Vrije Universiteit Amsterdam

Bachelor Thesis

Design and Experimental Exploration of Operational Techniques for Serverless Graph Processing

Author: Jelena Masic (2645593)

1st supervisor:	Prof. dr. ir. Alexandru Iosup
daily supervisor:	Ir. Sacheendra Talluri
2nd reader:	Dr. ir. Animesh Trivedi

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

July 5, 2022

Abstract

Graphs are ubiquitous in our society as we model real-world data such as human connections or maps by means of vertices and edges. To lower the entrance barrier for small and medium enterprises, the AtLarge group proposes a serverless graph processing framework called Graphless [1] that offers flexibility and ease of configuration. Thanks to the serverless nature of Graphless, companies can avoid up-front capital expenditure by leveraging public cloud infrastructure which is documented and accessible on the market.

Nevertheless, we identify further directions in which Graphless can be extended by targeting a reduction in the time needed to execute a graph algorithm from start to end. First, we notice that loading a graph, in some extreme cases, requires 75% of the overall Graphless execution time. Second, the authors of the original work identify a bottleneck in a central component, which makes Graphless several orders of magnitude slower than alternatives.

This thesis aims at exploring operational techniques to address the two major shortcomings we identify for Graphless by using an alternative implementation and by designing a new architecture after analyzing the existing data model. On top of this, we extend a testing tool [2] to reliably support repeatable evaluations of our changes.

Our results exhibit a reduction of up to 50% in the overall execution time, which we mainly attribute to different operational techniques applied to the loading phase. On the other hand, the new Graphless architecture does not help alleviate the central performance bottleneck, for which we instead achieve comparable performance. Nonetheless, the proposed architecture indirectly enables new non-functional requirements to be satisfied, which ultimately results in an easier adoption process for the Graphless framework as a whole.

Contents

1 Introduction

In the past few years, interest for graph processing methodologies has surged due to technological and societal progress [3]. This renewed interest has driven many initiatives and influential stakeholders are actively investing in improving their ETL (*Extract, Transform, Load*) and graph processing pipelines to analyze data stored in complex structures.

Simultaneously, we are witnessing growing popularity for cloud computing, both regarding public and private installations, and sometimes even a hybrid mix of the two [4]. This increase in popularity is caused by recent advancements that allow these technologies to efficiently address fundamental economical problems such as large infrastructural and operational investments, and seasonal infrastructural scalability of resources. To push this trend even further, large cloud providers such as Amazon, Microsoft, and Google developed each their own version of *FaaS* (Function-as-a-Service) [5], a paradigm that enables operation-less deployments of software without requiring engineers to manage the virtual machines on which their applications run [6, 7].

Pursuing these trends, academic research in the AtLarge group has developed architectures and models of computation that allow graph-processing in a serverless environment over the past few years, e.g., Graphless [1]. Such developments provide a generic and cloud-ready framework to deploy graph algorithms on the public cloud by leveraging the serverless platform Amazon Web Services Lambda. Nevertheless, these prototypes have exhibited performance degradation when compared to mature and optimized graph-processing solutions such as Apache Giraph [8] or GraphMat [9], that have proved to scale vertically more efficiently [2].

In this thesis, we intend to investigate whether we can overcome some of the performance limitations that Graphless has exhibited.

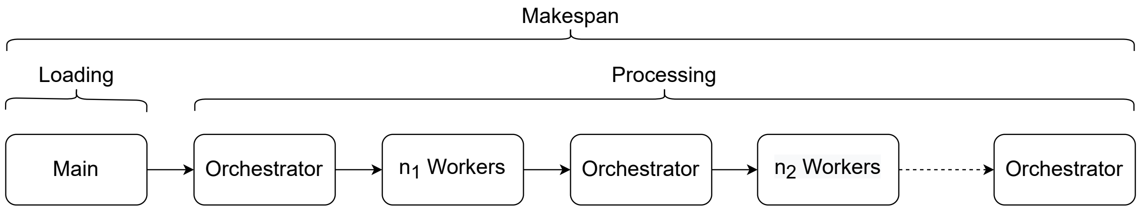

Figure 1.1: High-level Graphless execution flow.

1.1 Problem Statement

The adoption of serverless graph processing frameworks has not become a common reality nowadays. Dedicated technologies that can be run on bare-metal servers such as GraphMat or Neo4j are currently the de facto standards of practice, offering competitive performance at the expense of requiring expertise to size, install, operate, and maintain the setup of these complex systems.

To democratize access to cost- and time-effective graph processing, the AtLarge research group developed Graphless, a cloud-native solution that can fully run on Amazon AWS, whilst requiring minimum human intervention, i.e., uploading the graph data to a remote storage, provisioning all the components via an automated set of commands, and writing the algorithm operating on the graph itself. This completely automated approach is oriented towards allowing developers to focus only on writing the algorithm correctly and not spending significant amounts of time and resources on operating the infrastructure. To the best of our knowledge, Graphless currently positions itself as state-of-the-art technology for serverless graph processing, hence in the remainder of this thesis work we use it as a reference architecture and implementation.

Each Graphless execution proceeds as in Figure 1.1, where, at a high-level of detail, the **Main** function loads the graph data and then a sequence of mutual invocations between **Orchestrator** and **Worker** functions applies a given algorithm in a series of iterations, with each iteration possibly having a variable number of **Worker**s. Although the graph data is present at the end of an execution, each new Graphless run always requires to perform the loading phase again, because the previous execution typically modifies the initial state of the graph, therefore requiring a "fresh" instance of the graph to be re-loaded before running the processing phase for a second time.

Previous studies on Graphless present results by measuring the processing time, achieving acceptable results but only for graphs of modest size. Nevertheless, focusing on processing time is not enough to make the Graphless framework accessible to a wider audience. The "makespan", i.e., the period of time equal to the sequential composition of the loading and processing phases within a single Graphless execution, should be taken into account as well, because the time required by an end-user to run a whole computation depends on both phases.

Furthermore, we identify an issue with how the programming model matches the system architecture. The original implementation of Graphless combines a data-centric computation model, i.e., the vertex-centric programming model of Pregel, and the inherently stateless nature of serverless architectures such as AWS Lambda. Overcoming this mismatch requires an auxiliary memory component to maintain the state across multiple invocations; otherwise, the state not persisted in the auxiliary memory component gets lost as soon as the serverless function stops. Because the computation model can require an abundant number of states to be passed across invocations, especially in the case of some algorithms such as PageRank [1], the original Graphless architecture could lead to limited scalability for dense graphs.

1.2 Research Questions

Given the current state-of-the-art solutions for serverless graph processing, the following points of reflection derive for these technologies:

RQ1. How to implement more efficient graph loading techniques?

In the original Graphless paper, the makespan computed during empirical assessments of the system was omitted and only the processing time was taken into consideration when comparing the solution against alternative software. This choice did not explore a drawback in using Graphless, as its architecture requires the graph to be loaded into memory at every run and this overhead can easily become non-negligible for even moderately sized datasets, as in some cases loading the complete graph requires up to three times as much as the processing time, e.g., in the case of the Breadth-First Search algorithm as we found in the empirical results from the original Graphless work. In this experimentation work, we aim to implement concurrency models such as barriers and thread pools, and alternative data models to answer this question.

RQ2. Would different graph processing architectures based on serverless computing affect the processing time?
In the context of graph processing, Graphless was able to position itself as economically competitive against mature solutions such as GraphMat and Giraph but, at the same time, only achieved significantly lower performance. As identified by the Graphless authors, the main performance bottleneck in Graphless consists in the overhead caused by sending a large amount of messages during a computation step to transfer state across function invocations. Furthermore, at the end of the original Graphless work, the authors pointed out that alternative architectures could help alleviate this issue; hence, we aim to explore a new design-oriented towards reducing the overhead needed for sending messages between vertices.

RQ3. How to conduct quantitative and repeatable experiments?
Comparing two different versions of the same system might not be trivial as different parameters could be taken into account, even though evaluation is one of the cornerstones of scientific analysis based on experimentation. This problem is exacerbated with graph processing, where we might risk considering insufficiently different graphs, different algorithms, and different metrics. In particular, when considering Graphless, (1) no explicit schema definition was provided for the input, thus hindering convenient evaluation, and (2) the source code for *evaluating* the framework was not provided. Hence, for this thesis work, we aim to provide a convenient, largely automated methodology that facilitates reproducible and reliable comparisons between different graph-processing systems.

1.3 Methodology and Contribution

After enunciating the main research questions, we present here how we plan to address them.

For reducing the makespan, as highlighted in **RQ1**, we notice that the loading time may last significantly longer than the processing time. This is why, by exploring several implementation techniques, we aim to optimize this phase in the execution of Graphless. Specifically, we aim to minimize the loading time and maximize the resource usage of the loading component.

To answer **RQ2**, we explore an alternative architecture for Graphless, where graph data and messages reside in two different data stores; this is in contrast to the combined data and messages data-store in Graphless. We create and then evaluate the design, along with implementation challenges encountered during its prototype implementation.

Finally, for evaluating the outcome with mature automation, we identify two actions (**RQ3**). First, as support to rapidly produce input graphs for Graphless, we plan to develop a small script that is capable of transforming graphs from the LDBC dataset into the input format of Graphless. Then, to evaluate the changes introduced to address **RQ2**, we configure and use LDBC Graphalytics [2], which is a mature and extensible benchmark that supports multiple graphs and algorithms.

Through this work, we make the following contributions:

1. (*Technical*, **RQ1**) Implementing modifications to the loading component of Graphless.
2. (*Conceptual*, *Technical*, **RQ2**) Designing an alternative architecture for Graphless.
3. (*Experimental*, **RQ3**) Evaluating experimentally the results from the first two research questions with a mature benchmark.

1.4 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment. For the Graphless platform on which this work builds, there exist multiple sources that are easy to consult, including a peer-reviewed publication [1] and open-access source code on Github [10].

1.5 Thesis Structure

After this introductory chapter, we present useful background notions in Chapter 2 that aids the comprehension of the rest of the document. Then, we describe the solution from both a design and implementation perspective in Chapter 3. The empirical evaluation of our solution can be found in Chapter 4, which includes how we assess the resulting performance and show the outcome of our evaluation. After this chapter, we find lessons we learned along the way in Chapter 5. Lastly, the thesis concludes by coming back at the research questions and proposing future work in Chapter 6.

2

Background

This chapter describes the fundamental concepts that we deem necessary to understand the current thesis, by first introducing the notion of a graph and a computation model for distributed graph processing called Pregel in Section 2.1. In Section 2.2, we describe the databases we use for graph processing, i.e., Redis and Neo4j. Then, Section 2.3 introduces the components that serve as infrastructure during our experiments on Amazon Web Services (AWS). In Section 2.4, we delve into the details of the original Graphless framework. Finally, we survey related work in Section 2.5.

2.1 Pregel

A *graph* is composed of two sets: a set of points V (*vertices*) which represent entities of a given system, and a set of connections E (*edges*) between pairs of vertices. In turn, vertices connected by an edge are called *adjacent* vertices or *neighbors*. When the relationship between two vertices is mutual, edges are *undirected*, whereas in the case of asymmetric relationships between vertices, e.g., "is-mother-of", edges are *directed*. Additionally, edges might be associated with a numeric value called weight that often represents a cost or a capacity, e.g., weighted edges can be used when computing the shortest path between a source and a destination, with the weight being the cost or distance required to traverse a connection between two vertices [11].

Therefore, graphs are frequently used for modeling relationships that naturally exist within data and subsequently inferring information. In literature, many algorithms have been developed for solving recurrent problems and platforms, tools, and frameworks have been built to facilitate the application of such algorithms [12]. Nevertheless, the canonical versions of these algorithms often adopt a computation model that scales vertically on a single machine, i.e., they become feasible or faster by adding more computational resources such as a higher number of CPU cores or memory. Even though advancements occur frequently for computer hardware, vertical scalability is not a sustainable scaling model for large systems such as Facebook, that apply algorithms such as Page Rank on trillion-edge graphs [13].

To address this limitation, Google historically proposed a computation model called Pregel [14], whose vertex-centric approach allows horizontal scalability, therefore enabling parallel processing on multiple computation nodes which can potentially hold much larger graphs than a single machine with state-of-the-art hardware installed on it. Pregel has been created for achieving efficient, scalable, and fault-tolerant executions, reason for which it is reminiscent of MapReduce [15] as it processes each item in isolation and, only at a later stage, the system combines the individual results.

When applying Pregel as a computation model for graph processing, the program execution is divided into a sequence of iterations called *supersteps*. In this computation model, edges are directed[1] and have a value; each vertex also has a value and can be either active or halted. At each superstep, active vertices perform one step of computation, that may affect only themselves and their neighbors. During the first superstep all vertices are active, whereas at each subsequent superstep a non-strict subset of the vertex set are active.

During each superstep, each active vertex can: receive messages from the previous superstep; send messages to adjacent vertices; modify its own state; and/or modify the state of its outgoing edges. Then, before proceeding, there is a global synchronization point that allows all messages to be delivered so that the input for the following superstep is complete. After all the individual computations have successfully completed within a superstep, the next set of active vertices is equal to the union of currently active vertices and halted vertices which have received a message from at least another vertex in the superstep that just finished. A Pregel computation terminates when all vertices are halted at the beginning of a superstep or, in the case of algorithms such as Page Rank that have an unlimited number of iterations, when a given limit of iterations or time has been exceeded.

