Towards Interactive Design of Graph Data Structures

Towards Interactive Design of Graph Data Structures

Rachna Sha

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

May 2021

Abstract

If I have seen further, it is by standing on the shoulders of Giants. – ***Sir Isaac Newton.***

Graph Frameworks and Databases are critical components of modern software. The need to analyze massive graph datasets have spurred the development of Graph Systems. Graph Frameworks and Libraries with tuned Graph data structures are continuously being developed to handle new workloads and data patterns. This presents a need for a Graph system that has knowledge of its design space and is capable of combining fundamental design constructs to generate optimal graph data structures for a given hardware, data pattern and workload. We propose leveraging non-graph systems with these capabilities and with an overlap in its design space with Graph systems, to bring this intelligence to Graph Systems. As a first step in this process, we have implemented a Key-Value Graph Generator, that demonstrates the use of key-value approach in designing Adjacency List and Compressed Sparse Row (CSR). Our hypothesis is that if we can successfully model Graph data structures using key-value approach then we can leverage learned key-value system and create an interactive and automatic Graph system.

Dedication

Dedicated to my beloved parents, who have raised me to be the person I am. My father, who changed my life in grade 7 by teaching me math and science in a way that I fell in love and never looked back, and my mother who gave me the foundation on which my life exists. Thank you can never be enough for all the sacrifices you made to give me and my sisters the best education possible, for your unconditional love, your prayers and most of all for your belief in us.

Forever in Gratitude.

Acknowledgments

This thesis would not be possible without the help and support of so many wonderful people who guided me in many ways.

I would like to start by thanking my Guruji first and foremost for all his blessings, guidance and support - RadhaSwami Maharaj.

My sincere thanks and deepest regards to Professor Stratos for serving as my Thesis Director. I could not have written this thesis without his guidance, feedback, expertise and valuable time. Thank you so much Professor Stratos.

I want to thank my Research Advisor, Professor Hongming Wang for her feedback, guidance and for helping me stay on track during the whole process. Thanks also to my Research Advisor, Professor Sylvain Jaume for his inputs and guidance in developing the thesis topic. Thanks also to my Academic Advisors, Nada El-Newahy and Maura B. McGlame, who provided guidance in navigating the degree program and answering my numerous questions.

Thank you to my 4-year-old, who never failed to put a smile on my face through those long days-nights. I am so grateful to my sisters for reviewing my draft multiple times without complaining, for providing me actionable feedback and for their motivation. And finally, thanks to my husband for all his sacrifice and support in making this a reality.

Table of Contents

List of Tables

List of Figures

Chapter I.

Introduction

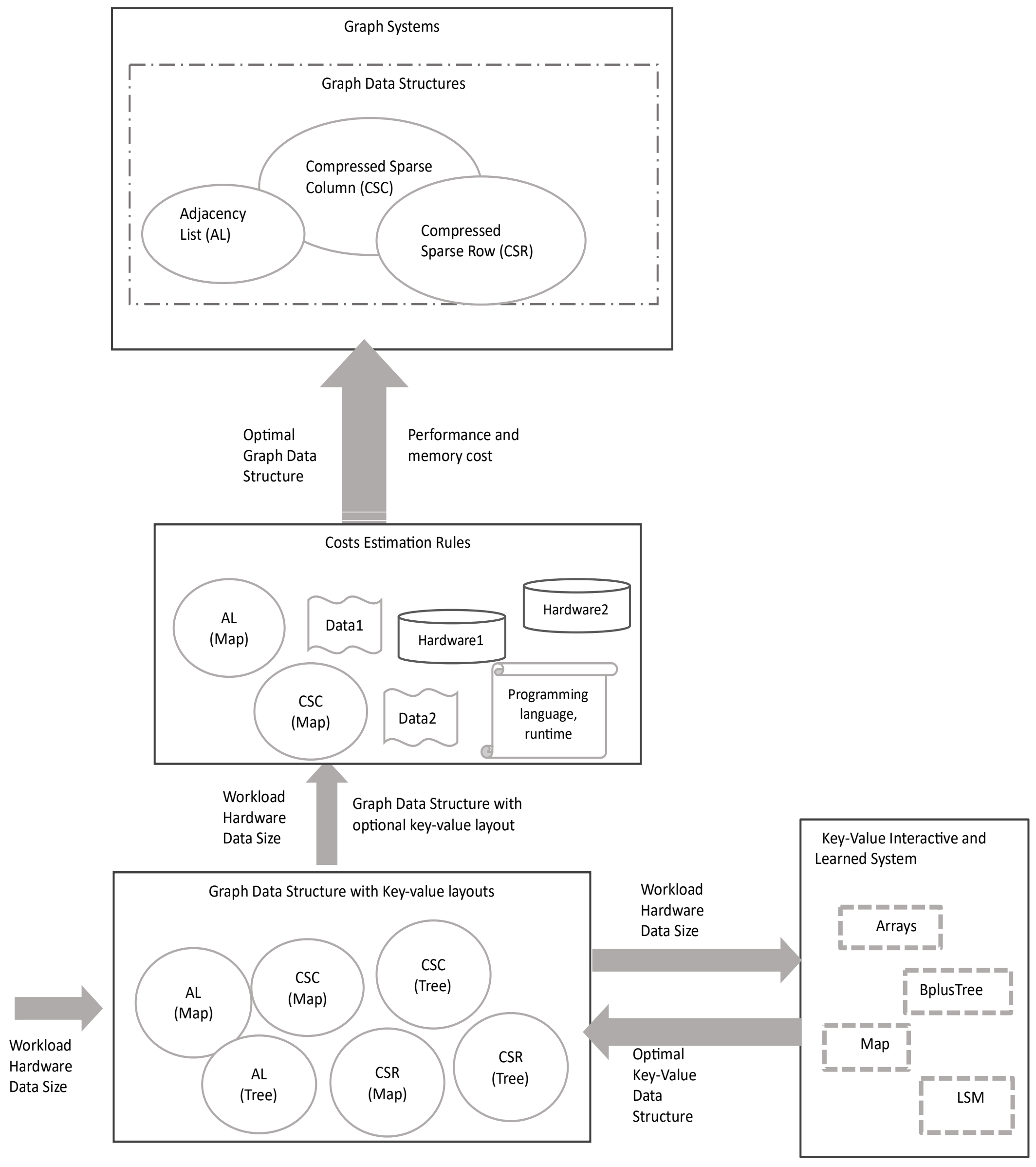
Graph data structures excel in modeling interactions and relationships making them a popular choice in most modern software. Social media Systems, Bioinformatics Systems, Recommendation Systems, Fraud Detection Systems, Network Systems are most naturally designed and modelled as a Graph. Many of the well-known and established data structures have evolved after many years of research, vast commercial adoption and iterative development.

Growth in Data, Diverse Workloads and Specialized Solutions: Graph data structures are critical components of Graph Databases and Graph Frameworks. Growth in data, wide adoption and varied use-cases resulted in the development of specialized databases and frameworks, which differ both in their programming abstractions as well as underlying implementations.

Development Effort and Time: Implementing specialized solutions for each use-case requires huge development effort and time. Even these specialized systems need to be tuned regularly to meet the requirements of changing data patterns and workloads in order to maintain the performance of these systems.

Non-Graph Systems - Emergence of Learned Models and Automatic Design: To handle the growth of data, diverse workloads and hardware, we are seeing increased research in the use of machines learning to build intelligent systems that can automate some of this design process. Systems with understanding of design space and using machine learning to drive the design of data structure, algorithms, access patterns are emerging in the areas of key-value data stores (Idreos et al 2018), databases indexes (Kraska et al 2018) and database systems (Kraska et al 2019).

1.1 Towards Learned Graph Models and Interactive System Design



Graph Systems
Graph Data Structures
Compressed Sparse Column (CSC)
Adjacency List (AL)
Compressed Sparse Row (CSR)
Optimal Graph Data Structure
Performance and memory cost
Costs Estimation Rules
AL (Map)
Data1
Hardware1
Hardware2
CSC (Map)
Data2
Programming language, runtime
Workload Hardware Data Size
Graph Data Structure with optional key-value layout
Graph Data Structure with Key-value layouts
AL (Map)
CSC (Map)
CSC (Tree)
AL (Tree)
CSR (Map)
CSR (Tree)
Workload Hardware Data Size
Workload Hardware Data Size
Optimal Key-Value Data Structure
Key-Value Interactive and Learned System
Arrays
BplusTree
Map
LSM

Figure 1: Leveraging Learned Key-Value Systems to create Learned Graph Systems

Mapping the Design Space: Graph systems are typically more complex and mapping a complete design space ground up takes much longer and is a more involved process. If we could leverage a mapped design system that has an overlap in its design space with Graph Systems, we could rely on that system to map the overlapped design space. The design space specific to Graph systems could be mapped separately. We propose leveraging mapped Key-Value systems because of the overlap seen in the design space. For example, Arrays are used in the default implementation of CSR and Adjacency List, and they also represent a fundamental design primitive for key-value systems.

Cost Estimation: An interactive system allows users to vary inputs to understand how these changes impact performance of the data structure. This requires us to be able to quickly estimate the cost of using a data structure for a given input and workload, without needing to fully create the data structure. We propose starting with a set of rules derived from benchmarking to estimate the cost of workload for a given set of inputs. As the system matures and rules becomes more complete, we can derive a mathematical function which can be used for cost estimation.

1.2 Leverage Key-Value Systems

Graph as Key-Value Pair: In order to enable Graph systems to use key-value system we need to be able to express Graph data structures via key-value data structures, while still keeping the properties of the original data structure in place. For example, the basic implementation of an Adjacency List uses Array of Arrays (or List) for modeling Vertices and Edges of the Graph. If we think of the Vertices of the Graph as Keys, and its Edges as values, we can then use any of the key-value data structures such as – Map, Tree, LSM - to represent a Graph. We use this analogy to model Graph data structures ensuring that the core properties of the original data structure are maintained.

Goals: We present Key-Value Graph Generator which models this concept of Graph as a key-value pair. Our goals are

1. To demonstrate the use of common key-value data structures – Array, Map and BplusTree, in building Adjacency List and Compressed Spare Row (CSR)
2. Benchmark these Data Structures to determine performance impact and memory usage and generate rules for cost estimation.

1.3 Prior Work

This thesis draws inspiration from research in areas of Graph Databases and Frameworks, Learned and Self-designing systems, Graph models and Graph data structures.

Customizable Databases: Research in the area of customizable databases resonates with the most fundamental idea of this thesis, which is to provide optimal graph data structures based for a given input. (Batory et al 1992, Batoory et al 1988, Chaudhuri et al 2000) presents the idea of building customizable database management systems with narrow interfaces to introduce modularity, and to address the issue of rising complexity in building database management systems ground up for each use case.

Learned and Self-Designing Systems: This thesis also learns a great deal from research of Learned Systems. (Idreos et al 2018, Idreos et al 2019, Idreos et al 2019) discuss the idea of design continuum and self-designing key-value data stores, that are optimized for specific workloads. Self-tuning database management (Weikum et al 2002, Chaudhuri et al 2007, Idreos et al 2007, Pavlo et al 2017), Learned Databases indexes (Kraska et al 2018) and Learned database systems (Kraska et al 2019), all discuss how databases can be benefit from using the learned model.

Graph Databases and Frameworks: Pregel (Malewicz et al, 2010), Galois (Pingali et al 2011), Cassovary (Gupta et al 2013), Graphmat (Sundaram et al 2015), Helios (Davoudian, 2019) represent some of the state-of-art Graph Processing system that handle very large Graph datasets efficiently. These frameworks provide different models which were compared (Satish et al 2014) to understand how they scale. Graph Databases research and survey (Güting et al 1994, Angles et al 2008, Dominguez-Sal et al 2010, Angles 2012, Hong et al 2012, Miller et al 2013, Mattson et al 2013, Ghrab et al 2016, Rawat at al 2017, Angles 2018, Besta et al 2019) has a strong connection with this thesis as it brings together various data structures that have been used in building graph database over the years.

Graph Modeling: In this thesis we try to model graph as key-value pair therefore, any research that attempts to model Graph differently is an important resource for this thesis. CombinatorialBLAS (Buluç et al 2011), GraphBLAS (Mattson et al 2013, Bader et al, 2014), introduces a new way to model Graph via common building blocks in linear algebra. LAGraph (Mattson et al, 2019), is a position paper to create high-level Graph Algorithms on top of GraphBLAS.

Graph Data Structures: Graph Data structure research (Saad 1994, Bell et al 2008, Bell et al 2009, Valiyev 2017) and new developments in this space also strongly influence this thesis. (Wheatman et al 2018), introduced a new graph storage layout - Packed Compressed Sparse Row, a Dynamic data structure for representing graph based on packed memory array. It allows for fast inserts, while maintaining good cache for fast searches and traversals. (Macko 2015) presents mutable Compressed Sparse Row to enable using Compressed Sparse Row for write operations.

Chapter II.

Design of Key-Value Graph Generator

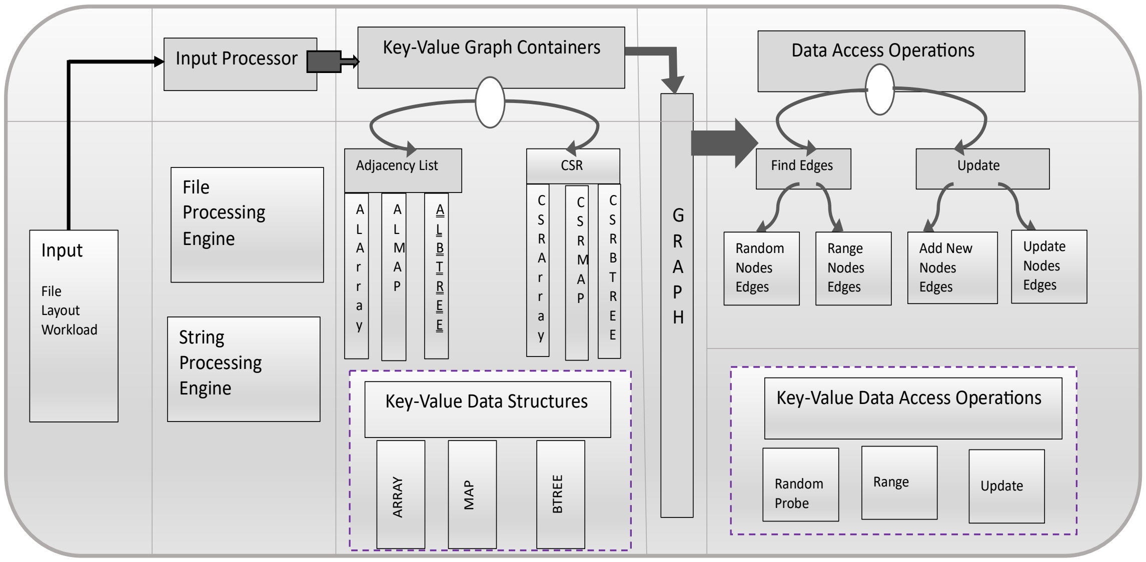


Figure 2: Key-Value Graph Generator.

