in protein-protein interaction networks. *BMC bioinformatics*, 10(1):1–6, 2009.