An example of how the Breadth-First Search (BFS) algorithm can be executed with Pregel is depicted in Figure 2.1. Given a source vertex, the BFS algorithm traverses all the reachable vertices and, in this variant, computes the minimum distance from the source. With Pregel, the algorithm works as follows. During the first superstep, all vertices are active but only the source sends a message with “0” inside to all its neighbors and sets 0 as the distance from itself. Then, at each following superstep, active vertices send a message

[1]this is not a limitation, as graph theory allows to transform any graph from directed to undirected

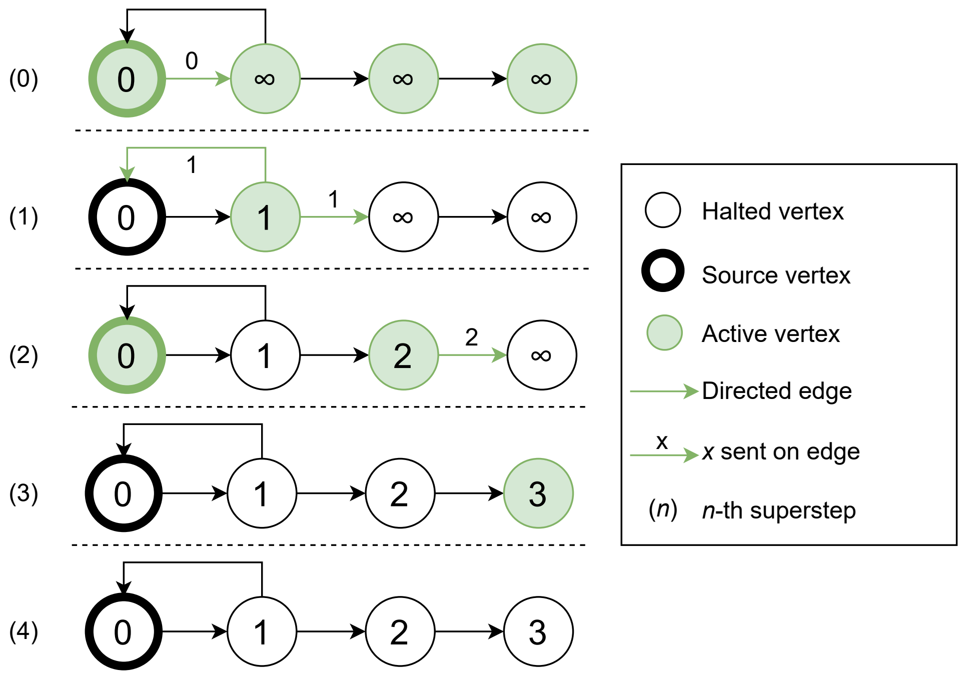

Figure 2.1: Breadth-First Search (BFS) with Pregel.

with their own distance from the source, computing this value as the content of the message received incremented by one, unless the value has already been computed, and set their value to such distance. The algorithm ends when no vertex is active at the beginning of a superstep, i.e., when all the distances of the vertices reachable from the source have been computed.

2.2 Graph Databases

Despite the current landscape of database technologies is rich and full of options [16, 17], we selectively present two suitable approaches that we apply in the current thesis work to store graph data at runtime.

2.2.1 Neo4j

Neo4j is an open-source JVM-based graph-native database as its underlying storage layer is implemented as a connected graph. Additionally, Neo4j supports property types, e.g. numbers and strings, composite types, e.g., lists and key-value structures, but also and more importantly structural types, i.e., nodes, relationships, and paths. These structural types allow Neo4j to literally represent graph data in terms of nodes and relationships, and query it by using paths, i.e., a composition of nodes and relationships. When a graph is held in memory, nodes and relationships are implemented with direct references to records, therefore queries that traverse entities represented with structural types enable to perform joins in depth faster than other types of data stores, e.g., relational databases such as MySQL [18]. On the other hand, joining data in databases that are not graph-native is usually aided by introducing indexes [19], i.e., separate data structures that allow asymptotically faster look-ups, but this approach slows down exponentially with the data size and the joining depth of the queries.

Neo4j was originally designed to scale vertically as the aforementioned data structures allow efficient storage and querying even though, as of version 4.x, operators can also deploy Neo4j in a cluster mode for better availability. Despite a full in-memory version of Neo4j exists for testing purposes, this configuration is not reliable and Neo4j as a database reliably satisfies the ACID properties [20], i.e., atomicity, consistency, isolation, and durability, by persisting transactions to disk. *Cypher* is the query language used in Neo4j, whereas *Bolt* is the application protocol used to communicate over TCP with Neo4j instances. The Neo4j driver supports many languages such as Java, Scala, JavaScript, Python, and Go. Thanks to these features and a thriving community, Neo4j is often regarded as one of the most popular and used graph databases [18].

2.2.2 Redis (Remote Dictionary Server)

Redis[1] is a key-value in-memory modular open-source NoSQL database written in C, often used as a cache, database, message broker, and streaming engine. In Redis, simple types, e.g., numbers and strings, and composite types, e.g., lists and sets, can be stored for a given key. Therefore, to be adapted for acting as a graph database, vertices can be stored as keys and their adjacency lists can be stored as list values.

Being written in C, Redis offers low runtime overhead and therefore constitutes a natural candidate for being used as a fully in-memory caching layer, e.g., to improve performance by avoiding access to slower data stores. Nevertheless, Redis also supports persistency as it allows the usage of either snapshots or AOF (Append Only File), which is comparable to the approach taken by Neo4j by logging every write operation to disk. Since its first days, Redis was designed to scale both vertically and horizontally. For the latter option, there are two main alternatives which can be combined, i.e, sharding and replication. With sharding, data is partitioned among Redis instances based on some hashing function

[1]`https://redis.io/`

applied to the key, thus addressing the need to distribute reads and writes across several machines. Conversely, replication produces a copy of the same data on multiple instances, therefore addressing availability.

2.3 Amazon AWS

Small and medium companies do not always have enough capital initially available to invest in infrastructure, e.g., machines and middleware on which applications can be set up, but also experienced human workforce to maintain complex installations. To fulfill this demand, companies such as Amazon offer an array of services that are often identified with the broad name *cloud computing*, thanks to which such companies can avoid significant initial investments by provisioning only their initial demand and then flexibly scaling up these resources as necessary.

In the case of Amazon Web Services (AWS) [21], products can be divided by means of a commonly accepted taxonomy: Infrastructure-as-a-Service (IaaS), which consists in provisioning and maintaining computational resources such as machines or storage that exist in Amazon's datacenters; Platform-as-a-Service (PaaS) allows users to abstract away some of the complexity in managing an underlying infrastructure on which they can develop software; and Software-as-a-Service (SaaS), i.e, a set of API or high-level tooling available to the end users. In addition to these categories, a fourth one named Function-as-a-Service (FaaS) was later introduced to denote "serverless" services in which users write code that executes a short-living action (*function*) when triggered, without the need of specifying the machine on which the function runs.

In the following paragraphs, we provide an overview of the main AWS services that we use for our experimental activities.

Amazon Elastic Compute Cloud (EC2). AWS allows to create IaaS computational nodes, called Amazon EC2 [21], to provide users with a service comparable to a virtual machine. When setting up an EC2 instance, users can specify operating system, vCPU, memory, networking, storage, security, and more advanced configuration parameters depending on the workload. By default, each EC2 instance is accessible via Secure Shell Protocol (SSH) by using a private key, even though endpoints can be exposed both within a private network existing in the AWS infrastructure or by opening network ports and allowing incoming traffic from IP addresses that can be configured by the end user.

Amazon Lambda. AWS offers a FaaS solution called Amazon Lambda [21], thanks to which user-defined triggers can be linked to start the execution of a program. This service has serverless traits since it only requires users to specify the language in which the logic is encoded, without being aware of the underlying infrastructure provisioned by AWS to actually run this piece of software. These functions are executed in a public network by default and users can specify memory, time constraints, and further tuning parameters such as the number of concurrent runs allowed. Each function invocation is paid proportionally to its execution time to fulfill elasticity and be aligned with the rest of the billing model offered by Amazon AWS.

Amazon S3 and Amazon EBS. One of the first services offered by AWS was Amazon Simple Storage Service (*S3*) [21], i.e., a remote object storage through which files can be managed in "buckets" as objects via HTTP REST API. This storage service is an alternative to other solutions, such as Amazon Elastic Block Storage (*EBS*) [21], i.e., a block storage device that can be mounted to IaaS or PaaS instances.

Amazon ElastiCache and Amazon MemoryDB. Amazon ElastiCache [21] is an in-memory data store provided by Amazon AWS, which is fully managed and supports multiple backends, i.e., Memcached and Redis. This data store has been designed by Amazon as a transient memory layer, e.g., for caching purposes, to achieve faster write operations. On the other hand, Amazon MemoryDB[1] provides a durable memory layer, also fully-managed and backed by Redis, at the cost of being relatively slower at executing writes when compared to Amazon ElastiCache. Both services have similar features, e.g., both are required to run on a VPC, allow horizontal and vertical scaling, and are characterized by high availability as they can be configured with up to 5 replicas.

2.4 Graphless

This thesis work builds upon a serverless graph processing framework called Graphless, which at its heart is an implementation of Pregel running on Amazon AWS infrastructure written in the Go programming language [22]. Other than Figure 1.1, which shows how a sequence of supersteps is organized in Graphless, an execution can also be represented at a lower level as in Figure 2.2, where after loading a graph there is a sequence of supersteps with orchestration and execution and, finally, results are uploaded upon completion.

[1]`https://docs.aws.amazon.com/memorydb/latest/devguide/memorydb-guide.pdf.pdf`

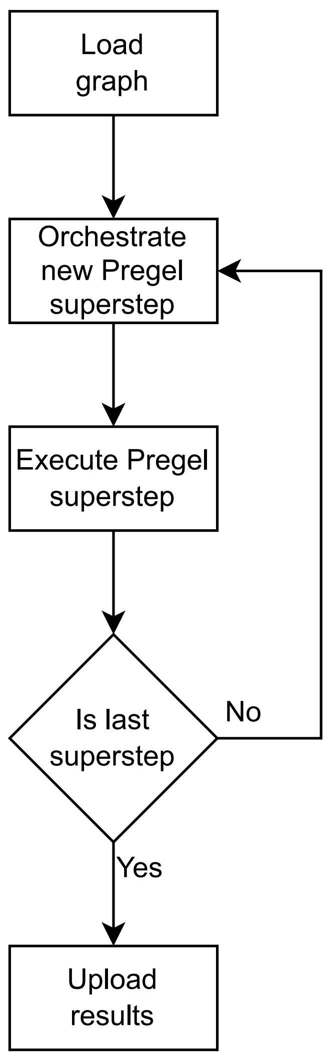

Figure 2.2: Graphless functional flow.

On the other hand, the original system architecture of Graphless can be found in Figure 2.3. The diagram includes the end user, three functional components, two data stores, and how they relate to each other.

At a very high level, a typical Graphless execution can be described as follows: the user starts the **Main** function, which pulls the graph data from a storage. First of all, the **Main** function erases the contents of a database called MaaS (*Memory-as-a-Service*), which acts as a data store, faster than the storage layer, for the system to operate on graph data. Then, **Main** loads the graph data, composed of vertices and edges, into MaaS. After loading the complete graph into MaaS, the **Main** function triggers the execution of an **Orchestrator**, which is responsible for managing the graph algorithm execution. Based on

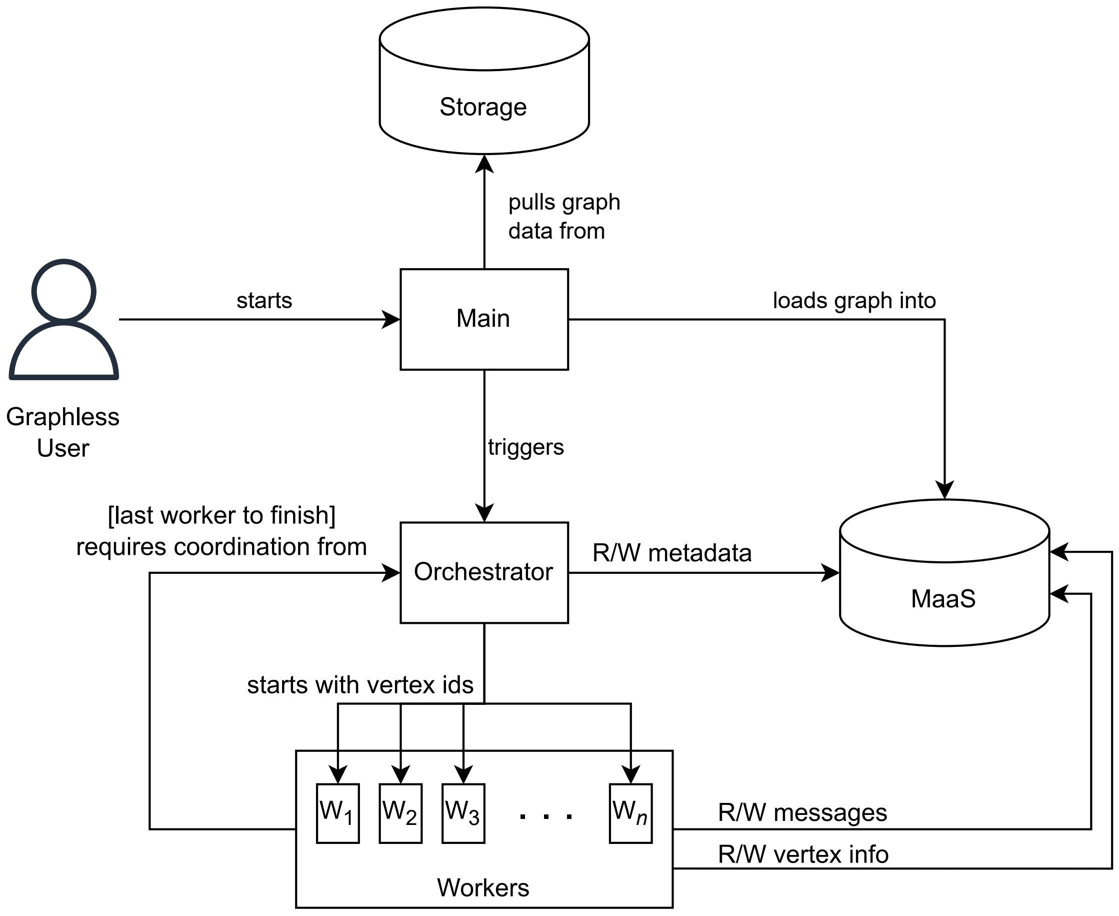

Figure 2.3: Original Graphless design.

the quantity of remaining work, the **Orchestrator** invokes one or more **Worker** functions. Each **Worker** function computes a Pregel superstep on a separated set of vertices. At the end of a superstep, the **Worker** function that finishes last invokes the **Orchestrator**, which in turn may decide to execute another superstep or terminate the execution if there is no work left to be done.