The components of the Key-Value Graph Generator are captured in Figure 2 (from left to right):

- Input Processor: Input Processor validates user input to ensure it can support the requested graph layout, workload and parse the file format.
- Key-Value Graph Containers: Provides the abstraction for creating Graph using different key-value layouts. We discuss supported containers in the following sections.
- Data Access Operations: This module provides various Data Access operations that can be used to Load, Find and Update the Graph. The framework supports

find operation for all layout of Adjacency List and CSR data structures and provide support for both Random probe and Range search. Write operations update the Graph in one of the following ways (a) Add Edges to an existing node (b) Add new Nodes and edges. Write operations are supported by all data layout of Adjacency List and all CSR layouts **except** Array CSR layout. The Array CSR layout stores the edges and vertices in Array and in order. An update to one vertex can result in update to all vertices and edges of the Graph, making the operation unscalable.

2.1 Key-Value Graph Containers

Graph Container models a Graph storage unit which determines how its vertices and edges are laid out. It defines an API to enable seamless interactions across various concrete implementations. Below we describe how Adjacency List and CSR data structures are modeled using three key-value layouts - Array, Map and BplusTree.

2.2 Adjacency List Containers

An Adjacency list representation for a graph associates each vertex in the graph with its list of edges. An adjacency list is usually implemented as a linked list of arrays or array of arrays. Below we describe how we model Adjacency List using three different key-value containers while maintaining the key property of Adjacency List Vertex associated to its edges.

ArrayALContainer: Using Array Container to model the vertices and edges is the default Adjacency List implementation. Figure 3 shows this implementation for the example graph.

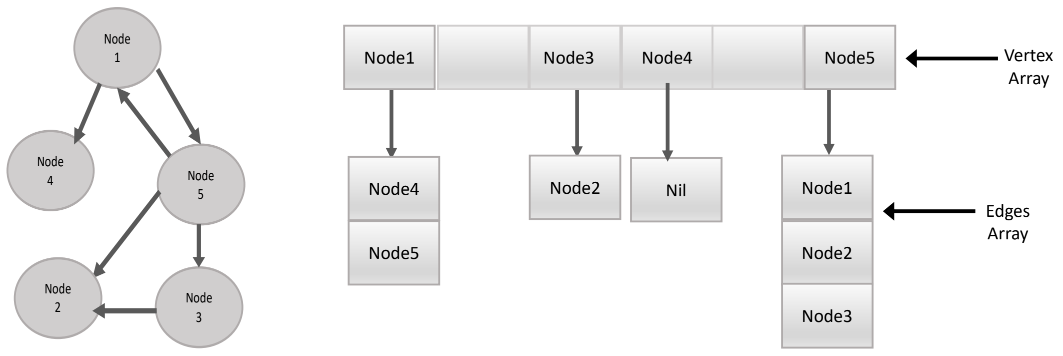


Figure 3: Graph (left) represented using Adjacency List Array Container (right).
The default implementation of Adjacency List uses Array to store its vertices and edges.

HashMapALContainer: When using Map Containers, vertices become keys of the Map and its list of edges the Map values. A key (vertex) with no edges has an empty list as its value. Figure 4 shows this implementation for the example graph.

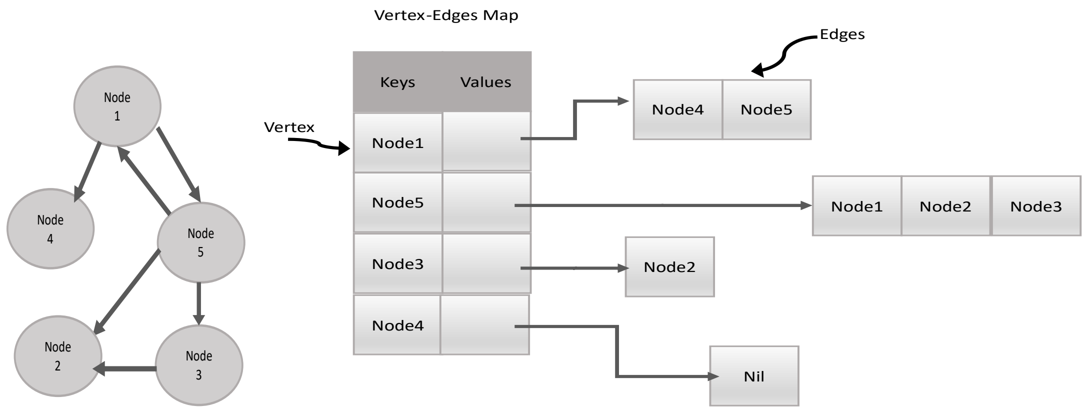


Figure 4: Graph (left) represented using Adjacency List Map Container (right).

Map layout for Adjacency List – Keeping vertex associated to its edges by using vertex as keys and its list of edges as values.

BplusTreeALContainer: To represent an Adjacency List using a BplusTree, we store the vertex in index and leaf nodes, and the list of edges as the data of the leaf nodes. A vertex with no edges points it data to an empty list. Figure 5 shows this implementation for the example graph.

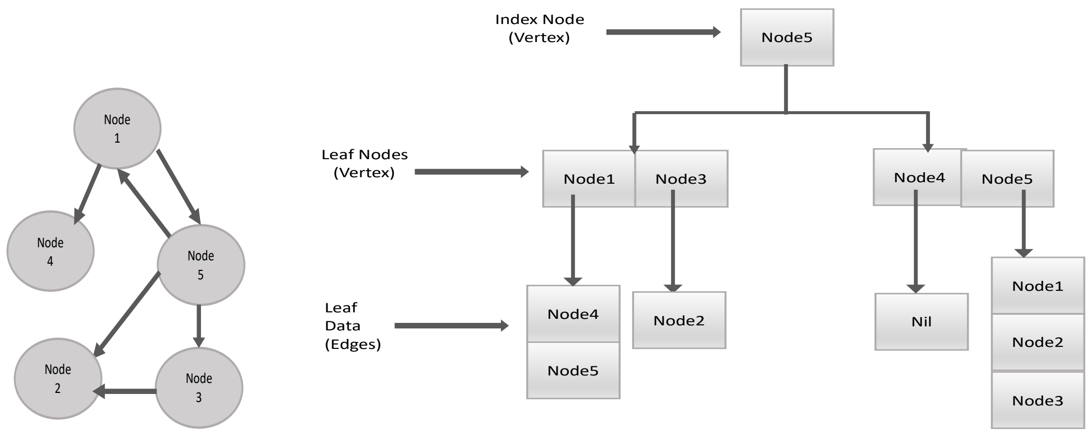


Figure 5: Graph (left) represented using Adjacency List BplusTree Container (right).

BplusTree layout for Adjacency List –vertex association with its edges is maintained through the Leaf Node's value and data. Navigation to Leaf Node is through Index Nodes.

2.3 CSR Containers

A CSR representation for a graph uses at least 2 arrays to stores it vertices and edges. Each vertex has the start index, which points to the index in the edges array where its edges start. Below we describe how we model CSR using three different key-value containers while maintaining the key property of CSR of using different data structure to hold vertices and edges, and the Vertex has information of where in the edges structure its edges live**.**

ArrayCSRContainer: This is the default implementation of CSR, and it uses two Arrays to model the vertices and edges of the Graph. Figure 6 shows this implementation for the example graph.

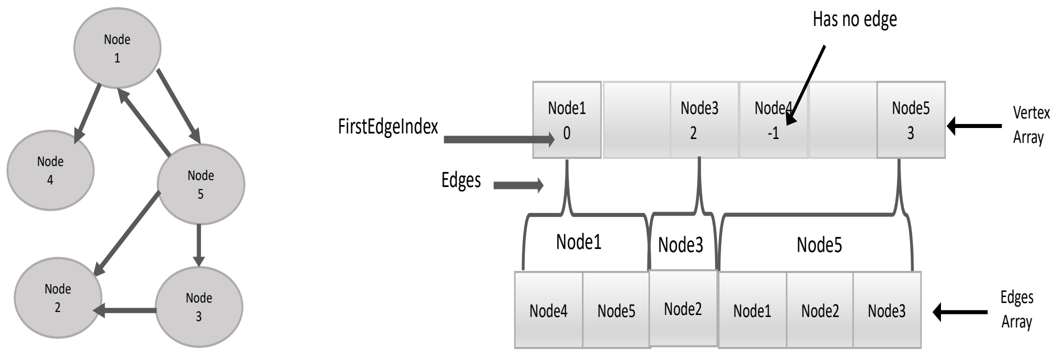


Figure 6: Graph (left) represented using CSR Array Container (right).

The default implementation of CSR uses at least 2 Array to store its vertices and edges.

HashMapCSRContainer: The Map based implementation of CSR uses two maps - one for Vertex and one for list of Edges. The vertex map uses vertex as key, and a String consisting of - VertexId and the first EdgeId - as the value. This value of the vertex map forms the key of the EdgeMap. The values of the edges map are the list of edges themselves. Figure 6 shows this implementation for the example graph.

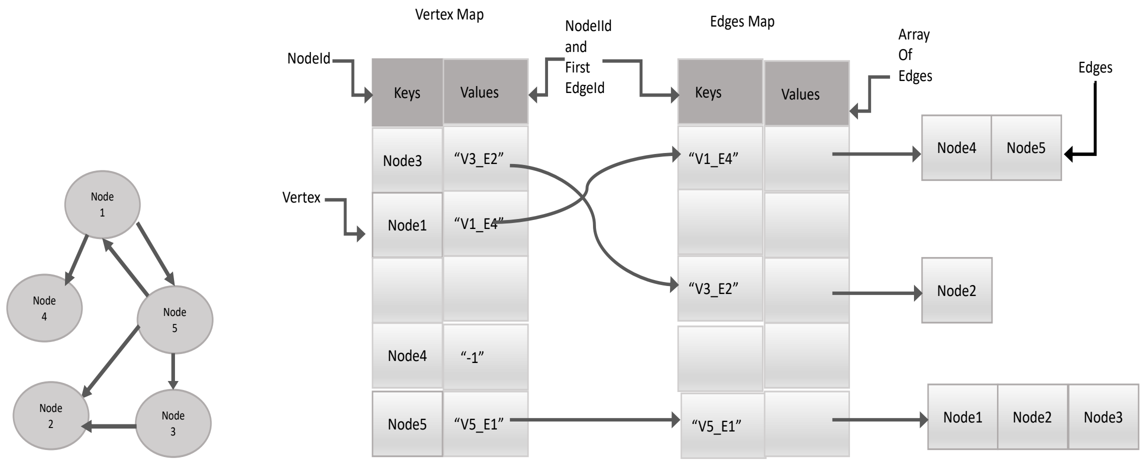


Figure 7: Graph (left) representation using CSR Map Container (right).

Maintains properties of CSR – by using different data structures for vertex and edges. The values of vertex map are the keys of the edge map and the list of edges are the values of edge map.

BplusTreeCSRContainer: To model CSR using BplusTree, we store the vertex in the Index and the Leaf nodes. The edges are in a separate array and the leaf nodes data is the index in the edges array that has the edges for that vertex. Figure 8 shows this implementation for the example graph.

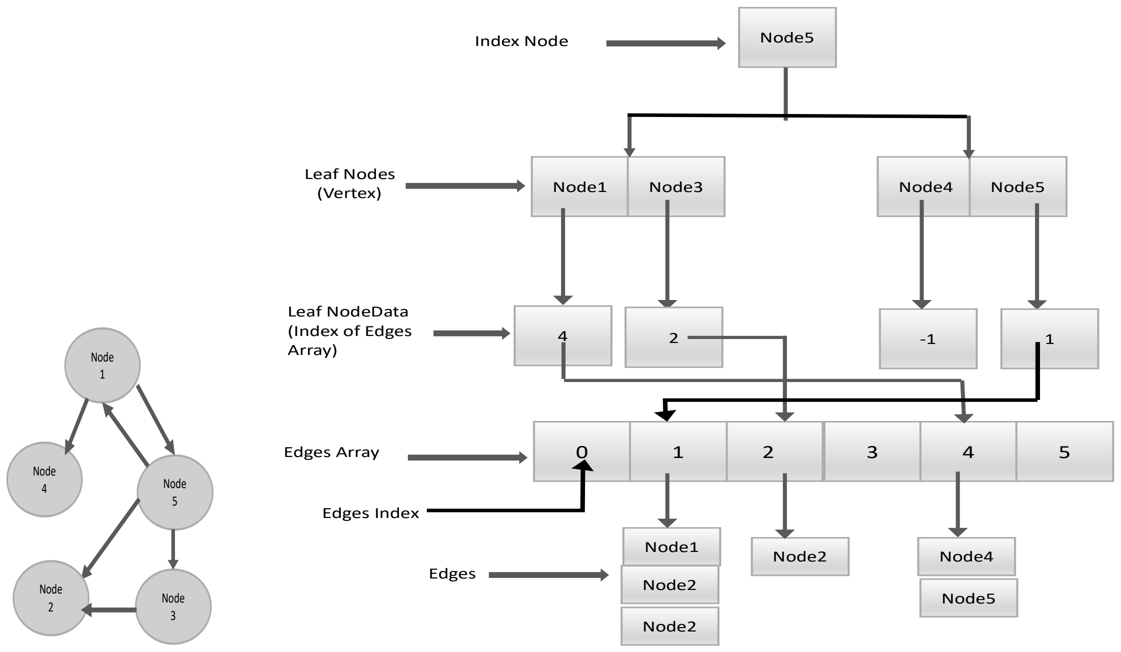


Figure 8: Graph (left) representation using CSR BplusTree Container (right).