These components are realized with infrastructural technologies described in Section 2.2 and Section 2.3. The storage layer consists of an Amazon S3 bucket, the three functions **Main**, **Orchestrator**, and **Worker** run within Amazon Lambda instances, and finally, MaaS is Redis configured with sharding running on an Amazon EC2 instance.

2.5 Related Work

Other than Graphless, which we describe in detail in Section 2.4, there are no past works that offer general-purpose graph processing in a serverless environment, even though there are research efforts focused on either graph computing or evaluation of serverless environments.

Apache Giraph. As discussed in Section 2.1, the MapReduce framework developed at Google [15] has been highly influential in giving inspiration for creating new architectures and frameworks that leverage distributed computation divided into immutable units of work that are later combined for obtaining the result. One of such projects, which Meta has employed for graph processing on Facebook, is Apache Giraph [8, 13]. This framework, which also offers Pregel API to its users, is based on Hadoop's implementation of MapReduce to distribute load across distinct computation nodes. Since Apache Giraph is written in Java, its instances run on Java Virtual Machines (JVMs) on each node. Being specifically designed for graph processing, the computation model of Apache Giraph revolves around this idea with an architecture similar to the one found in Section 2.4 for Graphless. The Orchestrator, called *Master*, coordinates the execution and is responsible for the fan-in and fan-out parts of Pregel supersteps. Then, *Worker* instances are not ephemeral such as for Graphless, rather they are stateful programs that are active throughout the whole execution.

Apache Spark GraphX. Another project being maintained by the Apache software foundation that adopts this approach is Apache Spark [23]. Apache Spark presents an implementation of MapReduce in which computations are mere transformations, i.e., `map`, `filter`, and `groupBy`, applied to immutable objects called *Resilient Distributed Datasets* (RDDs). Also in this case, Apache Spark clusters run on several JVM instances as this technology is written in Scala. Since Apache Spark is designed to be a generic framework, subsequent work has been focusing on providing specialized modules for specific domains and fields, i.e., Streaming, SQL, ML, and, more relevant to our experimentation, GraphX [24]. The GraphX module for Apache Spark embodies graph concepts, e.g., vertices and edges, like RDDs, but also provides Pregel API to offer ease of portability for algorithms written with other technologies on top of the Pregel computation model.

Other graph processing systems. An alternative model to Pregel, which includes asynchronous computation, is GraphLab [25] that, even though initially designed for shared-memory execution of machine learning algorithms, was later extended for distributed processing [26]. Another system with its computation model and architecture can be found in [27], where Hong et al. propose an engine for graph processing based on "beefy" (vertically scaled) machines and low-overhead communication patterns based on data pulling. Furthermore, in 2015, researchers from Intel presented GraphMat [9], a C++ framework that does *not* embody Pregel as underlying computation model, rather it executes matrix operations by leveraging parallel processing on possibly distributed modern hardware. Taking another direction, in [28] the authors present different techniques by which it is possible to improve cache locality on single node graph processing, by reducing accesses to the main memory. Finally, in [29] the authors thoroughly explain Pregel and other graph processing computation models and present a survey of representatives for each category.

Alternative graph-native databases. In Section 2.2, we describe Neo4j, i.e., a database that incorporates concepts belonging to graph theory at its core. Nevertheless, other graph-native options are available on the market. Within the Amazon Web Services offering, we find Amazon Neptune [30], which is a fully managed distributed database that supports property graphs and standards such as Resource Description Framework (RDF). This database is integrated with the Amazon AWS ecosystem, offers ACID guarantees, and additionally provides users with a UI tool similar to Jupyter Notebooks [31] to visualize, query data, and issue commands from a console. Other than Amazon's commercial offering, feature-oriented comparisons among graph database technologies have been in the works by Angles [16] and Fernandes et al. [17], which explain the capabilities of AllegroGraph, ArangoDB, InfiniteGraph, and other alternatives in detail. Lastly, Redis Labs developed RedisGraph [32], a module to extend Redis capabilities to natively support graph entities, e.g., vertices and edges, and offer competitive graph processing by implementing the Cypher query language with sparse matrix operations.

Serverless. During the past few years, the serverless aspect of cloud computing has been gaining attention from both academia and industry. In [33], the authors provide a formal foundation for serverless paradigms of computation, defining core properties to aid the categorization of such technologies. At the same time, other works focused on proposing the design and implementation of general-purpose serverless platforms [34, 35] to foster development and interest in this research area. In the context of graph processing, the ERMer platform has been proposed in recent times [36], even though its scope (biotechnology) is restricted when compared to Graphless, as ERMer combines Amazon Lambda and Amazon Neptune to visualize data about the *escherichia coli* bacteria. Then, both studies from Kim et al. [37] and Singhvi et al. [38] analyze and discuss how applications developed by serverless users, i.e., software engineers, are affected by challenges such as cold starts and network variability, which we also encounter during our experimental work. Finally, to evaluate serverless platforms and workloads operating on them, benchmarks [39, 40, 41] have been defined to measure at different levels of granularity metrics impacting the overall performance in public cloud environments. Nevertheless, we decide to extend Graphalytics [2] for our assessment as none of these alternatives for serverless benchmarking was designed with graph processing in mind.

3

Design Space Exploration for Serverless Graph Processing

In its original work, the authors designed Graphless as a system to be maintained and adopted by small and medium companies as a cost-efficient solution compared to more expensive and mature platforms. In this chapter, after defining the requirements that drive our experimentation, we present modifications that we apply to the original Graphless project to address the two research questions that explore operational techniques.

3.1 Requirements

Before delving into how any specific research question is addressed by operational techniques, we define the system requirements based on our research questions, which we use as our guiding principles during this experimentation work.

R1. Vertically scalable processing. In elastic cloud environments such as Amazon AWS, efficient usage of resources is a focal point when deciding which technology is suitable for solving a generic problem [42]. This problem is exacerbated with graph processing, a domain in which the size of the problems often imposes tight constraints on the hardware choice on which software is run. The system should be vertically scalable, i.e., capable of reducing the processing time when adding additional resources.

R2. End-to-end efficiency. Modern graph processing technologies offer competitive execution times, both regarding loading and processing the graph, hence for the overall makespan. In particular, this holds for systems in which there is no clear distinction between the memory holding graph data and the working memory, in which case users incur the efficiency penalty for loading the graph at each run, both in terms of time and monetary costs. Thus, the system should be optimized both in terms of loading and processing operations.

R3. Ease of extension. The system, when deployed in a serverless environment, requires fast and scalable remote memory to store graph data, synchronize state, and coordinate work across functions. To achieve fine-grained scalability at this layer, infrastructural components should be easily replaceable to allow the most suitable backend to be selected for different parts of memory, each fulfilling a specific purpose. Furthermore, end users should not be asked for a significant learning investment besides understanding how the system works, hence choosing an arbitrary backend makes the system more flexible in terms of infrastructure setup and optimization. Finally, end users should be capable of deploying the system locally as a first way of learning, prototyping, and developing new graph algorithms without incurring unnecessary costs, e.g., setting up a cloud environment.

3.2 Loading a Graph

For a better understanding of the alternative techniques we later propose for loading graph data, we describe in detail how graphs are loaded in the original Graphless work and weigh the advantages and disadvantages of such an approach. Then, we present three different approaches that alter the original behavior to aim for better resource usage and improved loading time.

3.2.1 Original Loading Mechanism

In the original Graphless implementation, the **Main** function starts by erasing the contents of the memory layer (MaaS), then reads a file containing the number of vertices, number of edges, and number of files (chunks) that compose the whole graph. With this information, the same function iteratively reads each graph chunk, which contains a subset of vertices and their respective adjacency lists, initializes the vertices to an arbitrary value, and loads this information into MaaS. Finally, when all chunks are loaded, the **Main** function invokes the **Orchestrator** function which, in turn, starts the graph processing.

The implementation shown in Figure 3.1 takes into account an important infrastructure factor, i.e., Amazon Lambda instances can only be run with limited resources, especially when compared to more vertically scalable solutions like Amazon EC2. Prior to December

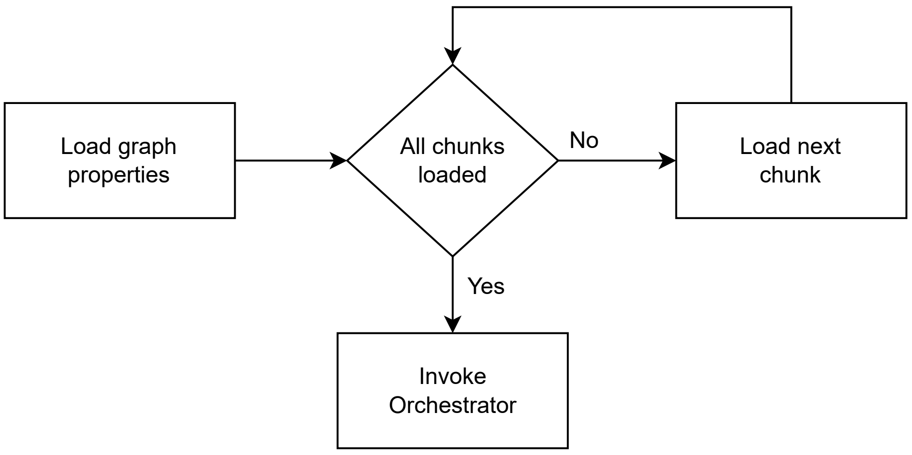

Figure 3.1: Sequential loading implementation.

2020[1], the maximum memory that could be allocated to an Amazon Lambda execution was 3,008 MB, whereas now we can start functions with at most 10,240 MB of memory. Even though this is not explicitly mentioned in the original work, graphs are split into chunks allegedly because of resource constraints, so that Graphless could incrementally load the entire graph without exceeding the memory available.

3.2.2 Loading based on Barriers

Even though the previous approach works correctly, it might not optimally use the resources provisioned for an invocation on an Amazon Lambda instance. First, some graph chunks might be significantly smaller than others, in which case a large part of the resources provisioned for an Amazon Lambda execution would remain idle for a given period of time. Second, several operations do not need to be executed sequentially, such as reading, parsing, or unmarshaling a graph chunk, as all of these operations could be potentially executed each by a distinct thread in isolation. These two drawbacks highlight a shortcoming in fully achieving satisfying requirement **R1**, as a large part of the available resources within the provisioned Amazon Lambda could end up remaining unused. Lastly, one of the requirements of MaaS is being scalable against parallel read/write requests, hence further enabling parallel faster processing of the graph chunks, thus addressing requirement **R2** thanks to a possibly reduced makespan. Therefore, we seek different approaches that are capable of leveraging the newly increased memory size and potential parallelism.

[1]`https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-supports-10gb-memory-6-vcpu-cores-lambda-functions/`

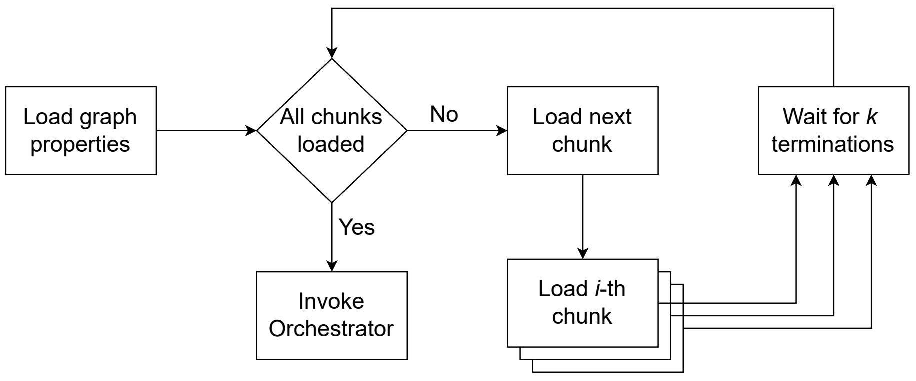

Figure 3.2: Loading implementation based on barrier.

Contrarily to other low-level programming languages as C, Go natively provides thread-safe API to handle concurrent workloads [22, 43]. Among the different utilities, a barrier implementation called `WaitGroup` is present[1]. Such API can be used to start several lightweight threads, called `goroutines`, and have the main thread waiting until all the goroutines which execute sub-tasks complete.