Maintaining properties of CSR – by using different data structures for vertex and edges. Index and Leaf nodes of the BplusTree stores the vertex. Data of Leaf nodes is index of edges array that stores the edges for that leaf node (i.e., vertex)

.

Chapter III.

Cost Estimation by Benchmarking

We benchmark data structures to derive a set of rules which will allow us to infer the cost of performance and memory used by a data structure for a given workload, data size, hardware, programming runtime. As these rules grow in complexity and as we continue to benchmark across more diverse set of data, we can use these to train models and build a Learned Cost model which can be expressed through a mathematical function.

Scala: We chose Scala, a language which bring together the concept of object oriented and functional program, to code Key-Value Graph Generator. Ease of development, a strongly typed system and a build system that can support multiple sub-projects using the same build file were some of the key drivers for this choice.

Benchmarking Code running on JVM: Scala relies on the JVM (Java virtual machine) runtime. Benchmarking JVM code using the Stopwatch benchmarking approach may not produce accurate results because of the Just-In-Time (JIT) optimization that can be done by the virtual machine. The many optimizations done by the JVM makes it difficult to ensure that what we are benchmarking is actually what we expect to benchmark.

3.1 Performance benchmarks – Java Microbenchmark Harness (JMH)

JMH (OpenJDK/JMH) is Java harness library for writing benchmarks on the JVM developed as part of the OpenJDK project. JMH provides a foundation for writing and running benchmarks and ensuring results are not diluted due to virtual machine optimizations.

SBT-JMH: In this project JMH library is made available via sbt-jmh, a sbt plugin which brings the jmh tool more natively to the scala ecosystem. All the JMH benchmarks are recorded with the following setup

- 2 warm up iterations.
- 3 iterations
- 1 fork.
- 1 thread
- max jvm heap size - 12 gb

3.2 Memory Footprint – Java Object Layout (JOL)

JOL (OpenJDK/JOL) is a Java library to analyze object layout on JVMs and is developed as part of the OpenJDK project. It provides a more reliable way of measuring the footprint of java objects on the JVM, when compared to taking heap dumps or using other libraries.

3.3 Benchmarking Engine

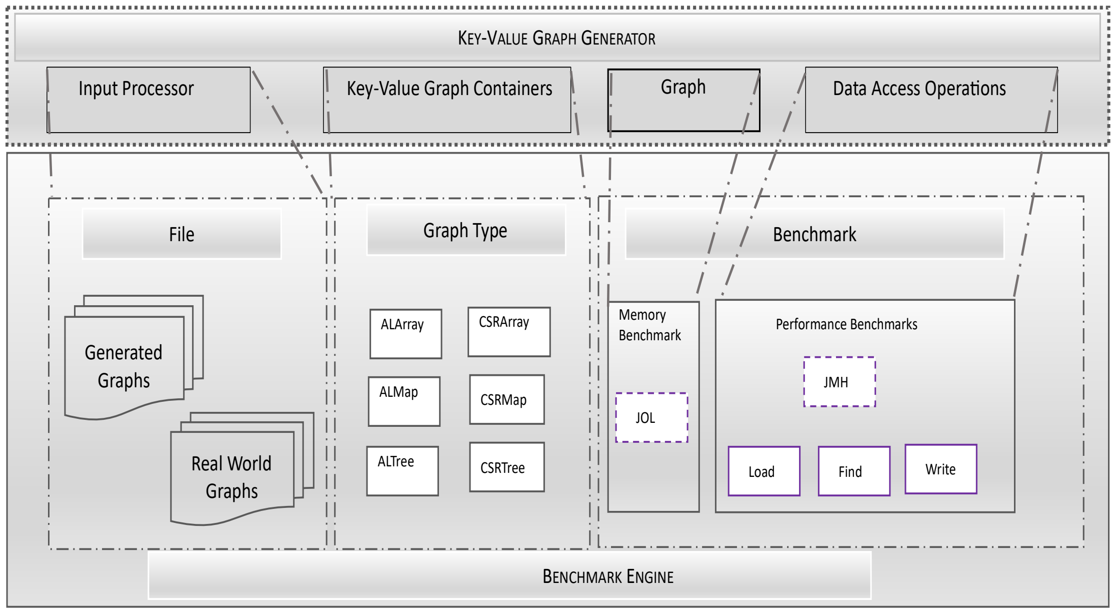


Figure 9: Benchmark Engine.

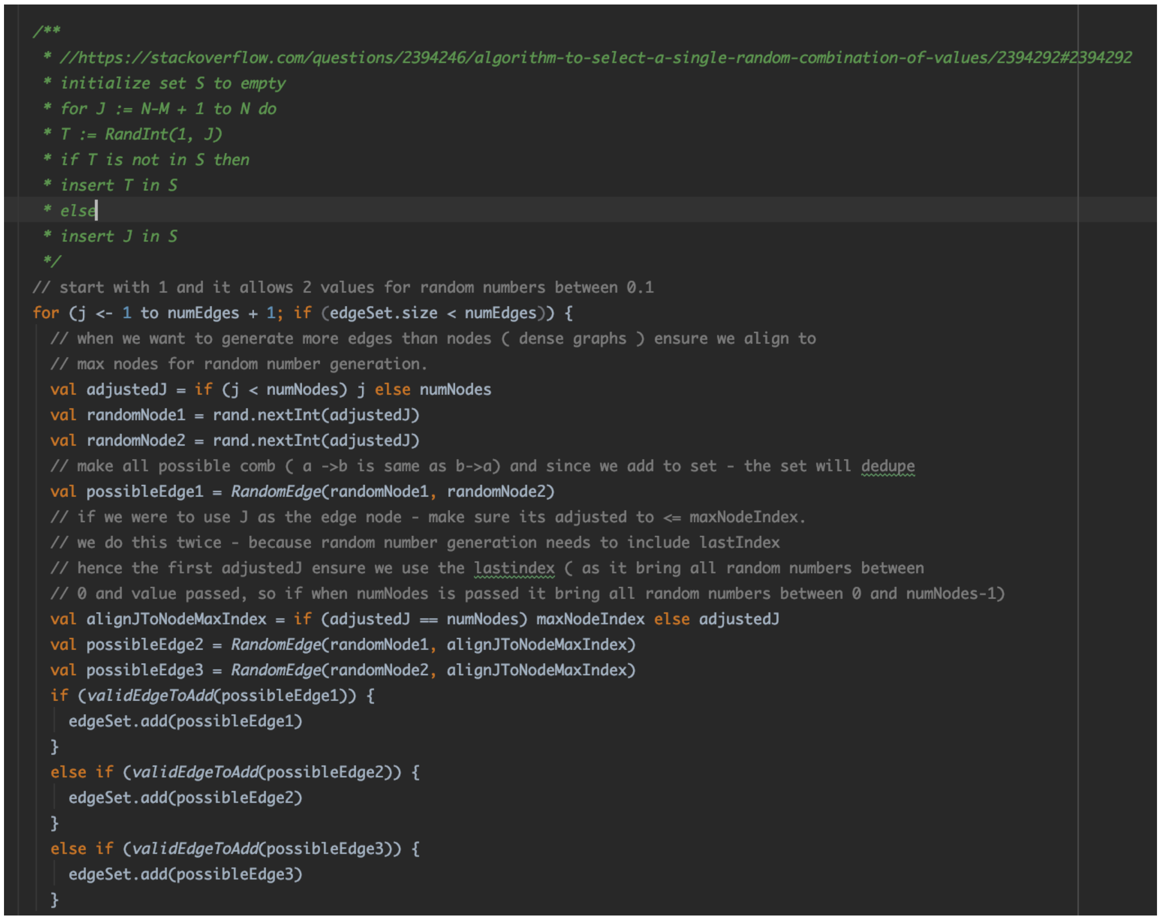
The dotted lines show how components of Benchmarking engine leverages different components of Key-Value Graph Generator to benchmark a workload.

The Benchmarking Engine (as shown in Figure 9) is responsible for the logic needed to interact with the benchmarking libraries, generate test data and to deliver these recorded benchmarks. It provides a consistent API for benchmarking different workloads across different layout of Adjacency List and CSR. The engine depends on the Graph Generator for file processing, graph creation and operations required to run the workload. We benchmark the Key-Value Graph Generator using both Real-world Graphs and Synthetic Graphs. Real-world datasets consist of social network graphs of varying sizes

(Stanford Large Network Dataset Collection). For the Synthetic Graphs the Engine has a built in Graph Generator.

Decoupling and Better Abstractions: In order to provide better abstractions and interaction with these benchmarking libraries, we have modeled Benchmarking Engine and Key-Value Graph Generator as sub-projects in the scala build system. This set up also allows us to manage and use Key-Value Graph Generator on its own. The benchmark engine also benefits from this decoupling as a change to any internal workings and implementation in Graph Generator doesn't require a change in the benchmark engine.

Synthetic Graph Generator: The Benchmarking Engine has a RandomGraphGenerator which is used to generate graphs with variable number of nodes and edges. It is based on Robert Floyd's sampling algorithm to generate edges in the graph uniformly at random



```
/**
 * //https://stackoverflow.com/questions/2394246/algorithm-to-select-a-single-random-combination-of-values/2394292#2394292
 * initialize set S to empty
 * for J := N-M + 1 to N do
 * T := RandInt(1, J)
 * if T is not in S then
 * insert T in S
 * else
 * insert J in S
 */
// start with 1 and it allows 2 values for random numbers between 0.1
for (j <- 1 to numEdges + 1; if (edgeSet.size < numEdges)) {
  // when we want to generate more edges than nodes ( dense graphs ) ensure we align to
  // max nodes for random number generation.
  val adjustedJ = if (j < numNodes) j else numNodes
  val randomNode1 = rand.nextInt(adjustedJ)
  val randomNode2 = rand.nextInt(adjustedJ)
  // make all possible comb ( a ->b is same as b->a) and since we add to set - the set will dedupe
  val possibleEdge1 = RandomEdge(randomNode1, randomNode2)
  // if we were to use J as the edge node - make sure its adjusted to <= maxNodeIndex.
  // we do this twice - because random number generation needs to include lastIndex
  // hence the first adjustedJ ensure we use the lastindex ( as it bring all random numbers between
  // 0 and value passed, so if when numNodes is passed it bring all random numbers between 0 and numNodes-1)
  val alignJToNodeMaxIndex = if (adjustedJ == numNodes) maxNodeIndex else adjustedJ
  val possibleEdge2 = RandomEdge(randomNode1, alignJToNodeMaxIndex)
  val possibleEdge3 = RandomEdge(randomNode2, alignJToNodeMaxIndex)
  if (validEdgeToAdd(possibleEdge1)) {
    edgeSet.add(possibleEdge1)
  }
  else if (validEdgeToAdd(possibleEdge2)) {
    edgeSet.add(possibleEdge2)
  }
  else if (validEdgeToAdd(possibleEdge3)) {
    edgeSet.add(possibleEdge3)
  }
```

Figure 10: Random Graph Generator.

Chapter IV.

Experimental Setup

All Key-Value Graph layouts are evaluated for performance and space usage. Load, Find and Write workloads are triggered to measure performance of the Graph operations with different layouts. For Write operations, we do not include CSR Array layout, due to its inability to scale for write operations. We also exclude Real-World graphs from the write operations.

We run our experiments both on generated graphs and real-world graphs. The goal is to measure

- Scalability - How different graph layouts scale for different Graphs sizes (nodes) and Graph connectivity (edges).
- Suitability – Determine which layouts is better suited for a given workload.

4.1 System

All experiments are run on mac with 4 cores, 2.3GHz clock speed and 16GB of RAM. It has 32K of L1 cache, 256K of L2 cache, and 6000K of L3 cache. All code – Graph Generator and Benchmarking - are written in scala (2.12) and compiled and run using sbt (1.4.4).

4.2 Test Data Files.

In order to study how different layouts scaled for different sizes of nodes and edges, we generated graphs starting with 1K nodes, increasing the nodes by factor of 10 until we got to 10mm nodes. Each of these graphs have edges starting at 1K, increasing by a factor of 10 until we reach 10mm edges. This allows us to test graphs with low to high connectivity for varying graphs sizes. In total we generated 25 different Graphs (Table 1) to capture varying sizes and connectivity. We also use 3 Real-world graphs (Stanford Large Network Dataset Collection) listed below (Table 2).

We ran 7 different kinds of experiments against these generated graphs and Real-world graphs. Six of these were to measure performance of different graph operations, and the last one was to measure the memory usage of the layouts.

Nodes	Edges	FileName
1K	1K	generated1000_1000.txt
1K	10K	generated1000_10000.txt
1K	100K	generated1000_100000.txt
1K	1MM	generated1000_1000000.txt
1K	10MM	generated1000_10000000.txt
10K	1K	generated10000_1000.txt
10K	10K	generated10000_10000.txt
10K	100K	generated10000_100000.txt
10K	1MM	generated10000_1000000.txt
10K	10MM	generated10000_10000000.txt
100K	1K	generated100000_1000.txt
100K	10K	generated100000_10000.txt
100K	100K	generated100000_100000.txt
100K	1MM	generated100000_1000000.txt
100K	10MM	generated100000_10000000.txt
1MM	1K	generated1000000_1000.txt
1MM	10K	generated1000000_10000.txt
1MM	100K	generated1000000_100000.txt

1MM	1MM	generated1000000_1000000.txt
1MM	10MM	generated1000000_10000000.txt
10MM	1K	generated10000000_1000.txt
10MM	10K	generated10000000_10000.txt
10MM	100K	generated10000000_100000.txt
10MM	1MM	generated10000000_1000000.txt
10MM	10MM	generated10000000_10000000.txt

Table 1: Generated Graphs used in Benchmarking

Nodes	Edges	FileName
1,134,890	2,987,624	com-youtube.ungraph.txt
36,692	367,662	email-Enron.txt
403,394	3,387,388	amazon0601.txt
27,770	352,807	cit-HepTh.txt

Table 2: Real-world Graphs used in Benchmarking

4.3 Memory Footprint

We profile the memory used by different graph layouts, in order to understand how each of these scale with increasing number of Nodes and Edges in a Graph. We used 28 files, 25 of these contain generated graphs and 3 of these are the real-world graphs. Each file generates 6 graphs (using 6 layouts) and memory usage in bytes were recorded.