We use this mechanism to allow several graph chunks to be loaded in parallel, by concurrently running loading operations over several iterations in fixed-size groups to avoid (1) exceeding the memory constraints imposed by the Amazon Lambda runtime for the **Main** function and (2) overloading either the network or MaaS. Therefore, the new loading protocol works as follows: first, the graph is erased from MaaS; then, a first group of k goroutines loads the first k chunks; when all of the goroutines in the first group finish their execution, a second group of k goroutines starts loading second k chunks and so on, until all chunks are loaded; finally, the **Orchestrator** function is invoked. Because of this, $\lceil \frac{n}{k} \rceil$ iterations are needed to load n chunks, as visible in Figure 3.2.

3.2.3 Loading based on Work Queue

Despite organizing the work by means of goroutines and barriers allows to parallelize work that is performed sequentially in the original implementation, there is still some room for improvement for one specific reason, that is, there might be non-negligible variance in the time spent by each goroutine to load a graph chunk. First, as pointed out in Section 3.2.2, there is no guarantee that graph chunks have equal size, especially if non-optimized logic generates them, since each chunk contains adjacency lists and their length distribution

[1]`https://pkg.go.dev/sync#WaitGroup`

could be non-uniform. Second, great variability often characterizes the resources available for the execution of Amazon Lambda functions, especially when considering network bandwidth [38]; as most of the logic in the **Main** function involves communication over the network in a cloud environment, i.e., both for reading the chunks and loading them into MaaS, we can not assume that each goroutine is exempt from such bursts.

To address these concerns, we devise a wait-free implementation in which all units of work are loaded into a queue and several workers consume from the queue until completion. Luckily, the Go programming language natively includes another concurrency API to coordinate work between goroutines, i.e., buffered `channels` [22, 43]. Buffered channels implement FIFO queues which provide non-blocking semantics if the underlying buffer is not full at the moment of insertion.

With this approach, displayed in Figure 3.3, the **Main** function creates two channels, one to distribute work and one to collect completion signals. Then, the main thread starts k goroutines, each holding references to both channels. After starting all goroutines, the main thread populates the buffer first with all units of work and then with k "poison pills", i.e., sentinel values that allow a goroutine to detect the absence of further work items. Each of the k goroutines concurrently consumes one work item at a time from the work queue and loads the corresponding chunk. When reading a poison pill, a goroutine signals termination with the secondary channel. Finally, the **Main** function, upon receiving k completion signals from the goroutines, can proceed invoking the **Orchestrator** function as the whole graph has been loaded into MaaS.

3.2.4 Loading based on Value Reset

Lastly, there is an approach that attempts to obtain the same end results as the other loading mechanisms, but by tackling the problem from a different perspective.

After a successful Graphless execution, all vertices in MaaS would contain the vertex identifier, the weighted array of edges, and the final vertex value, which may or may not be equal to an initial sentinel value. When reporting the results, Graphless adopts the same kind of approach as during graph processing, i.e., vertex-centric, by listing all vertex identifiers and the respective value at the end of the whole graph algorithm.

On the other hand, when starting a new Graphless run in the original implementation, the graph is completely erased from MaaS since identifier, edges, and value were stored inside an opaque byte array containing a JSON string with this information in order to optimize reads and writes when using Redis as memory layer. Then, each vertex would be

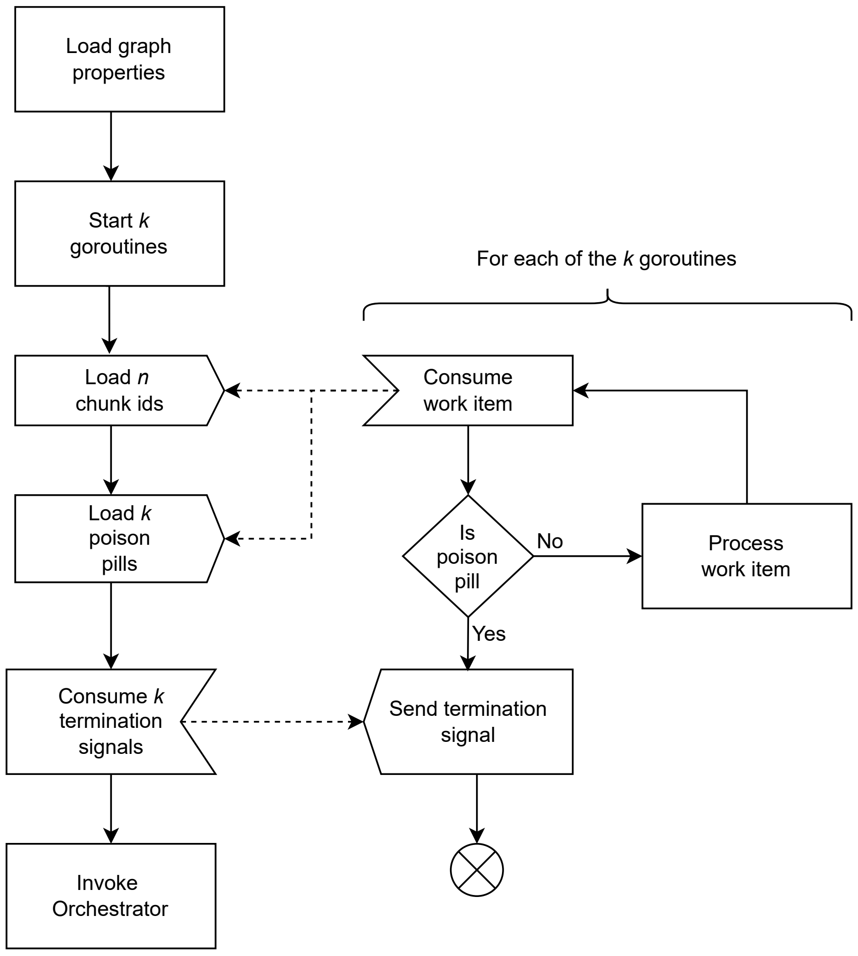

Figure 3.3: Loading implementation based on work queue.

recreated with its value set to a specific sentinel, i.e., the maximum integer value which can be stored with 32 bits.

When switching to Neo4j, the data format which is described more in detail in Section 3.3.3.1 allows us to modify the vertex value as a property without modifying the binary encoding of the vertex topological features, such as identifier and weighted edges.

Under the assumption that the same graph is reused for several algorithms or computations, with this approach we can reuse the topological features of a vertex that in Graphless are immutable after their creation, i.e., its identifier and edges. Therefore, we implement a new loading mechanism by erasing all data from MaaS besides vertices, which are retained in the memory component. Then, we (re)set the value of each vertex to the sentinel used as initial value. Finally, the execution proceeds as before by invoking the **Orchestrator** function.

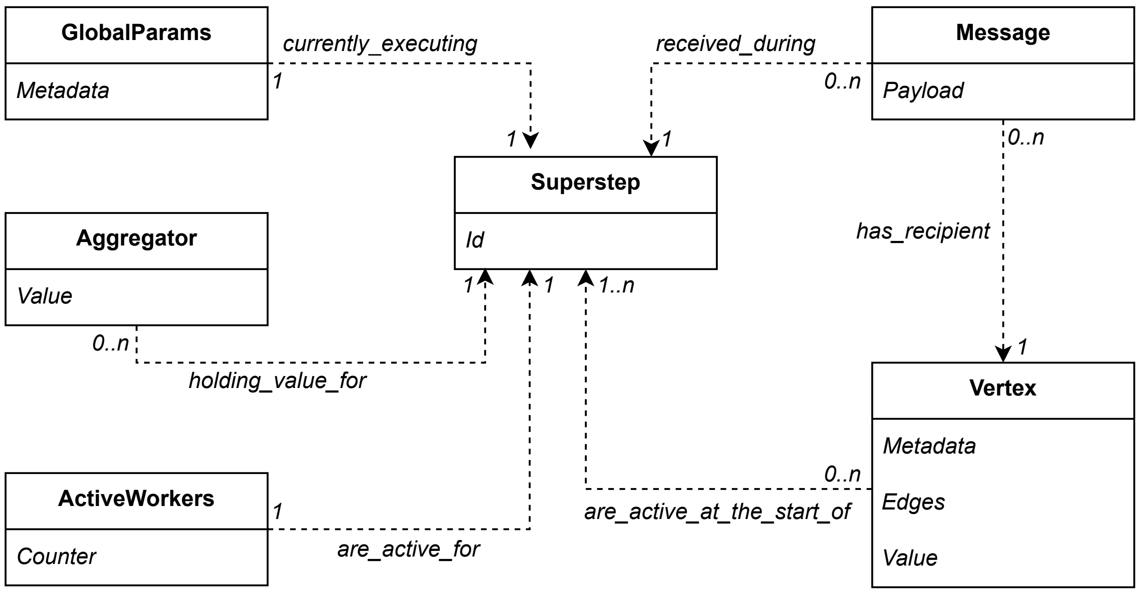

Figure 3.4: Graphless data model.

3.3 Processing a Graph

In the original Graphless work, the authors provide empirical evidence that, when using Pregel APIs as a computation model combined with serverless architectures, significant communication overhead is imposed on the processing phase when sending a large quantity of messages, e.g., with dense graphs. In this section, we first analyze the current data layout and then illustrate the design and implementation of an alternative architecture that we propose to mitigate this problem.

3.3.1 Data Model

Before introducing our new candidate Graphless architecture, we describe the data model of Graphless to provide our rationale behind the new architectural proposal. In Figure 3.4 we visualize a graphical representation of how data is organized in the MaaS layer of the original Graphless project.

In the diagram, relationships between entities help in understanding how they can be used together within the span of a single execution. For each **Vertex**, we store its identifier, a weighted adjacency list, and a value that is updated throughout the execution. At each invocation, the **Orchestrator** function retrieves the current **Superstep** from the **GlobalParams** metadata, hence determining which vertices are active at the moment and, depending on this quantity, sets the counter in **ActiveWorkers** to match the number of **Worker** functions that are being invoked. Furthermore, every time that a **Worker** sends a **Message** to a given vertex for the following superstep, this information is captured in the memory layer as well. Finally, some algorithms such as PageRank make us of an **Aggregator** to hold values accessible by any vertex; this entity helps circumvent a limitation of Pregel API, which would otherwise restrict communication to neighboring nodes only.

From the explanation above, three main distinct parts emerge for subsets of entities: (1) graph data, composed of vertices, i.e., the graph topology and results accumulated throughout the execution; (2) coordination data, used to orchestrate the execution of several components until completion; and finally, (3) message data, which stores communication between nodes as required by the adoption of Pregel API as underlying framework.

3.3.2 An Alternative Graphless Architecture

After analyzing the data model in Graphless, we notice that some of the entities can be extracted to a separate memory to better distribute the load within the MaaS layer. Therefore, we present a modified version of Graphless in Figure 3.5, where the MaaS component previously present in Figure 2.3 has been split into two sub-components, highlighted by means of a different color. The two new components are Graph MaaS, which contains the graph and coordination data, i.e., entities whose traffic patterns are less data-intensive, and Message MaaS, which contains the message data, i.e., payloads exchanged among the vertices during each Pregel computation superstep.

By proposing this new architecture, we gain two main benefits: (1) enabling dedicated optimizations and technologies for each of the two memory components, tailored to their respective purposes and query patterns; and, (2) tentatively isolating the load required from MaaS to send and receive messages. These two features address two requirements, i.e, **R2** by trying to improve the time spent running **Worker** functions at the cost of provisioning additional memory storage, and **R3** by allowing the replacement of specific components within the architecture for tailored optimizations on different technologies.

All the other components also present in Figure 2.3, i.e., **Storage**, **Main** function, **Orchestrator** function and **Worker** function remain unaltered. Because of this reason, the implementation efforts for splitting MaaS into two components were minimal, i.e., merely dividing a programming interface in two. On the other hand, integrating Neo4j in the Graphless implementation required non-negligible efforts, hence the following subsection describes the challenges and limitations we face when accommodating the new backend.