4.4 Performance Benchmarks

Load Times: Next we benchmark the time taken (in ms) to generate these graphs. Once again, all 28 files are used as inputs and each file generates 6 graphs (using 6 layouts). Here we are trying to understand how load times vary across graph types using different layouts.

Find Edges of Random Nodes: We choose a set of 100 random nodes for generated data and real-world data. For generated graphs random nodes are chosen by total number of nodes in a graph. So, a generated graph with same nodes and varying edges uses exactly the same random nodes.

Find Edges of Range of Nodes: We also record the time taken to find edges of a range of nodes. Once again, for generated graphs range is chosen by total number of nodes in a graph. So, a generated graph with same nodes and varying edges uses exactly the same range. When choosing the range, we ensure that that range isn't so large that it covers 90% of total nodes in the graph or so small to make only 5% of the total nodes in the graph. We aim to get a range which is between 50-60% of the total nodes in the graph.

Add Edges to existing nodes in the Graph: We choose a set of 100 random nodes and add 5 edges to each of them. Random nodes are chosen by total number of nodes in a graph. So, a graph with same nodes and varying edges uses exactly the same random nodes.

Add Nodes and Edges: Next, we benchmark adding new nodes and edges to the graph. We create 100 new nodes with 5 edges each. A graph with same nodes and varying edges uses exactly the same set of new nodes.

Chapter V.

Results

5.1 Memory Footprint

CSR Array Layout: We observe that CSR data structure using Array layout (CSRArray) scales best across varying Graph Sizes and Graph connectivity. (Figure 11) shows CSRArray having the lowest memory footprint across all Graphs and also within a Graph (with varying edges). The BplusTree layout for Adjancency List and CSR has the largest memory requirements, followed by Map layouts and Adjacency List Array layout. We also note that the growth in memory with increasing edges is higher for smaller graph size. At 10mm nodes in Graph the memory usage doesn’t vary much as edges are increased from 1K to 10mm.

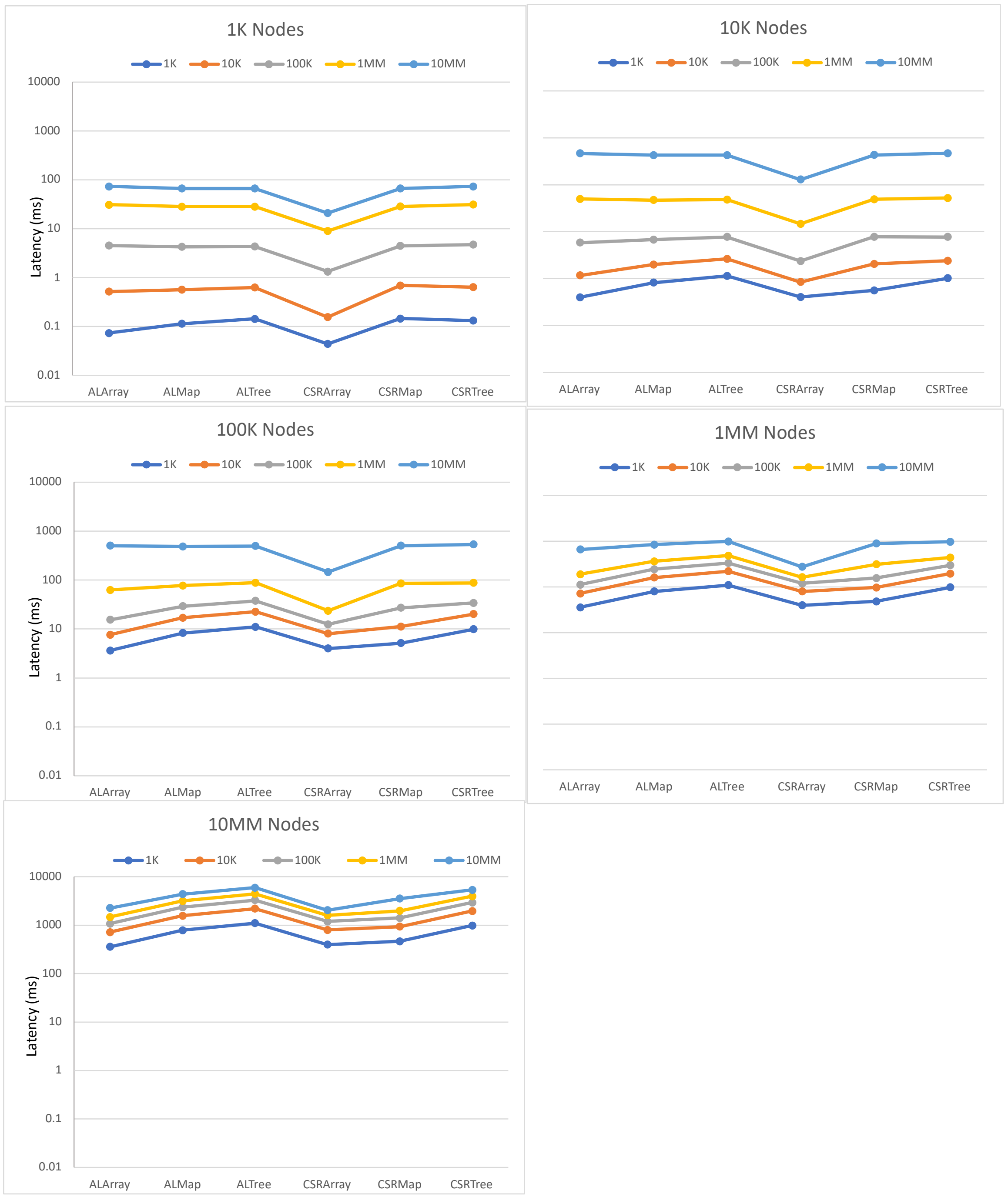


Figure 11: Memory footprint - Generated Graphs.

(in MB) using log scale

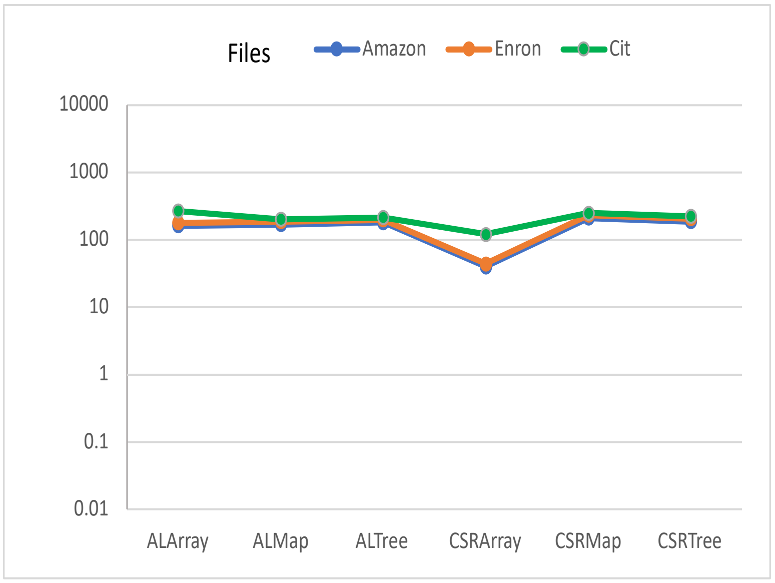


Figure 12: Memory footprint - Real-world Graphs.

(in MB) using log scale.

5.2 Load Times

CSR Array Layout: Once again we observe that CSR data structure using Array layout (CSRArray) takes the least time to load across Graph Sizes and Graph connectivity (Figure 13)**.** BplusTree layout for Adjacency List (ALTree) and CSR (CSRTree) take the most time to load. Each addition of a node or edge in the tree requires tree traversal to identify the right position in the tree where the node or edge will be added, which explains some of the latency we see in layouts using BplusTree. We also note that the connectivity of the Graph (i.e., number of edges) has a larger impact on the latency on smaller Graph size when compared to larger Graphs

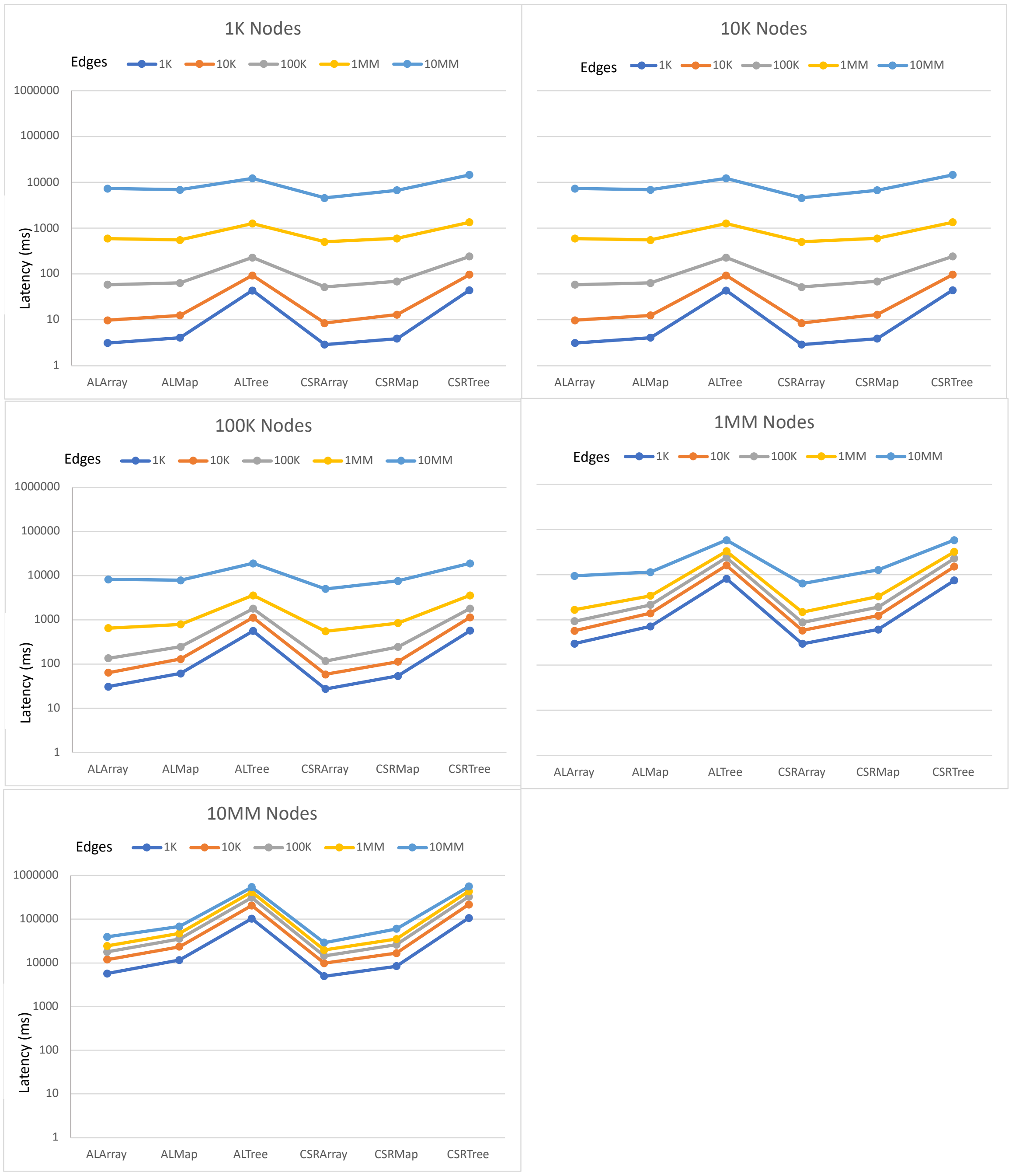


Figure 13: Load Latency - Generated Graphs.

Time to load (in MilliSeconds) using log scale.

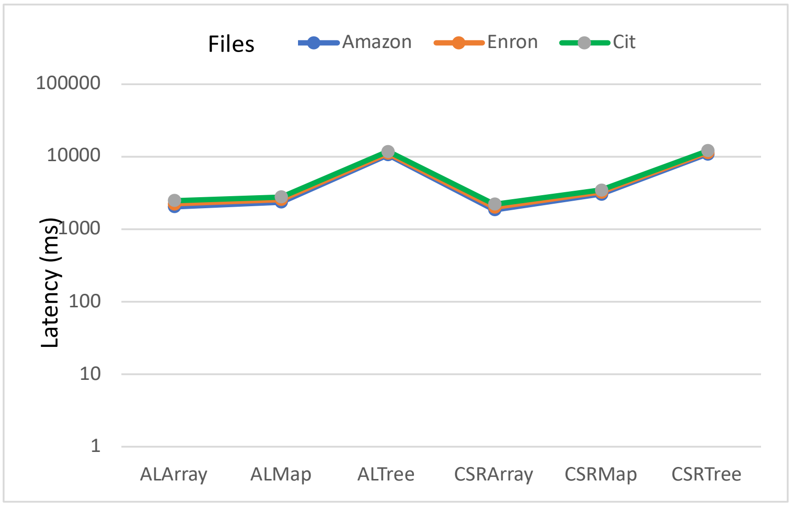


Figure 14: Load Latency - Real-world Graphs.

Time to load (in MilliSeconds) using log scale.

5.3 Find Edges – Random Nodes

Adjacency List – Array Layout: When searching for edges of random nodes in a graph the Adjacency List Array (ALArray) layout performance best across graph sizes and graph connectivity (Figure 15). The Map layout for Adjacency List (ALMap) also scales well and requires almost constant time to find the edges on a graph, even as the number of edges vary. CSR Array layout has a varying performance, with higher cost on smaller Graphs with high connectivity and lower cost for larger Graphs.

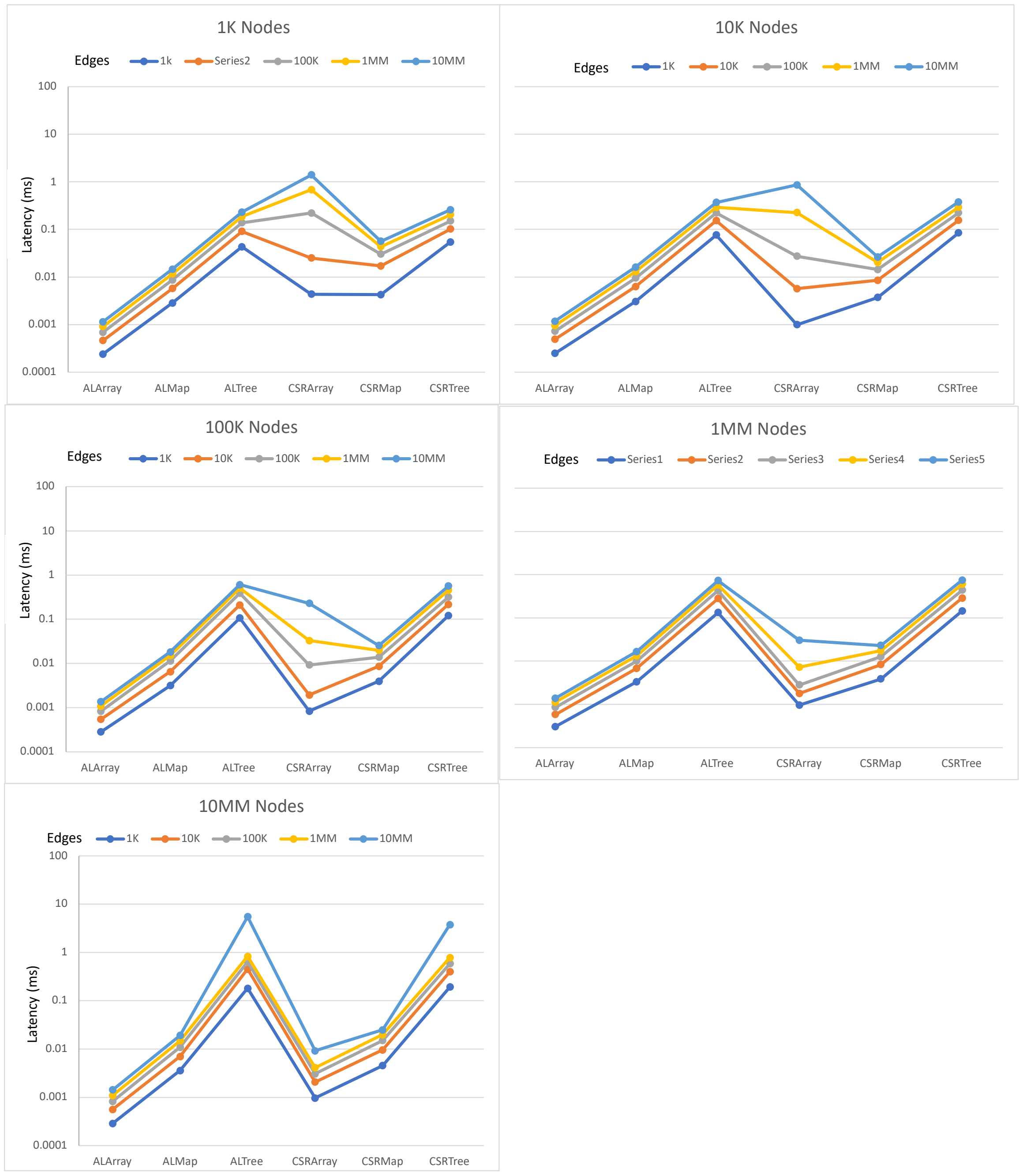


Figure 15: Find Edges Random Nodes - Generated Graphs.

Time to find 100 random nodes (in MilliSeconds) using log scale.

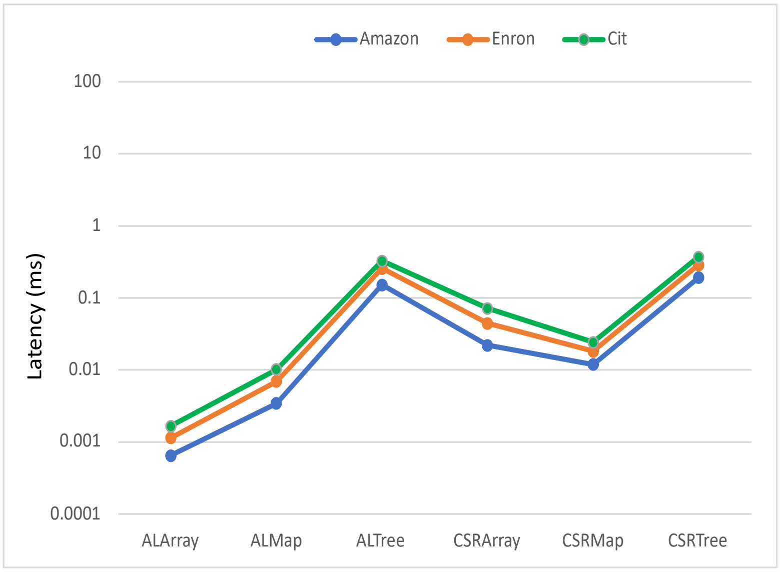


Figure 16: Find Edges Random Nodes - Real-world Graphs.

Time to find 100 random nodes (in MilliSeconds) using log scale.

5.4 Find Edges – Range of Nodes

BplusTree Layout - Adjacency List and CSR: Searching for edges of Range of nodes scales best with a BplusTree layout of both Adjacency List and CSR data structures (Figure 17), doing noticeably better for smaller graphs. Adjacency List Array also scales well and does slightly better than the BplusTree layout for larger graphs. CSR Array shows varying performance across graph sizes and connectivity.

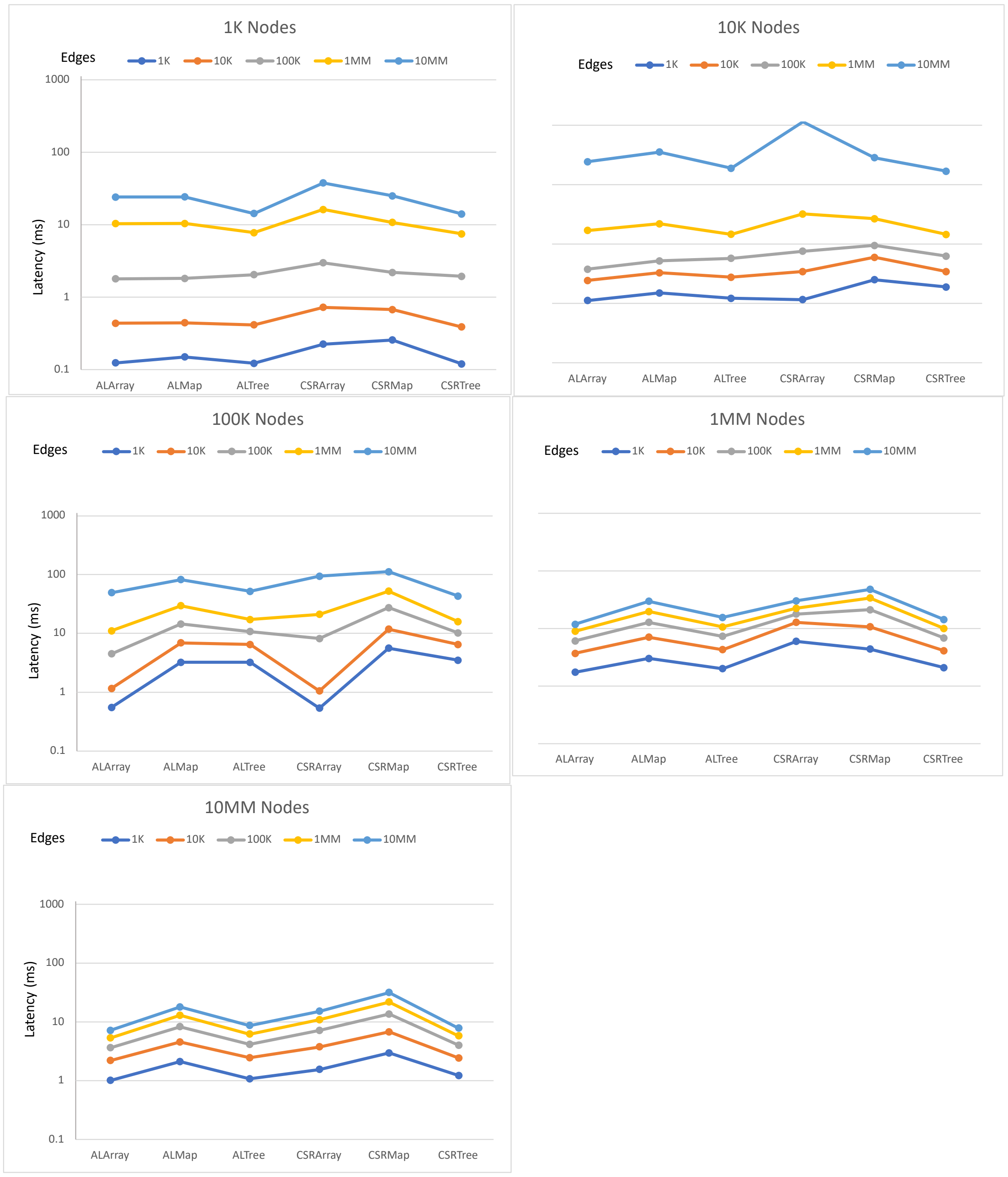


Figure 17: Range Search – Generated Graphs.

Time to find a range of nodes (in MilliSeconds) using log scale.

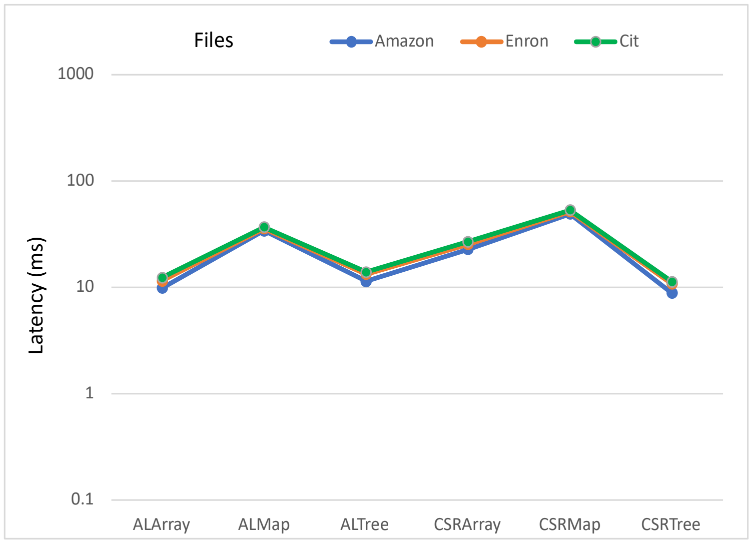


Figure 18: Range Search - Real-world Graphs.

Time to find a range of nodes (in MilliSeconds) using log scale.

5.5 Update – Add Nodes and Edges

Map Layout - Adjacency List and CSR: We observe that the update operation of adding new nodes with edges to the Graph, was handled best by Map layouts of both Adjacency List and CSR data structures (Figure 19). BplusTree layouts displays stable performance without any sudden spikes across Graph size and connectivity. Adjacency List Array layout shows spikes for small Graphs and then it stabilizes as the size of the Graph increases.

5.6 Update – Add Edges to existing Nodes

There is no single layout for all sizes and connectivity: Our results (Figure 20) show that the size of the Graph and Graph Connectivity greatly impacts which layout is more efficient. For example, we notice that Map layout for Adjacency List does well for Graph with 1MM Nodes but not for Graph with 10MM nodes.