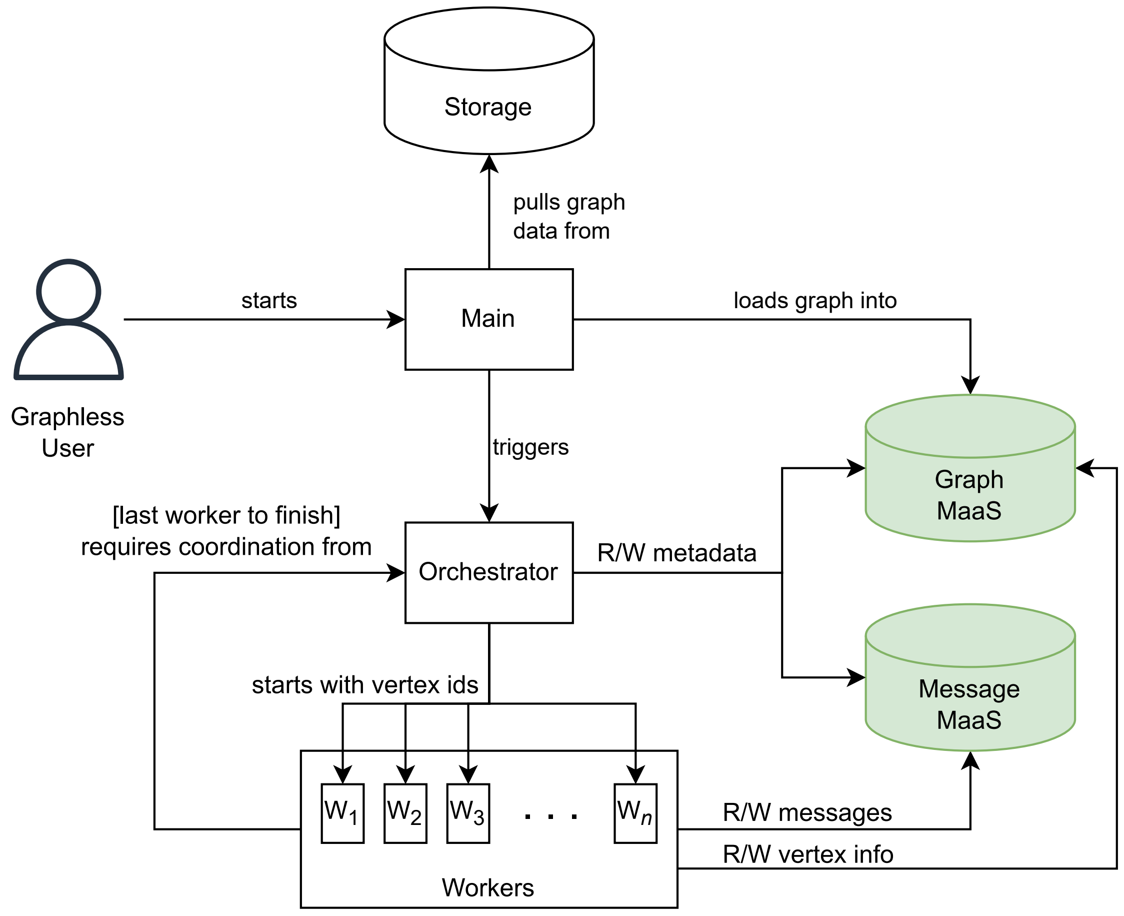

Figure 3.5: Alternative Graphless design with split MaaS layer.

3.3.3 Introducing the Neo4j Backend

Hereafter, we present how we integrate Neo4j into the MaaS layer, with a focus on graph data. Despite we provide a fully-working implementation for Message MaaS, this option is not usable in practice due to the limited scalability in terms of write operations offered by Neo4j as a technology.

3.3.3.1 Storing Graph Data

Neo4j natively supports structural types as nodes, relationships, and paths, therefore making it a natural option when choosing an underlying technology where graph data can be stored. Nevertheless, in our case, we are *not* storing vertices and edges respectively as Neo4j nodes and relationships for two main reasons that have been discovered after preliminary experimentation, one affecting loading and the other regarding the Graphless computation model.

First, the loading mechanism takes as input graph chunks from storage, with vertices and their respective adjacency lists being stored inside. If we were to decide to keep this input format, we would need to create the source node, all the destination nodes, and then the relationships between them for each graph chunk, but this option highly compromises time efficiency as it requires Neo4j to continuously check uniqueness constraints when creating the destination nodes, exhibiting poor performance, especially in the case of dense graphs. Alternatively, we could create each source node and its relationships with the destination nodes implicitly created, but this option also poorly scales since creating the destination of a Neo4j relationship by specifying a path does not strictly enforce uniqueness, hence the loading phase may finish with duplicate vertices inside the graph data. As a final consideration regarding the loading mechanism, we could change the input format to chunks of vertices and chunks of edges to first create all vertices and then create all edges, but this choice would entail modifying the loading logic and strategy for the Redis Graph MaaS backend, thus we do not apply it for ensuring a finer scope when evaluating our results.

Then, Graphless uses the Pregel API, a vertex-centric paradigm in which every node sends messages only to its neighbors. Therefore, leveraging the Neo4j capability of efficiently traversing a path of relationships would not be used at its full potential, as Graphless would always require to query nodes distant at most one relationship away.

Even with these limitations, we decide to evaluate Neo4j as a technology for Graph MaaS because of its scalability in terms of graph size, as Neo4j does not have an upper limit regarding the number of nodes, and its read efficiency, which we obtain by setting up optimized queries. Hence, to provide a scalable implementation we implement vertices in Neo4j merely as nodes without relationships, similarly to the Redis Graph MaaS backend, with the difference of storing values separately as a property to (1) reduce the overhead incurred at each superstep when updating the value of a vertex, as in Redis we would need to send the whole vertex including its adjacency list marshaled as JSON, and (2) to enable further optimizations at load time as described in 3.2.4. As a final remark, we decide to omit to present results in Chapter 4 for graph data being stored with Neo4j nodes and relationships, as all the test executions resulted in failures due to execution times exceeding the thresholds set for the Amazon Lambda invocations.

3.3.3.2 Detecting Superstep Termination

Each Pregel computation superstep involves a number of workers being executed and, at the end of a superstep, coordination from the **Orchestrator** function is again required to decide how to distribute the load for the following superstep. Because of this, Graphless embodies a mechanism to detect the termination of a superstep, i.e., the last **Worker** function to complete would become aware that it needs to invoke orchestration as shown in Figure 3.5.

In the original Redis MaaS backend, this is implemented by setting a key to the number of active workers at the beginning of each superstep and then each worker, upon completion, decrements and gets the value for the same key by using the Redis `DECRBY` command. As each operation within the same Redis instance is executed in an atomic fashion by the same thread, retrieving the value of the workers that still need to complete their execution is reliable in terms of concurrency and, when a **Worker** function reads 0 after completing, then it can safely invoke orchestration.

When detecting superstep termination in Neo4j, the same mechanism is not safely portable as Cypher, i.e., the Neo4j query language, doesn't support atomic decrement-and-get operations. To address this limitation, we identify three alternative options.

First, we could use the APOC API[1], which are community extensions for Neo4j that can be added as a Java archive when starting the database, thus enabling API with a richer semantics than the base Cypher syntax. Among these API, we have thread-safe operations, e.g., decrement-and-get, at the expense of further customizing the Neo4j installation and potentially degrading query performance as these operations require holding a lock on the Neo4j node being modified.

Second, we might invoke the **Orchestrator** function by designing an alternative Graphless architecture in which the **Orchestrator** function is automatically triggered within the Amazon AWS ecosystem in response to an event, e.g., a message queue reads a given number of notifications or after a database update, but this solution exhibits strong portability limitations by introducing tight coupling with Amazon AWS as an underlying platform.

Third, we could adopt an approach with an immutable data model by counting the number of **ActiveWorkers** that finished a superstep execution by creating a new Neo4j **FinishedWorker** node upon worker completion, each containing a timestamp computed on the server side at the time of insertion. Then, this protocol requires each **Worker** function to get the count of both the total amount of workers in the superstep and workers

[1] `https://neo4j.com/developer/neo4j-apoc/`

that completed their execution. If these amounts do not match, then some worker has not fully processed its backlog yet. If these amounts instead do match, then this worker might be the last to complete within a given superstep, hence it queries the database to retrieve the latest timestamp for a **FinishedWorker** to determine whether its timestamp was the last added, in which case the worker invokes orchestration. Finally, at the beginning of each superstep, the **Orchestrator** does not only set the number of **ActiveWorkers** but also resets the number of **FinishedWorker** nodes by deleting them.

Given these three alternatives, we opt for the third one, i.e., establishing an immutability-based protocol to handle concurrency, as it provides better performance, non-invasive changes to the overall design and implementation, and does not introduce a hard dependency on either a community-supported API or a specific cloud provider. A diagram representing this algorithm can be found in Figure 3.6.

3.3.3.3 Throttling Worker Execution

After implementing a thread-safe data model for reliable Graphless execution in presence of a Neo4j backend, we soon start discovering inherent limitations in how Neo4j scales with the number of queries compared to Redis.

Also in this case, these intermediate results are omitted from the evaluation in Chapter 4 as each Graphless run completes with failures due to execution timeout because of Neo4j not being capable of handling an increased number of queries in a time frame of some milliseconds. In the Neo4j logs corresponding to these runs, we notice long JVM GC pauses, i.e., garbage-collection interruptions in the Java Virtual Machine, for which the database would stop processing incoming requests for hundreds of milliseconds or, in some cases, even seconds.

To overcome this performance bottleneck, we decide to change the fan-out logic in the **Orchestrator**, i.e., the logic used to start new workers. As we extend the original Graphless push-based implementation, we choose to throttle the invocation of workers, therefore starting an arbitrary fraction of them at fixed-size time intervals. This approach, despite being simple and effectively reducing the concurrent load imposed from the **Worker** function instances on Neo4j at the beginning of each superstep, has a drawback in increasing the overall makespan as the processing phase would be executed with a throttling mechanism.

3.3.3.4 Supporting Local Runs

At last, we introduce one last modification to further satisfy **R3**, as we extend the Graphless framework for enabling inexpensive development of new modules.

While experimenting with Neo4j, we soon realize that running executions against small inputs could lead to infinite loops due to concurrency challenges, such as the aforementioned ones. Given the necessity of efficiently debugging similar extensions, prototyping in a cloud environment reveals itself to be prone to several drawbacks: lack of visibility, as by design there is no control over the machine running the serverless code; after inspecting the execution logs from Amazon CloudWatch, we experience seconds of delay between the time at which an event is produced and then ingested[1]; deploying the code and configuration for a set of Amazon Lambda functions taking up to minutes; and, finally, non-negligible monetary costs in keeping active the MaaS layer. Hence, these shortcomings present in the original Graphless implementation could severely hinder the adoption from small/medium companies and academic researchers.

For these reasons, we introduce support for local executions, i.e., new backends for the **Storage** layer and invoking functions. To support local storage, we implement some S3 API, i.e., `get` and `put`, to simply read and write from the local file system. On the other hand, implementing function invocation by programmatically invoking the **Orchestrator** and **Worker** functions, proves to be more difficult as this would introduce circular dependencies between code paths, which is forbidden by the Golang compiler [44]. To resolve this challenge, we decouple the main logic of these two functions in dedicated classes and inject the MaaS and storage API via constructor parameters.

After resolving these two implementation challenges, we successfully run the experiments locally with Docker containers for Redis and Neo4j, therefore obtaining a significantly faster feedback loop.

[1]we measure an ingestion delay between 60 milliseconds and 9.4 seconds, averaging 5.4 seconds

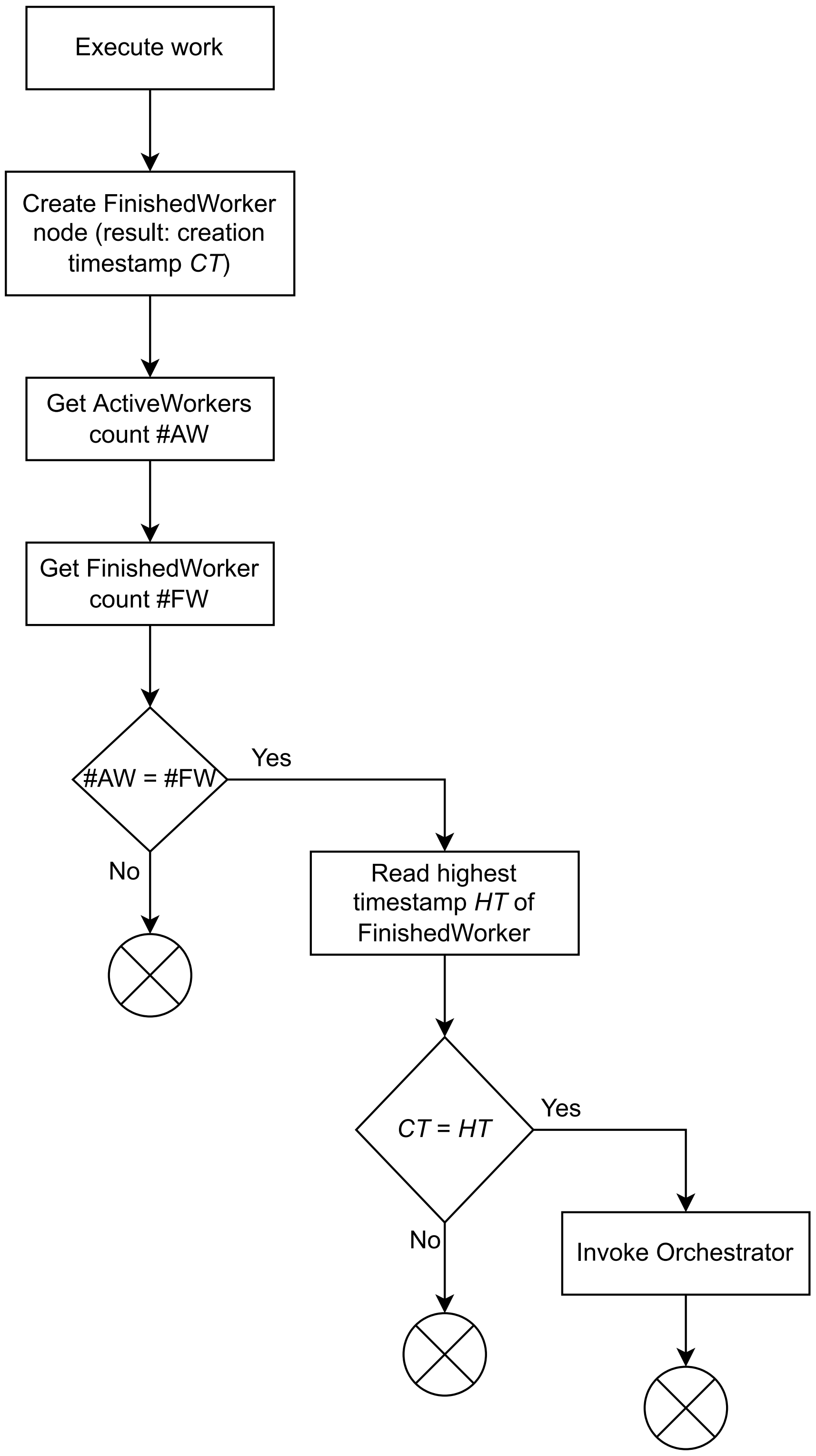
Execute work
Create FinishedWorker node (result: creation timestamp *CT*)
Get ActiveWorkers count #AW
Get FinishedWorker count #FW
#AW = #FW
Yes
No
Read highest timestamp *HT* of FinishedWorker
CT = *HT*
Yes
No
Invoke Orchestrator

Figure 3.6: Algorithm to detect superstep termination.