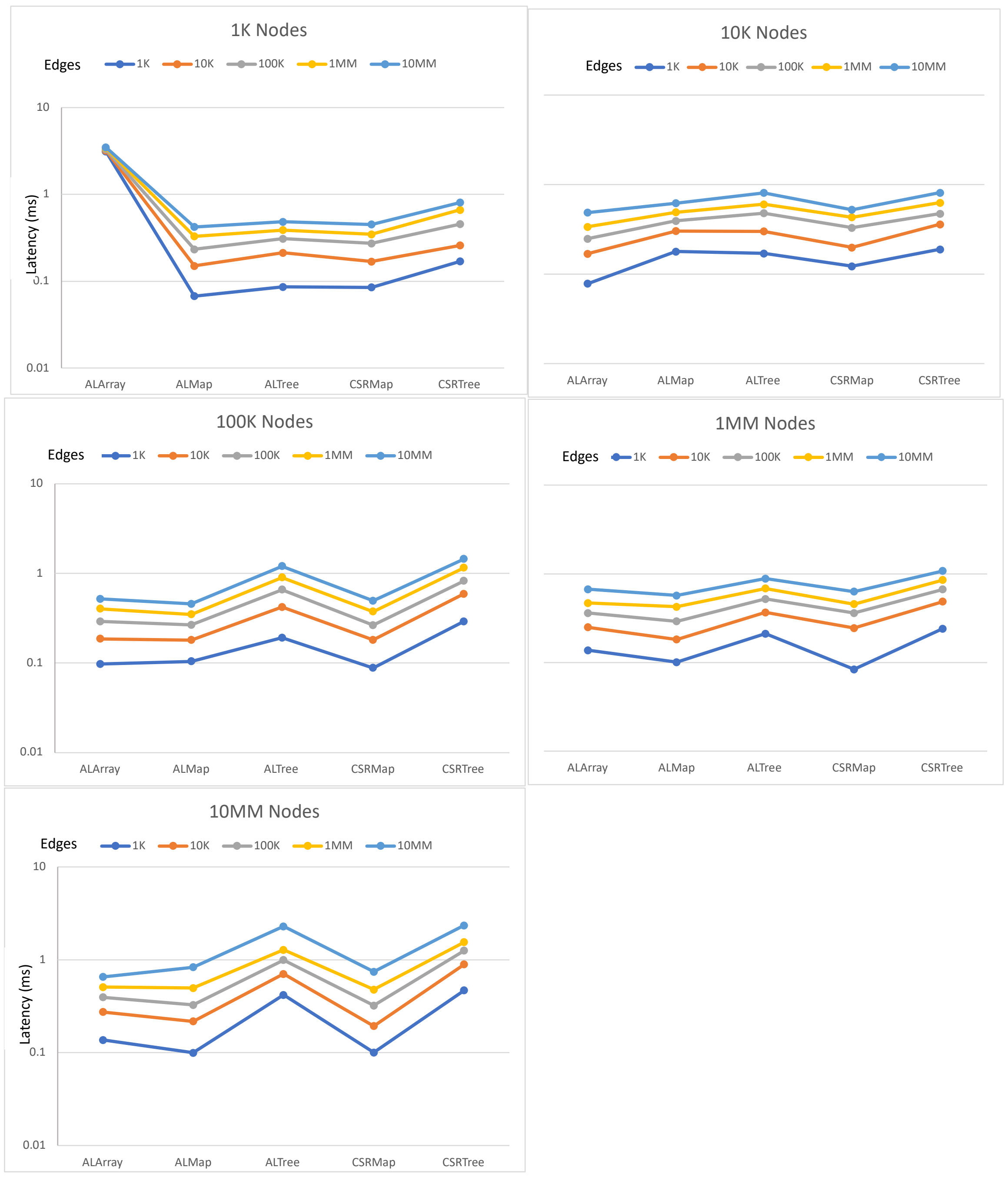


Figure 19: Add Nodes and Edges - Generated Graphs.

Time to add 100 new nodes with 5 edges each (in MilliSeconds) using log scale.

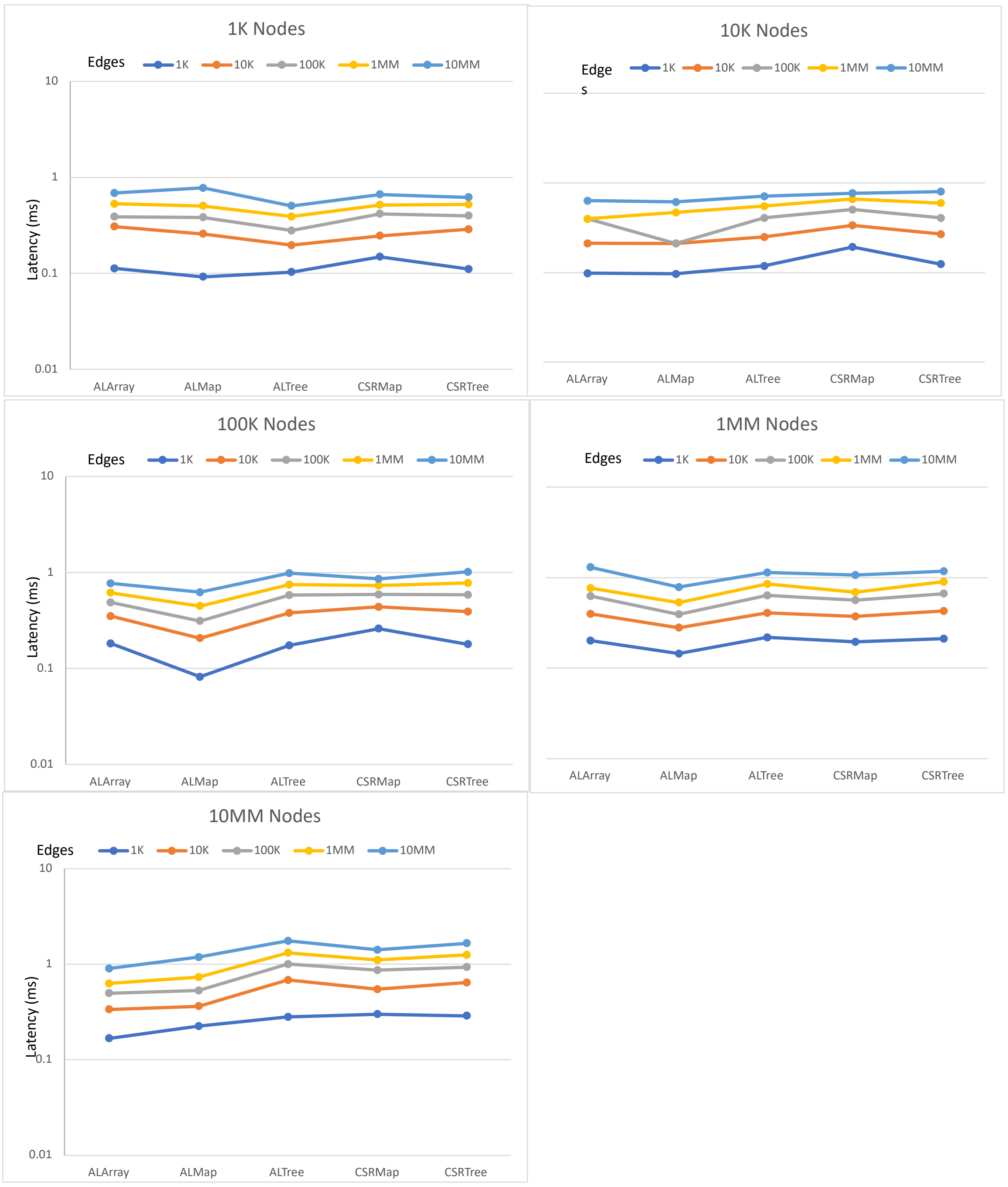


Figure 20: Add Edges - Generated Graphs.

Time to update edges of 100 random nodes (in MilliSeconds) using log scale.

.

5.7 Results Summary and Cost Estimation Rules.

The table below summarizes observations from all benchmarking experiments mentioned in the earlier sections. These rules can be used to find optimal Graph data structure in the context of performance and memory costs for a given set of inputs. They also allow us to reason about using a layout and a data structure without actually building the data structure or running the workload. For example, when we are looking for low memory and load costs – a CSR with Array Layout would be an optimal choice, and if the requirement is for most efficient way to find edges of random nodes, we would go with Adjacency List with Array or Map Layout.

Cost	Optimal Layout
Memory Footprint	CSR - Array Layout
Load Latency	CSR – Array Layout
Find Edges Latency –Random Nodes	Adjacency List – Array Layout, Adjacency List – Map Layout
Find Edges Latency–Range of Nodes	Adjacency List - Bplus Tree, CSR – Bplus Tree
Update Latency– Add New Nodes and Edges	Adjacency List – Map Layout, CSR – Map Layout
Update Latency – Add Edges to Existing Nodes	No single layout that is best for all sizes and connectivity

Table 3: Derived Cost Estimation Rules

Chapter VI.

Conclusion

6.1 Summary

In our pursuit for interactive and automatic graph data systems, Key-Value Graph Generator is a good starting point. It demonstrates the use of key-value approach in designing Graph Data Structures. We model Adjacency List and Compressed Sparse Row (CSR), using Array, Map and BplusTree. Performance and Memory benchmarks of these data structures confirm that different workloads can benefit from using different layouts. Cost estimation rules derived from these benchmarks, allow us to reason about data structures without building them and having to running a workload.

6.2 Future Work

Key-Value Graph Generator has the potential of working well with interactive and learned key-value systems, as it has been designed with this objective in mind. Using an interactive and learned key-value system with Key-Value Graph Generator is a natural extension to this project and a definitive future line of work. We would start by creating an API that can be used for communicating between the interactive Key-Value System and the Key-Value Graph Generator. We would also need interfaces in the Key-Value Generator which can pass the user requirement to the interactive Key-Value system and use the output from that system to create the recommended Graph Data Structure. We can then validate this recommendation using the cost estimation rules from our benchmarking as described in Section 5.7. Once we establish a working version of interactive Graph Data Structure Generator, we need to increase the surface area of modeling options available by adding more key-value constructs and Graph data structures to the framework.

Benchmarking performance with more datasets, graph algorithms (Page Rank, DFS), hardware, programming language (Rust or C) to understand how these may affect costs. All of these inputs will not only help the Graph Data Structure Generator to be more comprehensive, but also drive the creation of learned cost model and more reliable cost estimations.

References

Angles, R., & Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys (CSUR)*, *40*(1), 1-39.

Angles, R. (2012, April). A comparison of current graph database models. In *2012 IEEE 28th International Conference on Data Engineering Workshops* (pp. 171-177). IEEE.

Angles, R., & Gutierrez, C. (2018). An introduction to graph data management. In *Graph Data Management* (pp. 1-32). Springer, Cham.

Bader, D., Buluç, A., Gilbert, J., Gonzalez, J., Kepner, J., & Mattson, T. (2014, July). The Graph BLAS effort and its implications for Exascale. In *SIAM Workshop on Exascale Applied Mathematics Challenges and Opportunities (EX14)*

Batory, D., & O'malley, S. (1992). The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *1*(4), 355-398.

Batoory, D. S., Barnett, J. R., Garza, J. F., Smith, K. P., Tsukuda, K., Twichell, B. C., & Wise, T. E. (1988). GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering*, *14*(11), 1711-1730.

Bell, N., & Garland, M. (2008). *Efficient sparse matrix-vector multiplication on CUDA* (Vol. 2, No. 5). Nvidia Technical Report NVR-2008-004, Nvidia Corporation.

Bell, N., & Garland, M. (2009, November). Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis* (pp. 1-11).

Besta, M., Peter, E., Gerstenberger, R., Fischer, M., Podstawski, M., Barthels, C., ... & Hoefler, T. (2019). Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *arXiv preprint arXiv:1910.09017*.

Buluç, A., & Gilbert, J. R. (2011). The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, *25*(4), 496-509.

Chaudhuri, S., & Weikum, G. (2000, September). Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *VLDB* (pp. 1-10).

Chaudhuri, S., & Narasayya, V. (2007, September). Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases* (pp. 3-14).

Davoudian, A. (2019, June). Helios: An adaptive and query workload-driven partitioning framework for distributed graph stores. In *Proceedings of the 2019 International Conference on Management of Data* (pp. 1820-1822).

Dominguez-Sal, D., Urbón-Bayes, P., Giménez-Vanó, A., Gómez-Villamor, S., Martínez-Bazan, N., & Larriba-Pey, J. L. (2010, July). Survey of graph database performance on the hpc scalable graph analysis benchmark. In *International Conference on Web-Age Information Management* (pp. 37-48). Springer, Berlin, Heidelberg.

Ghrab, A., Romero, O., Skhiri, S., Vaisman, A., & Zimányi, E. (2016). Grad: On graph database modeling. arXiv preprint arXiv:1602.00503.

Gupta, P., Goel, A., Lin, J., Sharma, A., Wang, D., & Zadeh, R. (2013, May). Wtf: The who to follow service at twitter. In Proceedings of the 22nd international conference on World Wide Web (pp. 505-514).

Güting, R. H. (1994, September). GraphDB: Modeling and querying graphs in databases. In *VLDB* (Vol. 94, pp. 12-15).

Hong, S., Chafi, H., Sedlar, E., & Olukotun, K. (2012, March). Green-Marl: a DSL for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems* (pp. 349-362).

Idreos, S., Kersten, M. L., & Manegold, S. (2007, January). Database Cracking. In CIDR (Vol. 7, pp. 68-78).

Idreos, S., Zoumpatianos, K., Hentschel, B., Kester, M. S., & Guo, D. (2018). The Internals of The Data Calculator. *arXiv preprint arXiv:1808.02066*.

Idreos, S., Dayan, N., Qin, W., Akmanalp, M., Hilgard, S., Ross, A., ... & Zhu, Z. (2019, January). Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *CIDR*.

Idreos, S., & Kraska, T. (2019, June). From auto-tuning one size fits all to self-designed and learned data-intensive systems. In *Proceedings of the 2019 International Conference on Management of Data* (pp. 2054-2059).

Kraska, T., Alizadeh, M., Beutel, A., Chi, H., Kristo, A., Leclerc, G., ... & Nathan, V. (2019, January). Sagedb: A learned database system. In *CIDR*.

Kraska, T., Beutel, A., Chi, E. H., Dean, J., & Polyzotis, N. (2018, May). The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data* (pp. 489-504).

Macko, P. (2015). Llama: A persistent, mutable representation for graphs (Doctoral dissertation).

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010, June). Pregel: a system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (pp. 135-146).

Mattson, T., Bader, D., Berry, J., Buluc, A., Dongarra, J., Faloutsos, C., ... & Yoo, A. (2013, September). Standards for graph algorithm primitives. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)* (pp. 1-2). IEEE.

Mattson, T., Davis, T. A., Kumar, M., Buluc, A., McMillan, S., Moreira, J., & Yang, C. (2019, May). LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (pp. 276-284). IEEE.

Miller, J. J. (2013, March). Graph database applications and concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA* (Vol. 2324, No. 36).

OpenJDK/JOL, https://openjdk.java.net/projects/code-tools/jol/

OpenJDK/JMH, https://openjdk.java.net/projects/code-tools/jmh/

Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., ... & Zhang, T. (2017, January). Self-Driving Database Management Systems. In CIDR (Vol. 4, p. 1).

Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M. A., Kaleem, R., ... & Sui, X. (2011, June). The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (pp. 12-25).

Rawat, D. S., & Kashyap, N. K. (2017). Graph database: a complete GDBMS survey. Int. J, 3, 217-226.

Saad, Y. (1994). SPARSKIT: a basic tool kit for sparse matrix computations-Version 2.

Satish, N., Sundaram, N., Patwary, M. M. A., Seo, J., Park, J., Hassaan, M. A., ... & Dubey, P. (2014, June). Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (pp. 979-990).