4

Empirical Evaluation of our Techniques

In this chapter, we highlight how our evaluation was driven to assess the requirements elicited in Section 3.1. Then, we present our experimental setup and the results we obtain for the two research questions that are empirically measurable, i.e., **RQ1** and **RQ2**. While explaining the setup, we discuss how we use Graphalytics to answer **RQ3**.

4.1 Goals

In Chapter 3, we first present requirements and then propose solutions for them. While some requirements are inherently well-suited for an objective evaluation, e.g., answering whether an approach enables vertically scalability, others might need a better definition as their nature might feature subjectivity, e.g., whether an approach is easily extensible. Therefore, we define a set of evaluation goals to give clarity on the scope of our experimental activities.

G1. Reproducing the original results. A challenging part of any research activity is reproducing previous results, as the underlying hardware on which evaluations run might change after some years, or just because the methodology applied is slightly different from the original evaluation steps. Furthermore, this goal is especially important as failing to reproduce the original results of previous experiments might compromise any further claim or observation made on top of an ill configuration. Therefore, in Section 4.4.1 and Section 4.4.2 we include the results that we empirically obtain by running the original implementation of Graphless, which serve as a baseline for evaluating the proposed solutions.

G2. Assessing feasibility with alternative technologies. Despite being vertically or horizontally scalable, a technology could still require an economically unfeasible amount of computing or monetary resources to solve a specific problem. When evaluating at scale, even inputs of modest size might compromise the successful outcome of a Graphless run, i.e., an execution that completes all Pregel supersteps and produces the correct output. Therefore, even before taking into account further metrics, we are interested in verifying that executions terminate correctly for an arbitrarily chosen technology to satisfy requirement **R3**. Our results demonstrate that using Neo4j is a feasible Graph MaaS option for supporting serverless graph processing workloads even though, in some exceptional cases such as when running the SSSP algorithm, Graphless completes its execution in twice as much time compared to the same program when using Redis as Graph MaaS.

G3. Measuring vertical scalability. During our preliminary evaluations to reproduce the results of the original Graphless work, we observe that the resources provisioned for the **Main** function are underutilized, which is a limitation in terms of the requirement **R1** formulated in Section 3.1. On one hand, over-allocating memory for an Amazon Lambda invocation could be beneficial as it yields a higher share of CPU available for computation [45]. Nevertheless, if the function does not make use of the additional space available, there might be room for optimizing the loads and obtaining lower billing costs. This is why, while measuring the loading time for the techniques presented in Section 3.2, we also profile the memory actually consumed by the **Main** function. Our empirical evaluation shows how we can make better use of resources by employing concurrency, thus gaining a 9-fold improvement in terms of memory used.

G4. Measuring makespan. Contrarily to the original Graphless work, which presents results in terms of processing time, we aim to capture the whole picture in terms of execution time by including the time elapsed for loading the graph into the MaaS memory layer. Therefore, we provide results both for loading and processing time and, on top of that, we summarize these figures into a holistic view that represents the point of view of an end user as expressed for requirement **R2** in Section 3.1. Our exploration confirms that, by adopting operational techniques targeting both loading and processing time we obtain a competitive makespan, hence halving the end-to-end execution time in the most favorable case, i.e., when running the BFS algorithm.

4.2 Environment

Even though in Section 3.3.3 we describe how Graphless can now support local runs, we decide to use a public cloud platform to run our tests to obtain repeatable results in an isolated environment, as any other researcher interested in evaluating Graphless would be able to provision the same hardware and configuration.

At the same time, running experiments in a cloud environment such as Amazon Web Services allows users to satisfy two requirements of the original Graphless work, i.e., (1) obtain fine-grained elasticity that can be tuned according to the graph size and (2) automate resource management with minimal knowledge of the system, e.g., FaaS utilities thanks to which users can start an arbitrary number of function executions without having to configure the machines on which they run.

4.2.1 AWS Configuration

We provide a high-level infrastructural representation of our experimental setup in 4.1. To run any MaaS backend, we create EC2 instances of the same kind (`r4.8xlarge`) which, similarly to the original Graphless work, have a Linux operating system, 32 vCPUs, 244 GiB of RAM, Elastic Block Storage (EBS) as underlying storage, and 10 Gbps of networking performance. The R4 EC2 family comprises a range of instances that are optimized for memory-intensive workloads, making this family suitable for our use case since we configure both Redis and Neo4j to hold as much information as possible in main memory for the sake of throughput. Lastly, part of the experiments require two MaaS backends to run simultaneously; to have 32 vCPUs for both backends and avoid requesting a vCPU limit increase from the Amazon AWS support team, we set up a second AWS account on which we run the second backend.

Amazon EBS is a family of block storage volumes that includes both SSD and HDD options that have different flavors based on the optimization required for the application running on an instance. For the EC2 instances on which we install Neo4j, we attach a `gp3` (general purpose) block storage device with 40 GB of size and 6000 IOPS, i.e., twice as much as the maximum allowed for `gp2` devices. We choose these values empirically, but this discussion is omitted from the current chapter since this choice did not significantly influence the final results of our tests as a non-optimized, general-purpose volume is enough to guarantee an adequate level of performance as disk storage for Neo4j.

Finally, we set up similar resource limits for our FaaS infrastructure (Amazon Lambda) to the original work, with the following differences: the **Main** and **Orchestrator** function

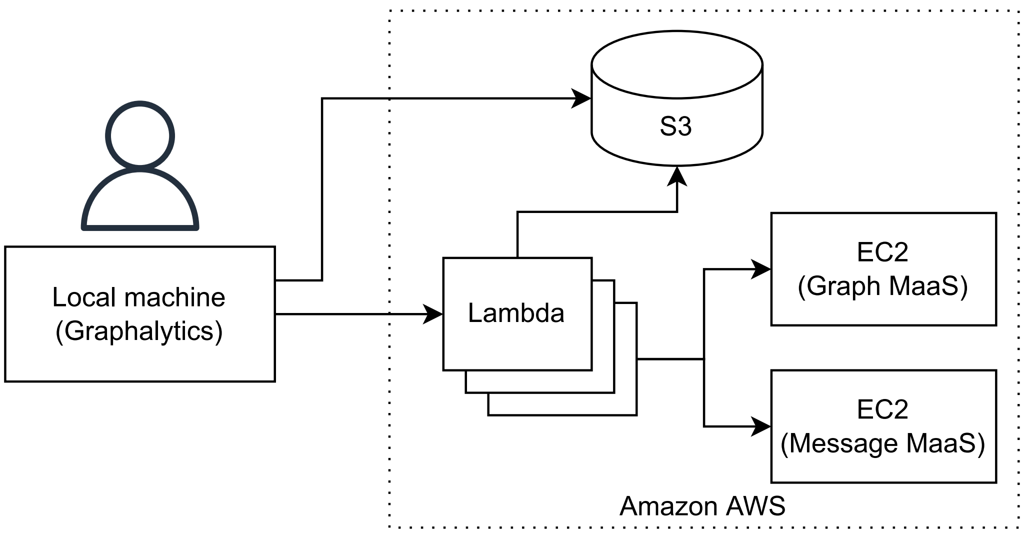

Figure 4.1: High-level representation of our experimental setup.

have a higher memory limit (6000 MB) whereas the worker function has a lower limit (1600 MB), due to the fact that the former requires to hold respectively large binary representations of several graph chunks and the results for the whole graph; all the functions now have a reduced execution duration limit of 600 seconds to prevent accidentally high expenditures due to programming errors such as infinite loops.

4.2.2 Database Configuration

Both of the database technologies we use for the memory layer (MaaS), i.e., Redis and Neo4j, are configured to achieve better efficiency under significant load. The discussion about how we conduct this optimization is omitted, as this configuration is based on best practices collected from publicly available knowledge (official documentation, conference presentations, and forums) and does not address specific research questions.

Redis. We keep the same Redis configuration as in the original Graphless work, including the database version to reproduce results as close as possible to the ones reported in the past. In the original Graphless repository available at the AtLarge GitHub group, a bash script allows to install and start an arbitrary number of Redis shards. We modify this script to pin different Redis shards to different CPUs and disable hyperthreading, as mentioned in the results chapter of the original Graphless thesis. Furthermore, we decide to set the number of shards to 16, as previous works provide empirical evidence that Graphless scales well with a higher number of Redis instances.

Neo4j. On the other hand, we start this exploration with no configuration for Neo4j, therefore we set it up from scratch as this technology is not present in the original implementation. We use Neo4j version 4.4.6 and change part of its default configuration options. As for the memory, we decide to allocate exactly 128GB as JVM heap size to avoid expensive resizing at runtime, with the page cache set to 64 GB, i.e., 50% of heap size as per recommendations from the official documentation. For the database scalability, we increase the maximum amount of Bolt connectors, i.e., query receivers, and the allowed number of file descriptors in the Linux service (systemd) respectively to 1000 and 60000. As for the disk usage, we decide to increase the frequency of checkpointing, i.e., the Neo4j process used to compact transaction logs into smaller files, to either 1 minute or 100000 transactions, to prevent the disk from rapidly exhausting the available space under burst loads. Finally, we automate the generation of database indexes and constraints set up for the reasons described in Chapter 5.

4.3 Experimental Setup

Two of the research questions, i.e., **RQ1** and **RQ2**, are strongly oriented toward analyzing changes in performance when applying some modifications to Graphless. At the same time, these questions are focused on two separate components of the framework under analysis, namely loading the graph into memory and processing the graph by applying a distributed algorithm. Therefore, we decide to employ two different approaches to precisely measure the effect of our changes, which are discussed in the remainder of this section.

4.3.1 Experimental Setup for Loading a Graph

Each Graphless run starts with loading the graph into the central memory (MaaS), an operation which takes place in the **Main** function. In the remainder of a normal Graphless execution, i.e., in the **Orchestrator** and **Worker** functions, a graph would not be loaded again under any circumstance. For this reason, when measuring the loading performance, we decide to run only the **Main** function and skip the invocation of the **Orchestrator** function to save time and financial resources. For the same reason, the results presented for loading performance do not depend on a specific graph algorithm, since its execution is entirely skipped.

To compare the difference among loading implementations, we decide to use one graph from the LDBC Graphalytics repository, i.e., *dota-league*, that is a real-world dense graph with $61,170$ vertices and $50,870,313$ edges which was also used in the original Graphless work. Also similarly to the original work, the graph data is divided into 50 chunks, each of which contains a distinct subset of vertices and their respective (weighted) adjacency lists.

We run the experiments for each of the loading techniques described in Section 3.2 and compare their results both when using Redis and Neo4j. For the first three techniques, i.e., sequential, barrier-based, and queue-based, we erase the contents of MaaS prior to the run and measure the time required to load the graph into memory; for the barrier- and queue-based implementations, we use 7 goroutines respectively as group and pool size. On the other hand, we only use Neo4j for the last technique, i.e., resetting the vertex value, where we "warm-up" MaaS by loading the whole graph once, we remove all the data but the vertices, we set an arbitrary value different from the sentinel for all vertices to avoid optimizations from Neo4j, and then run the **Main** function by measuring the loading time[1]. In this last case, we omit running the technique against Redis as there is no implementation for this MaaS variant as described in Section 3.2.4.

For each scenario, we measure both the time for loading or preparing the entire graph, which we store in an S3 bucket, and the memory consumption by Amazon Lambda, which is reported at the end of the run.

4.3.2 Experimental Setup for Processing a Graph

Conversely to Section 4.3.1, processing a graph involves all the functions except for the **Main** one. Since this phase of the experiment is independent of the previous one, we arbitrarily choose the most convenient loading technique based on the results gathered after evaluating the loading phase.

In this case, we run Graphless against the same algorithms and graph data used in the original work, i.e., Breadth-First Search (BFS), Community Detection using Label Propagation (CDLP), Page Rank (PR), Weakly Connected Components (WCC), and Single-Source Shortest Path (SSSP). We apply all these algorithms to the *dota-league* graph from the LDBC Graphalytics repository.

For quantifying the impact of using the alternative architecture proposed in Section 3.3, we first establish a baseline by generating fresh results for the original Graphless version, then we measure the processing time for the architecture with the split MaaS memory layer for both Redis and Neo4j. Specifically, we define processing time as in the previous chapters, i.e., the time elapsed between the first invocation of the **Orchestrator** function

[1]we arbitrarily used transactions in batches of 10,000 nodes to avoid overloading the Neo4j instance

and the last invocation of the same function, therefore without taking into account the time spent storing the results into an AWS S3 bucket.

Repeating such measurements can be error-prone and inaccurate if executed manually, hence we decide to extend LDBC Graphalytics with a new platform driver to address **RQ3** as, contrarily to the rest of the project implementation, the platform driver originally developed for Graphalytics is not publicly available. Also within the scope of Graphalytics, we provide a driver that works both against AWS infrastructure and on a local machine to fully support the adaptations described in Section 3.3.3.4. In both cases, our driver retrieves and uploads input graphs, clears previous results from the **Storage** component, starts Graphless with operating system commands via Java code, and polls the **Storage** component until Graphless generates new results. Furthermore, as Graphless requires a custom input format to ingest graphs, we also develop a Python script that, given the path at which standard Graphalytics graph files can be found, creates the weighted adjacency lists for each vertex, adds the backward edges for undirected graphs, and stores subsets of vertices and corresponding adjacency lists into compressed graph chunks. Lastly, both output values and metadata are fetched and analyzed, thus respectively validating the algorithm execution and extracting the processing time.

Finally, we configure Graphless to utilize at most 400 concurrent **Worker** instances per superstep, each handling at most 153 vertices at a time. In the Graphless version with the split architecture, we keep fixed the throttling parameters introduced in Section 3.3.3, with at most 16 workers invoked at a fixed rate of 400 milliseconds. These last values are empirically found suitable for the Neo4j backend to complete in time and, for a fair comparison, are also used for Redis.

4.4 Results

In this section, we present the results for **RQ1**, i.e., how the different techniques discussed in Section 3.2 affect the loading time of a graph for our system, and **RQ2**, that is how the new architecture discussed in Section 3.3 affects the performance when running different algorithms on a given graph.

4.4.1 Results for Loading a Graph

In Figure 4.2 and Figure 4.3, we respectively present the time and memory required to either load or prepare the graph for a new Graphless execution. On the x-axis, we find the different variants, i.e., MaaS backed by either Redis or Neo4j, grouped by loading

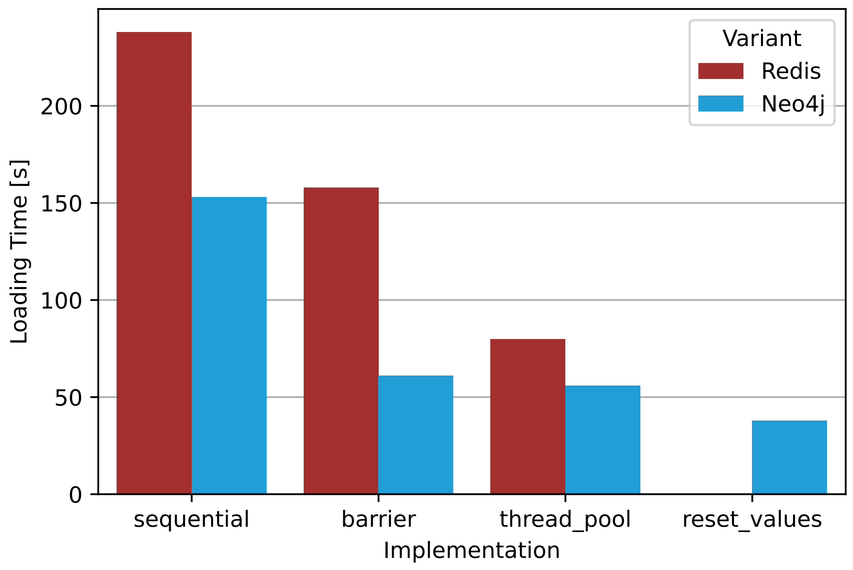

Figure 4.2: Time comparison while loading graph.

technique. On the y-axis, we instead find the loading time, in seconds, and the used memory, in Gigabytes. While a lower amount of seconds is in general preferable since the user would wait less before an operation completes, a higher memory consumption indicates a better resource usage as long as the used memory does not exceed the maximum amount of memory provisioned, which is drawn with a dotted line.

In the first group, we show the original sequential implementation which is already present in the public repository as a baseline, which achieves respectively 238 and 153 seconds of loading time when using Redis and Neo4j as MaaS. These values are in line with our expectations, as by computing the difference in the original Graphless work between the makespan and the processing time, we obtain approximately 250 seconds. As for Neo4j, we see that the loading process requires slightly more memory on the **Main** function side, but performs better than Redis, which we attribute to efficient configuration tuning and scalability with a reasonably modest workload. Nevertheless, we observe low peak memory consumption, since each graph chunk can be garbage-collected by the Go runtime after being loaded.

Then, in the second group, we find the results for the barrier-based solution, which in the case of Neo4j is even twice as faster, whereas for Redis the performance improvement is non-negligible but lower. This is because, in the case of Neo4j, once the query plan

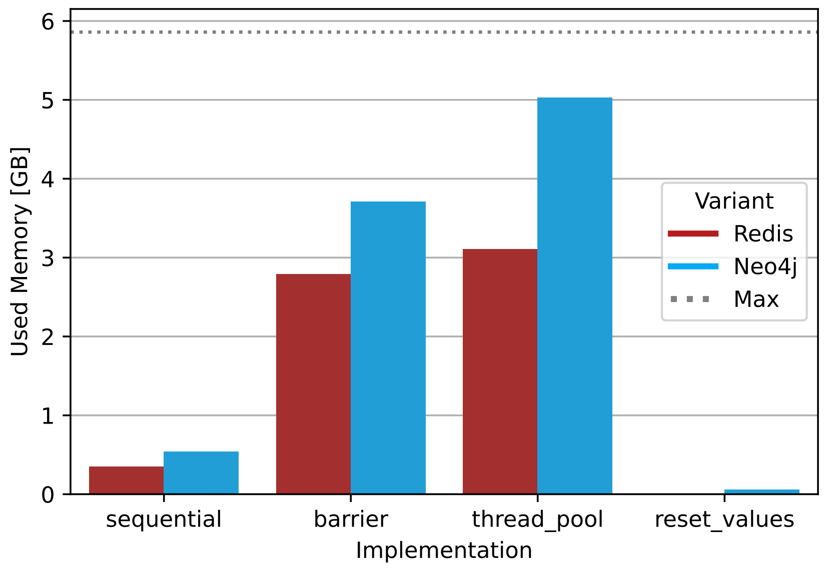

Figure 4.3: Memory comparison while loading graph.

to insert nodes is computed and optimized, several queries are executed at a stable rate, whereas with Redis we observe a more variable latency among different chunk insertions. In the third row, we can see how this factor actually plays a role in significantly improving Redis performance while loading a graph, as the time needed to entirely load the data drops to one-third of the baseline, i.e., 80 seconds. As for Neo4j, since the goroutines in the same waiting group execute at a similar speed, the improvement from the second row is more limited, also because there are some unavoidable penalties in terms of latency, i.e., downloading the chunk from S3, parsing, and unmarshaling, which are not optimized in this analysis. At the same time, the peak memory consumption increases towards the maximum allocated, with Neo4j consuming more memory as the driver requires to create auxiliary arrays of vertex data each time a graph chunk is loaded.

Finally, as explained in Section 4.3.1, the last implementation group shows the implementation in which vertex information is not inserted again, rather the **Main** function only resets their value to the sentinel. Without special tuning and optimizations, we achieve a decent 75% performance improvement compared to Neo4j and, if we consider the loading time in the original Graphless work based on the sequential approach with Redis as MaaS, an 84% improvement. This result reflects our expectations, as the **Main** function does not require any further communication with S3, parsing, unmarshaling, or sending vertices

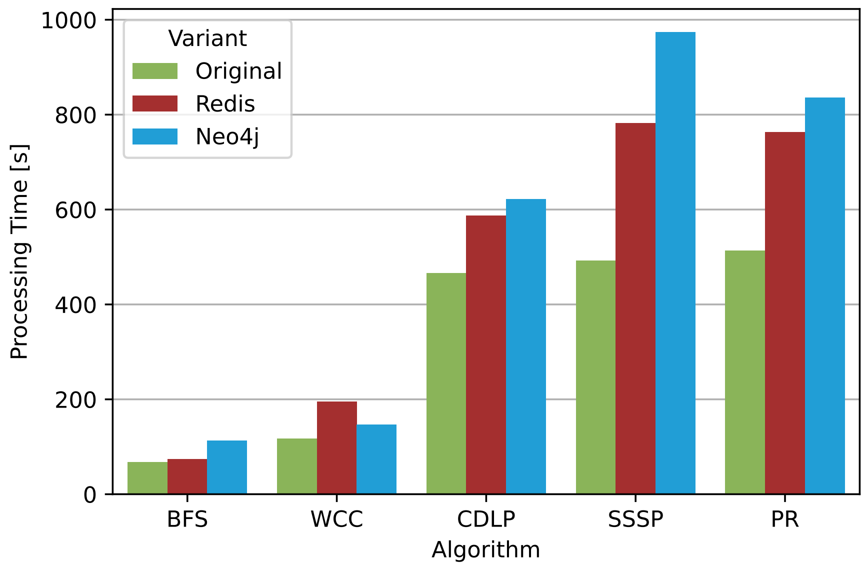

Figure 4.4: Processing time comparison among different MaaS backends.

encoded as JSON to MaaS, rather only executing a batched query on the existing data and, because of this, the function requires only 60MB memory. For the `reset_values` implementation, there are no results for Redis as this implementation can not be implemented with the current data model implementation with this MaaS backend as explained in Section 3.2.4.

4.4.2 Results for Processing a Graph

For the experiments regarding the alternative architecture, in Figure 4.4 we present the processing time for each combination between Graphless configuration and the algorithms, as previously described in Section 4.3.2. In these graphs, the algorithms are arbitrarily sorted on the x-axis by the lowest processing time obtained with the original implementation. For each algorithm, the first column ("Original") serves as a baseline by containing the results that we obtained when running the original implementation. On the other hand, the second column ("Redis") within each algorithm represents the version in which Redis is both being used for Graph and Message MaaS. Lastly, the third column ("Neo4j") represents the version in which we use Neo4j as Graph MaaS, even though Message MaaS is still backed by Redis. On the y-axis, we instead find the processing time for each combination of variant and algorithm.

Algorithm	**Supersteps**
BFS	5
WCC	4
CDLP	11
PR	11
SSSP	21

Table 4.1: Number of supersteps required by each algorithm.

Then, in Table 4.1, we can find the supersteps needed by each algorithm to complete when applied to the *dota-league* graph, including the final (empty) superstep in which the **Orchestrator** function detects the end of the execution. Some of these algorithms are endless by definition, e.g., PageRank, therefore LDBC Graphalytics defines a maximum number of iterations to be given as program input.

From the graph, a recurrent trend stands out for each algorithm, with the original implementation being the fastest to complete, whilst the one with Neo4j used as a Graph MaaS is the slowest and the Redis one in the middle. The first thing that can be observed is that the new Redis implementation runs now slower than the original one.

This is expected, as the one with split MaaS features throttling, which affects the speed of the fan-out in the **Orchestrator** function. Even though we observe **Worker** functions completing faster because of a more distributed load on the Message MaaS, the overall results are slower as the orchestration now requires around 10 seconds (400 workers / 16 workers per fan-out wave * 400ms of throttling) to completely invoke all workers in supersteps where all vertices are active. Therefore, in algorithms such as SSSP or PR, where the number of supersteps with all workers running at the same time is higher than alternatives such as BFS, the processing time sharply degrades for the implementation with Redis as Graph MaaS. As a side note, we also test the newly proposed architecture with Redis used in both MaaS components *without* throttling, where we achieve results in line, but not faster, with the original implementation. These additional results are presented in the Appendix B.

After explaining the difference between the two versions that only use Redis in the MaaS layer, we then notice that Neo4j does not perform significantly worse than the Redis counterpart, given the same throttling parameters, even though two important results stand out.

First, having Neo4j performing slower than Redis as a memory layer could be expected for several reasons: Neo4j efficiently executes queries that traverse graphs but, in this case, we are simply querying nodes by their identifier without spanning over a path, whereas Redis is designed and built for fast key-value querying patterns; then, a factor even more impactful, Neo4j actually persists and compacts graph data continuously to disk in the background, whereas Redis fully operates in memory and no thread is responsible for disk I/O, as the memory available in the AWS EC2 `r4.8xlarge` is enough to hold the whole graph data in memory without the need of using swap files.

Second, WCC is the only algorithm for which the Neo4j outperforms Redis as Graph MaaS backend with 196 against 291 seconds, i.e., a difference of approximately 100 seconds. The factor affecting this outlier relies upon the fact that, when an Amazon Lambda instance crashes, the AWS platform itself would retry it after some variable interval that is not under the control of the end user, hence for this and other runs are characterized by fluctuations in the processing time, with each Amazon Lambda usually being retried after 60 to 100 seconds. For mitigating these platform-dependent delays, we introduce an application-level bounded retry mechanism to both Graph MaaS implementation in addition to the respective Go database drivers thanks to which idempotent methods, e.g., `getVertices`, are retried for a maximum of 10 times with a random backoff ranging from 0 to 2 seconds. All the WCC executions with Graph MaaS backed by Redis are characterized by repeated errors, from which some Amazon Lambda invocations are not able to recover for a given superstep, hence resulting in an outlier. On the other hand, the same happens for some executions with Graph MaaS being Neo4j, e.g., SSSP. In both cases of Graph MaaS backed by Redis and Neo4j, the most frequent error consists of the database opening too many file descriptors, even if we increase the file limit for both installations as described in Section 4.2.2.

In Figure 4.5 we show an example of how we compute the error penalty due to Amazon Lambda retries. First, we identify all Amazon Lambda requests for a given superstep, i.e., λ_1 and λ_2 in the example. Then, each request might be invoked multiple times, which are counted with a superscript notation, e.g., λ_2' and λ_2'' are respectively the first and second attempt for λ_2. Then, we compute the corrected version of each request by adding the duration of the successful request to the first invocation, i.e., i.e., λ_1^{c} and λ_2^{c} in the example. By taking the latest end of all corrected requests, we determine the end of the corrected superstep. Finally, we compute the error penalty by subtracting the corrected superstep duration from the actual superstep duration.