Stanford Large Network Dataset Collection, https://snap.stanford.edu/data/

Sundaram, N., Satish, N. R., Patwary, M. M. A., Dulloor, S. R., Vadlamudi, S. G., Das, D., & Dubey, P. (2015). Graphmat: High performance graph analytics made productive. *arXiv preprint arXiv:1503.07241*.

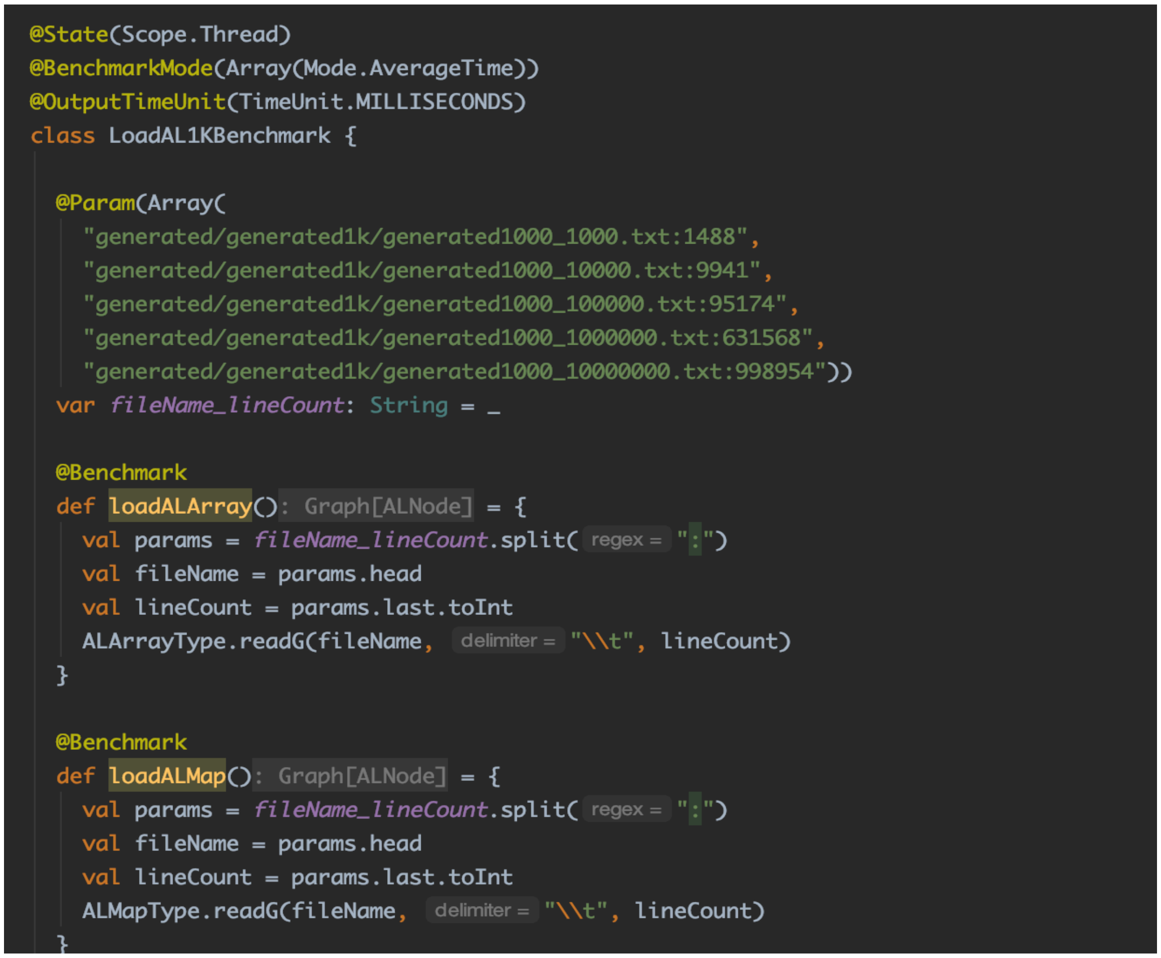
Valiyev, M. (2017). Graph Storage: How good is CSR really?. dated Dec, 10, 8.

Weikum, G., Moenkeberg, A., Hasse, C., & Zabback, P. (2002, January). Self-tuning database technology and information services: from wishful thinking to viable engineering. In VLDB'02: Proceedings of the 28th International Conference on Very Large Databases (pp. 20-31). Morgan Kaufmann.

Wheatman, B., & Xu, H. (2018, September). Packed Compressed Sparse Row: A Dynamic Graph Representation. In 2018 IEEE High Performance extreme Computing Conference (HPEC) (pp. 1-7). IEEE.

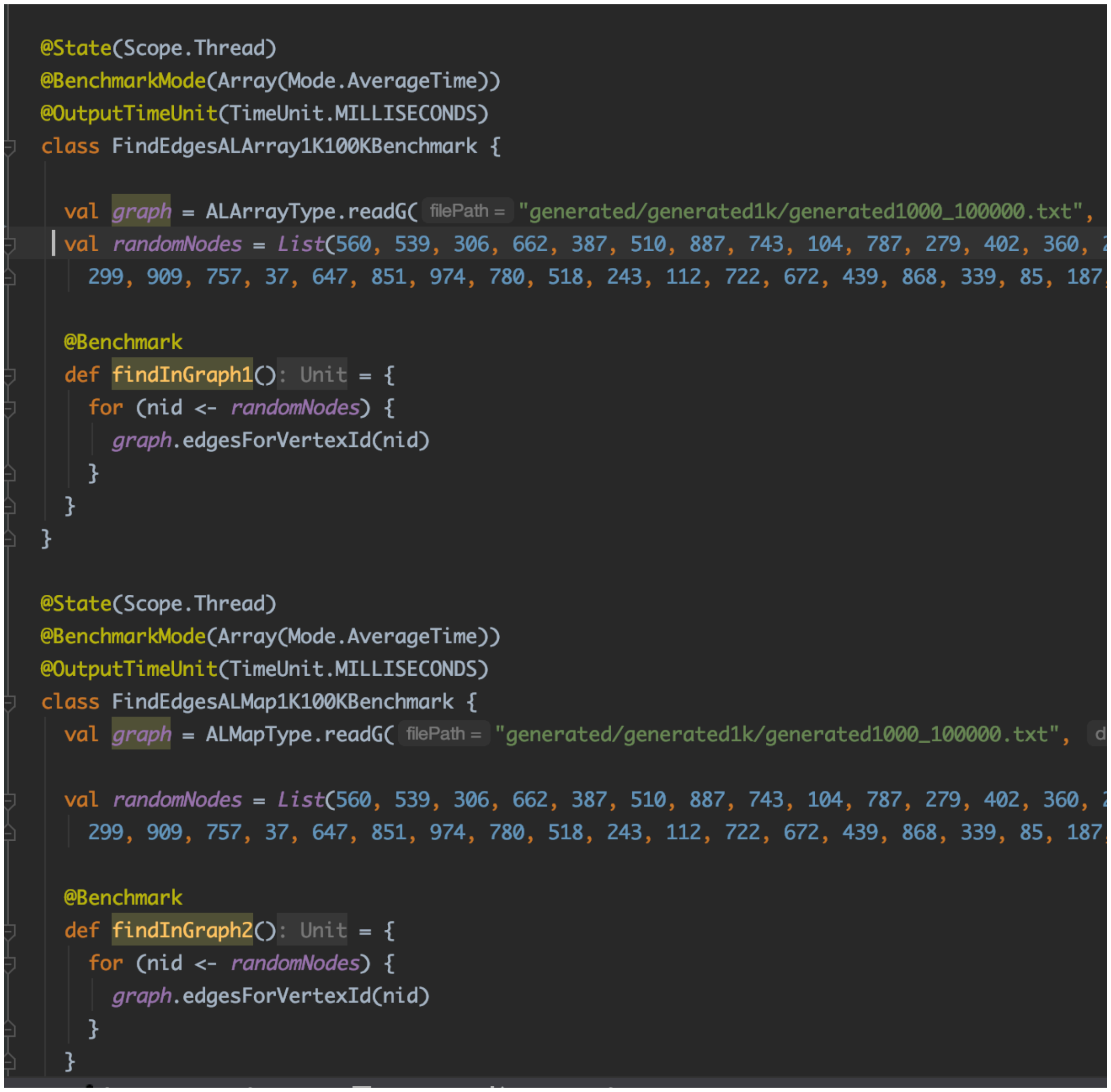
Appendix

All source code can be found on github - https://github.com/RshaHvd/graphdatastructure. In this appendix we provide code sample used during benchmarking using JMH and JOL libraries. These samples show how benchmarking using these libraries is different from the usual Stopwatch benchmarking approach.



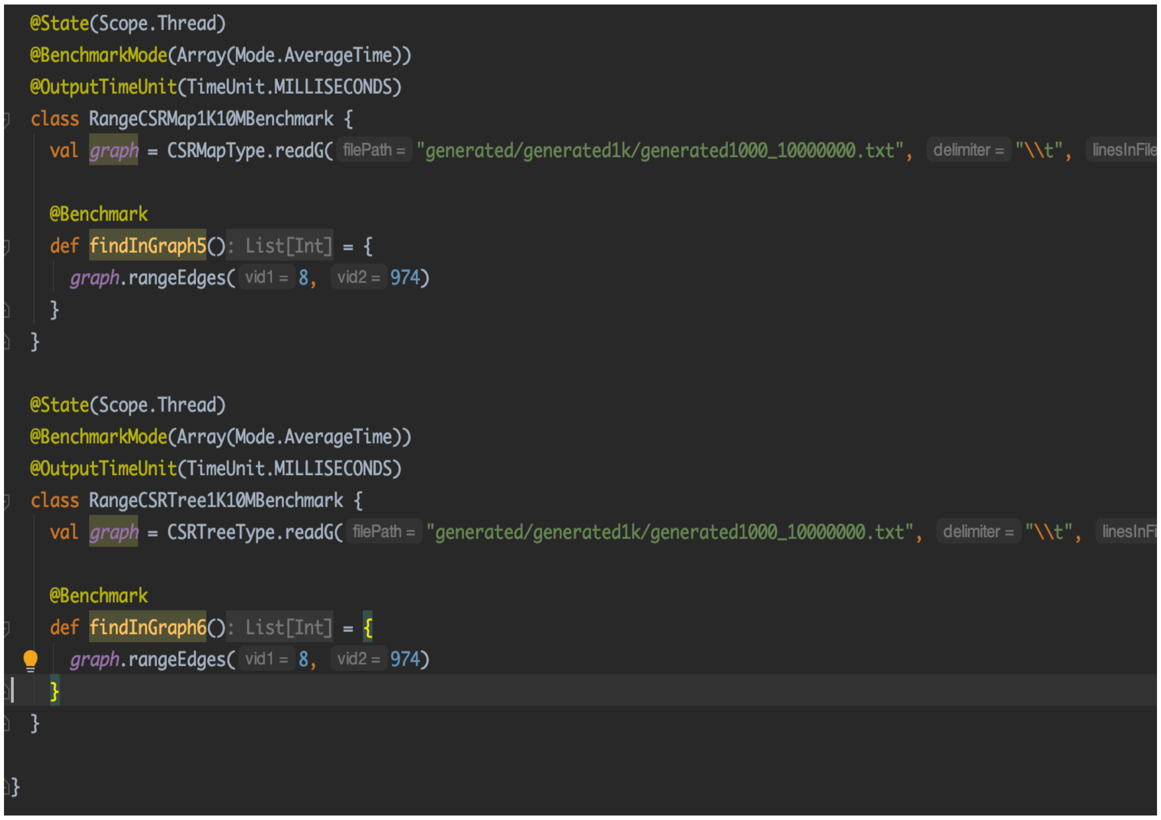
```
@State(Scope.Thread)
@BenchmarkMode(Array(Mode.AverageTime))
@OutputTimeUnit(TimeUnit.MILLISECONDS)
class LoadAL1KBenchmark {

  @Param(Array(
    "generated/generated1k/generated1000_1000.txt:1488",
    "generated/generated1k/generated1000_10000.txt:9941",
    "generated/generated1k/generated1000_100000.txt:95174",
    "generated/generated1k/generated1000_1000000.txt:631568",
    "generated/generated1k/generated1000_10000000.txt:998954"))
  var fileName_lineCount: String = _

  @Benchmark
  def loadALArray(): Graph[ALNode] = {
    val params = fileName_lineCount.split( regex = ":")
    val fileName = params.head
    val lineCount = params.last.toInt
    ALArrayType.readG(fileName,  delimiter = "\\t", lineCount)
  }

  @Benchmark
  def loadALMap(): Graph[ALNode] = {
    val params = fileName_lineCount.split( regex = ":")
    val fileName = params.head
    val lineCount = params.last.toInt
    ALMapType.readG(fileName,  delimiter = "\\t", lineCount)
  }
```

Load benchmark code snippet for Adjacency List Containers.



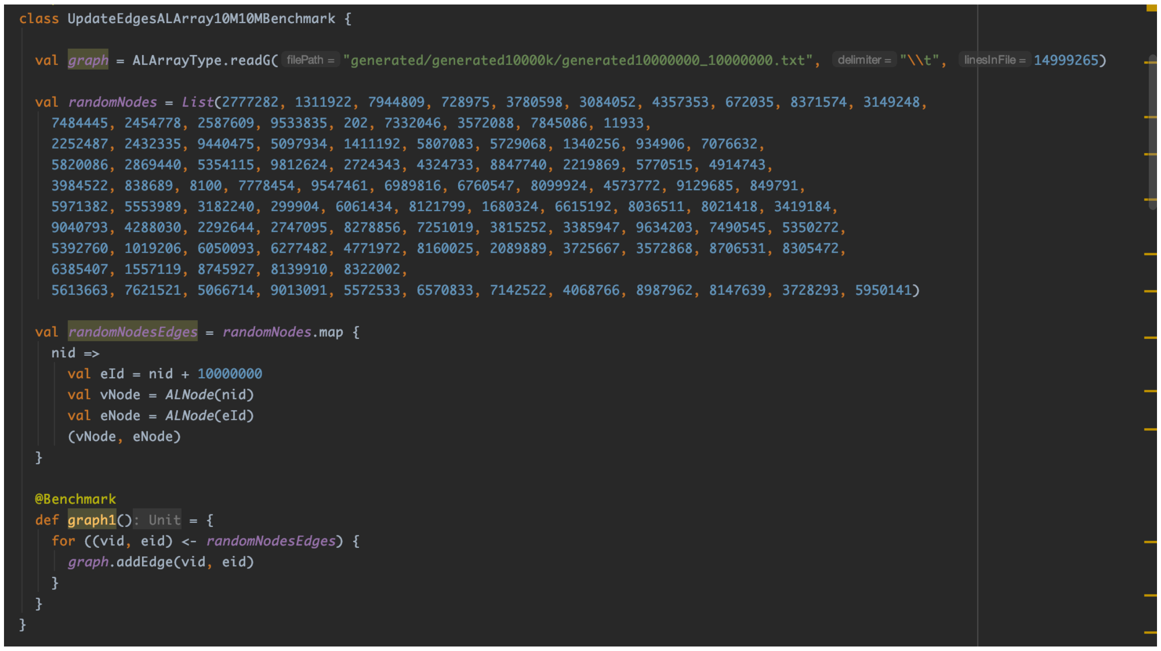
```
@State(Scope.Thread)
@BenchmarkMode(Array(Mode.AverageTime))
@OutputTimeUnit(TimeUnit.MILLISECONDS)
class FindEdgesALArray1K100KBenchmark {

  val graph = ALArrayType.readG( filePath = "generated/generated1k/generated1000_100000.txt",
  val randomNodes = List(560, 539, 306, 662, 387, 510, 887, 743, 104, 787, 279, 402, 360,
    299, 909, 757, 37, 647, 851, 974, 780, 518, 243, 112, 722, 672, 439, 868, 339, 85, 187

  @Benchmark
  def findInGraph1(): Unit = {
    for (nid <- randomNodes) {
      graph.edgesForVertexId(nid)
    }
  }
}

@State(Scope.Thread)
@BenchmarkMode(Array(Mode.AverageTime))
@OutputTimeUnit(TimeUnit.MILLISECONDS)
class FindEdgesALMap1K100KBenchmark {
  val graph = ALMapType.readG( filePath = "generated/generated1k/generated1000_100000.txt",

  val randomNodes = List(560, 539, 306, 662, 387, 510, 887, 743, 104, 787, 279, 402, 360,
    299, 909, 757, 37, 647, 851, 974, 780, 518, 243, 112, 722, 672, 439, 868, 339, 85, 187

  @Benchmark
  def findInGraph2(): Unit = {
    for (nid <- randomNodes) {
      graph.edgesForVertexId(nid)
    }
  }
```

Find Edges of Random Nodes for Adjacency List Containers.



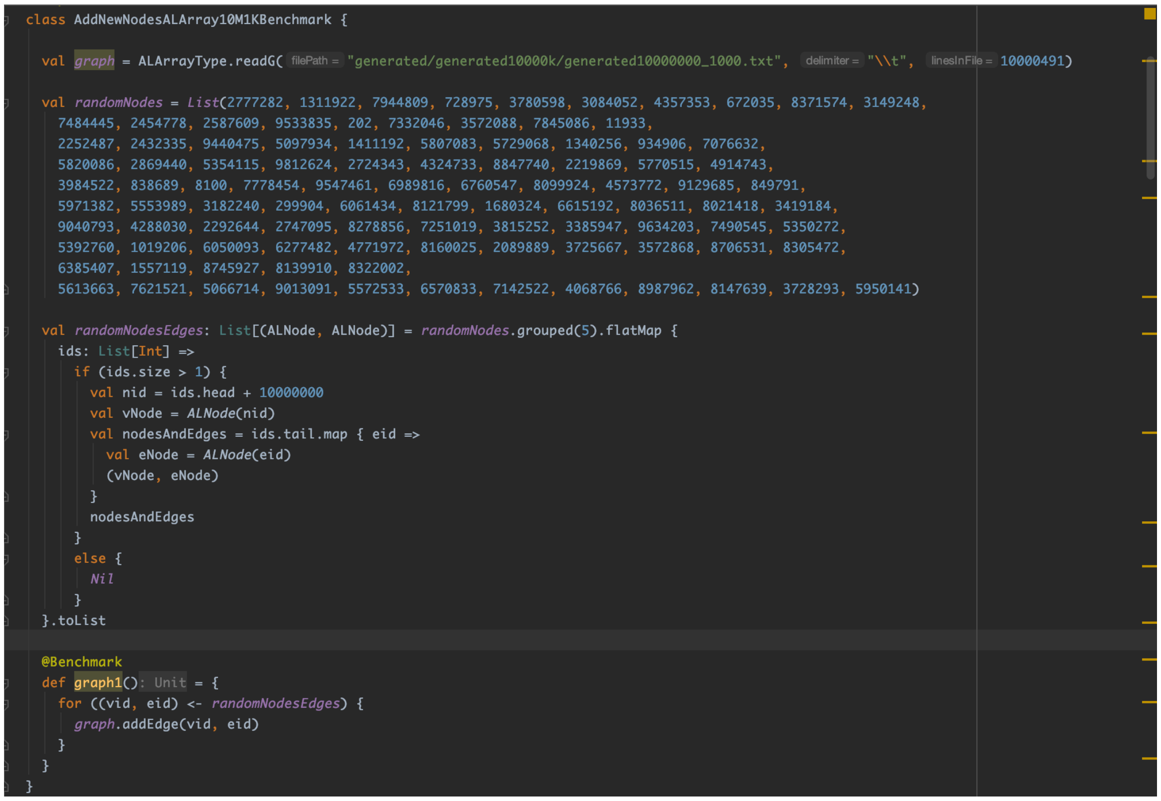
```
@State(Scope.Thread)
@BenchmarkMode(Array(Mode.AverageTime))
@OutputTimeUnit(TimeUnit.MILLISECONDS)
class RangeCSRMap1K10MBenchmark {
  val graph = CSRMapType.readG( filePath = "generated/generated1k/generated1000_10000000.txt",  delimiter = "\\t",  linesInFile

  @Benchmark
  def findInGraph5(): List[Int] = {
    graph.rangeEdges( vid1 = 8,  vid2 = 974)
  }
}

@State(Scope.Thread)
@BenchmarkMode(Array(Mode.AverageTime))
@OutputTimeUnit(TimeUnit.MILLISECONDS)
class RangeCSRTree1K10MBenchmark {
  val graph = CSRTreeType.readG( filePath = "generated/generated1k/generated1000_10000000.txt",  delimiter = "\\t",  linesInFi

  @Benchmark
  def findInGraph6(): List[Int] = {
    graph.rangeEdges( vid1 = 8,  vid2 = 974)
  }
}

}
```

Find Edges of a Range of Nodes from CSR Containers.



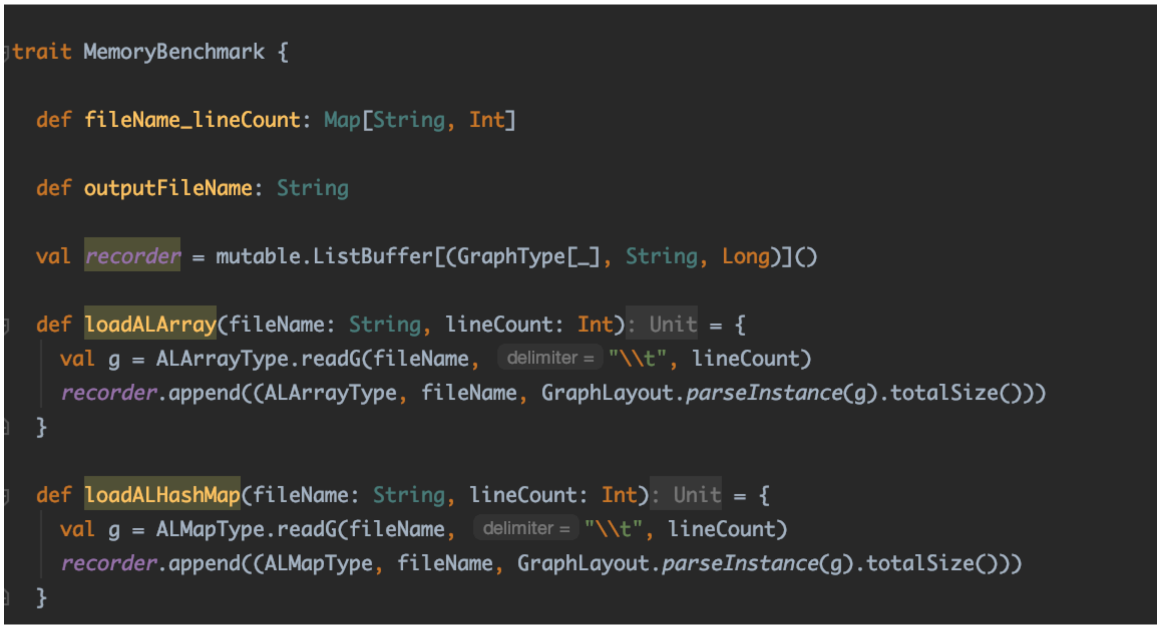
```
class UpdateEdgesALArray10M10MBenchmark {

  val graph = ALArrayType.readG( filePath = "generated/generated10000k/generated10000000_10000000.txt",  delimiter = "\\t",  linesInFile = 14999265)

  val randomNodes = List(2777282, 1311922, 7944809, 728975, 3780598, 3084052, 4357353, 672035, 8371574, 3149248,
    7484445, 2454778, 2587609, 9533835, 202, 7332046, 3572088, 7845086, 11933,
    2252487, 2432335, 9440475, 5097934, 1411192, 5807083, 5729068, 1340256, 934906, 7076632,
    5820086, 2869440, 5354115, 9812624, 2724343, 4324733, 8847740, 2219869, 5770515, 4914743,
    3984522, 838689, 8100, 7778454, 9547461, 6989816, 6760547, 8099924, 4573772, 9129685, 849791,
    5971382, 5553989, 3182240, 299904, 6061434, 8121799, 1680324, 6615192, 8036511, 8021418, 3419184,
    9040793, 4288030, 2292644, 2747095, 8278856, 7251019, 3815252, 3385947, 9634203, 7490545, 5350272,
    5392760, 1019206, 6050093, 6277482, 4771972, 8160025, 2089889, 3725667, 3572868, 8706531, 8305472,
    6385407, 1557119, 8745927, 8139910, 8322002,
    5613663, 7621521, 5066714, 9013091, 5572533, 6570833, 7142522, 4068766, 8987962, 8147639, 3728293, 5950141)

  val randomNodesEdges = randomNodes.map {
    nid =>
      val eId = nid + 10000000
      val vNode = ALNode(nid)
      val eNode = ALNode(eId)
      (vNode, eNode)
  }

  @Benchmark
  def graph1(): Unit = {
    for ((vid, eid) <- randomNodesEdges) {
      graph.addEdge(vid, eid)
    }
  }
}
```

Add Edges to existing nodes of Adjacency List Containers.



```scala
class AddNewNodesALArray10M1KBenchmark {

  val graph = ALArrayType.readG( filePath = "generated/generated10000k/generated10000000_1000.txt",  delimiter = "\\t",  linesInFile = 10000491)

  val randomNodes = List(2777282, 1311922, 7944809, 728975, 3780598, 3084052, 4357353, 672035, 8371574, 3149248,
    7484445, 2454778, 2587609, 9533835, 202, 7332046, 3572088, 7845086, 11933,
    2252487, 2432335, 9440475, 5097934, 1411192, 5807083, 5729068, 1340256, 934906, 7076632,
    5820086, 2869440, 5354115, 9812624, 2724343, 4324733, 8847740, 2219869, 5770515, 4914743,
    3984522, 838689, 8100, 7778454, 9547461, 6989816, 6760547, 8099924, 4573772, 9129685, 849791,
    5971382, 5553989, 3182240, 299904, 6061434, 8121799, 1680324, 6615192, 8036511, 8021418, 3419184,
    9040793, 4288030, 2292644, 2747095, 8278856, 7251019, 3815252, 3385947, 9634203, 7490545, 5350272,
    5392760, 1019206, 6050093, 6277482, 4771972, 8160025, 2089889, 3725667, 3572868, 8706531, 8305472,
    6385407, 1557119, 8745927, 8139910, 8322002,
    5613663, 7621521, 5066714, 9013091, 5572533, 6570833, 7142522, 4068766, 8987962, 8147639, 3728293, 5950141)

  val randomNodesEdges: List[(ALNode, ALNode)] = randomNodes.grouped(5).flatMap {
    ids: List[Int] =>
      if (ids.size > 1) {
        val nid = ids.head + 10000000
        val vNode = ALNode(nid)
        val nodesAndEdges = ids.tail.map { eid =>
          val eNode = ALNode(eid)
          (vNode, eNode)
        }
        nodesAndEdges
      }
      else {
        Nil
      }
  }.toList

  @Benchmark
  def graph1(): Unit = {
    for ((vid, eid) <- randomNodesEdges) {
      graph.addEdge(vid, eid)
    }
  }
}
```

Add Nodes and Edges to Adjacency List Array Container.



```scala
trait MemoryBenchmark {

  def fileName_lineCount: Map[String, Int]

  def outputFileName: String

  val recorder = mutable.ListBuffer[(GraphType[_], String, Long)]()

  def loadALArray(fileName: String, lineCount: Int): Unit = {
    val g = ALArrayType.readG(fileName,  delimiter = "\\t", lineCount)
    recorder.append((ALArrayType, fileName, GraphLayout.parseInstance(g).totalSize()))
  }

  def loadALHashMap(fileName: String, lineCount: Int): Unit = {
    val g = ALMapType.readG(fileName,  delimiter = "\\t", lineCount)
    recorder.append((ALMapType, fileName, GraphLayout.parseInstance(g).totalSize()))
  }
```

Memory benchmark for Adjacency List Containers.