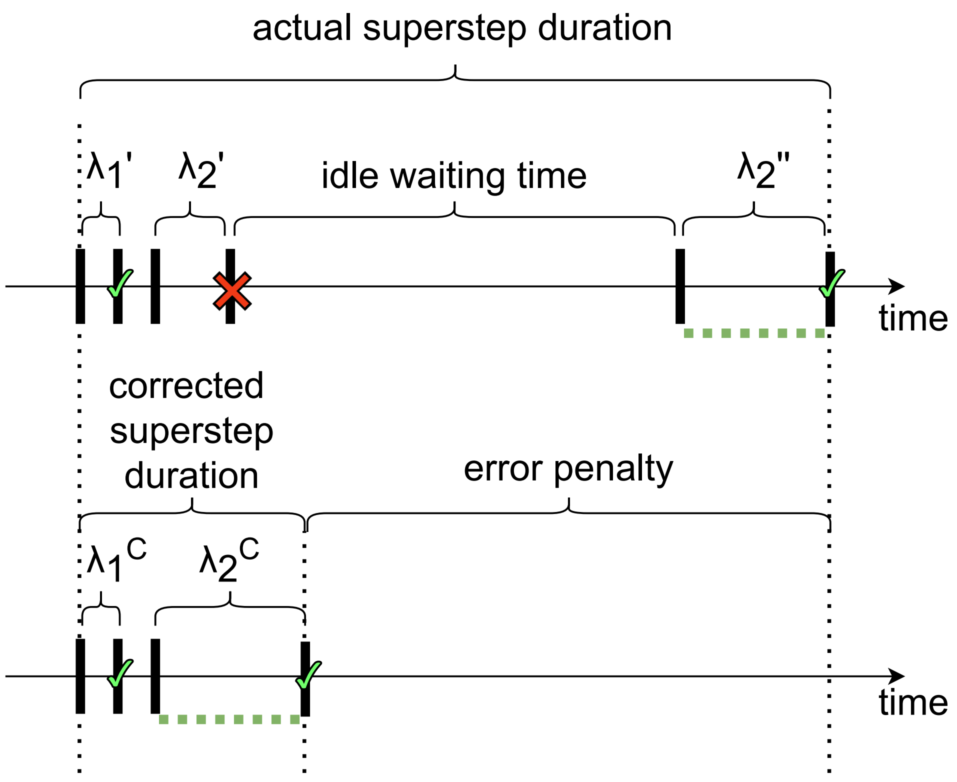

Figure 4.5: Computation of error penalty for Amazon Lambda retries.

Given this definition of error penalty, in Figure 4.6 we show the corrected processing time by subtracting this amount of time from the actual processing time already shown in Figure 4.4. In this graph, the error penalty is represented as a slashed bar stacked on top of the corrected processing time.

By looking at the corrected processing time, we observe that the newly proposed architecture with Redis used as a backend both for Graph and Message MaaS is significantly closer to the original processing time, where the difference can be explained because of the fan-out mechanism, as described earlier.

Finally, as a side note, an interesting collateral result deriving from the analysis of the Amazon Lambda retry mechanism lies in an observation that we made while running experiments. Because of the unpredictable delay applied by AWS as a platform in invoking again an asynchronous Amazon Lambda instance, the number of **Worker** functions crashing for a given superstep does not strongly influence the overall delay, as on the second attempt all of them would succeed as the concurrent load on MaaS and the number of active functions would be lower, assuming that a fraction of the **Worker**s invoked for the

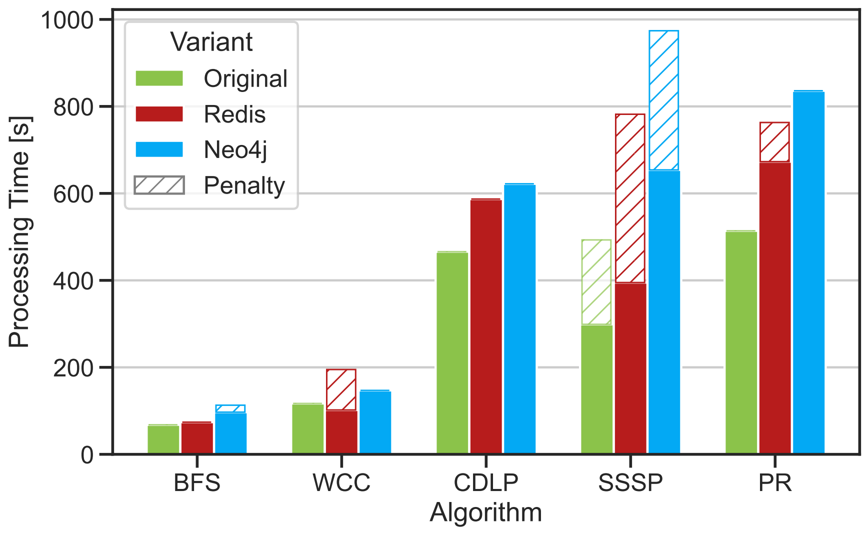

Figure 4.6: Processing time comparison among MaaS backends, with error penalty indicated.

superstep is already successful at the first invocation. Therefore, implementing a retry logic at the application level is highly beneficial for guaranteeing better performance.

4.4.3 Makespan Results

As we motivate in Section 1.1, final users of a graph processing technology are often interested in an inferior end-to-end duration of an algorithm execution, i.e., the makespan, as in Graphless the loading component is unavoidable.

In Figure 4.7, we present a consolidated view that shows how two main technical contributions affect the makespan when comparing our implementation to the original one. At the same time, in Figure 4.8, we show the makespan with error penalties due to Amazon Lambda retries indicated.

In both graphs, on the x-axis we group executions for each variant by algorithm whilst, on the y-axis, we report the makespan computed by adding loading and processing time. The first column of each group ("Original") serves as a baseline by containing the makespan that we obtain when running the original implementation "as-is", i.e., combining the loading time measured when using the sequential approach (cf. Section 3.2.1) and the processing time with a unique MaaS. Then, the second column ("Redis") represents Graphless with split MaaS backed by Redis, in which we choose the thread-pool-based loading technique since it achieves the most favorable results for Redis used as Graph MaaS, as shown in Section 4.4.1. Lastly, the third column ("Neo4j") represents Graphless with split MaaS

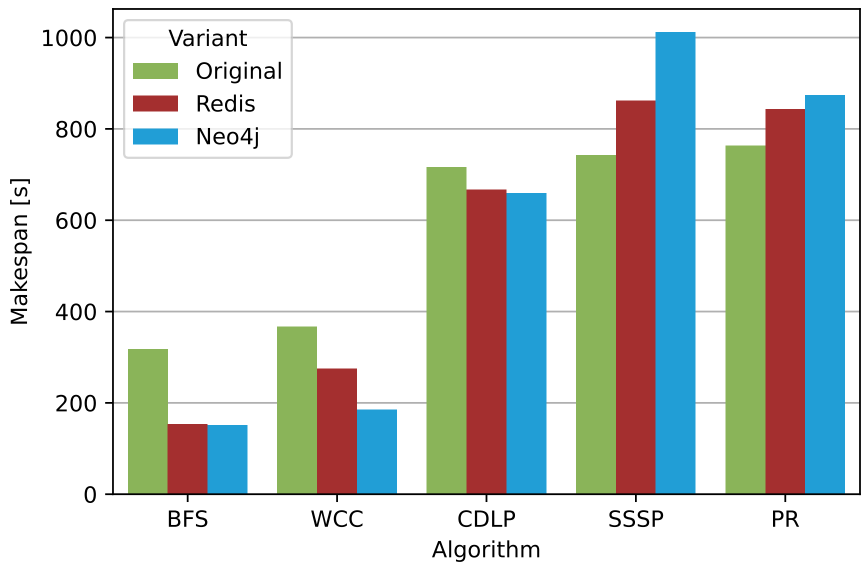

Figure 4.7: Makespan comparison among different MaaS backends.

with Neo4j backing the Graph MaaS, for which we use the value reset technique presented in Section 3.2.4 because, also in this case, this choice yields the best performance in terms of loading.

The variety inherently present in the LDBC Graphalytics benchmark in terms of algorithms allows us to discover workload-dependent behavior. In fact, for brief and sparse workloads, e.g., BFS, new loading techniques allow us to obtain a halved end-to-end execution time with a reduction of hundreds of seconds in terms of makespan, with our version outperforming the original one by several hundreds of seconds. On the other hand, a non-optimized throttling logic characterizes algorithms with a high number of supersteps and active vertices per superstep, e.g., SSSP and PR, thus resulting in penalties at processing time, which in this case outweighs the benefits gained by applying an improved loading process.

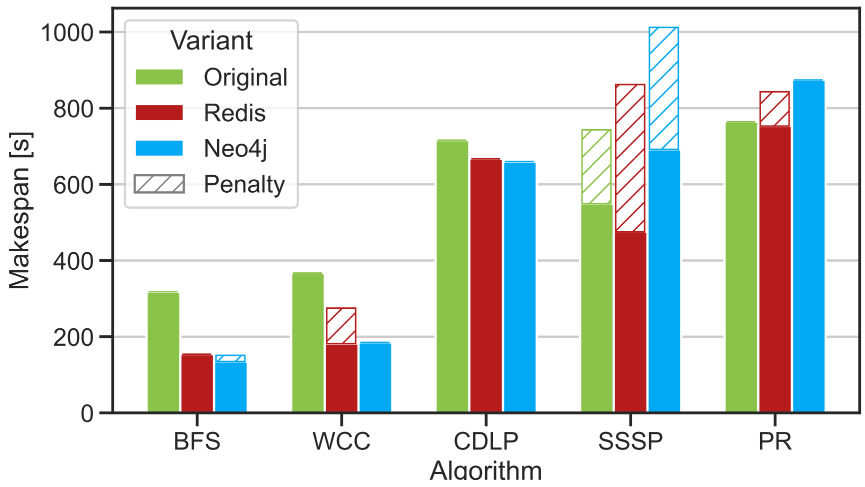

Figure 4.8: Makespan comparison among different MaaS backends, with error penalty indicated.

4.5 Summary

In Section 4.4, we report the results for the first two research questions. More precisely, Section 4.4.1 covers improvements regarding the loading time, whilst Section 4.4.2 shows how different MaaS backend can perform in terms of processing time, and finally Section 4.4.3 provides a holistic view of our results. Efficient loading techniques contribute to reducing the end-to-end duration by hundreds of seconds, whereas the newly proposed architecture enables the addition of new backends to store our graph data.

The combination of the results from the previous sections allows us to draw an interesting conclusion. We collect empirical data that shows how, even with some overhead caused by the throttling introduced in the fan-out executed by the **Orchestrator** during the processing phase, we obtain very favorable results using a different loading technique, thus justifying the choice of our proposal over the original one in terms of makespan.

5

Lessons Learned Along the Exploration

In this chapter, we discuss some of the key learning points we encountered throughout the experimentation part of this thesis work.

Function-as-a-Service platforms are hard to handle.
Contrarily to a process running on a machine, Amazon Lambda are not completely under the control of the operator. When owning a machine, an operator can monitor a process, check the current amount of open files, CPU usage, memory reserved and in use, and manage the running program by (forcefully) stopping or restarting it. On the other hand, when it comes to AWS Lambda, there is no control to immediately stop one or more invocations and immediately terminate their execution. Such a shortcoming can be decisive in terms of monetary costs, especially in the case of multiple running instances, in which case it is also difficult or not possible to get real-time information on the amount of Amazon Lambda functions that are currently running or scheduled to be executed. Furthermore, as previously mentioned in Section 4.4.2, failing Amazon Lambda invocations are retried by the platform itself, with no control over the delay after which an execution will be again scheduled or over the number of retries, which is fixed to three attempts in total.

In our case, this resulted in additional expenses due to long-running invocations and obsolete Amazon Lambda requests that would run concurrently to a newer execution. In the former case, often because of a bug in the **Worker** function, poorly performing queries, or the database being in a corrupted state, up to 400 Amazon Lambda instances would concurrently run until they would cause a timeout error. Especially when relaxing the timeout configuration for these functions, this could potentially result in 400 instances running for 10 minutes, which in turn would mean a high monetary cost as Amazon Lambda is priced proportionally to both allocated memory and execution time. Therefore, being unable to stop such long-running instances, we would wait for their termination and, in the meantime, deploy a version with an extremely short timeout, e.g., 2 seconds. On the other hand, obsolete Amazon Lambda requests are even harder to detect, as within the context of Graphless there might be even 400 instances running simultaneously. Out of these 400, finding all the failing invocations in real time could be compared to finding a needle in a haystack, as logs in Amazon CloudWatch[1] arrive with a non-negligible latency and such failing invocations should be checked individually to verify whether the three retry attempts have been consumed. From our experience, the best option to address this concern is waiting for 10 minutes before starting a new Graphless execution, as obsolete requests might corrupt the metadata required to coordinate the workers, which is stored in the same MaaS instance.

Security can be as important in research as in production software.
When setting up the original Graphless project, we noticed that there was no security in place regarding access to MaaS, i.e., a Redis group of shards running on EC2 instances publicly exposed to the Internet. Initially, we were not particularly concerned about it, since we deemed Amazon EC2 public IP addresses hard to guess and we thought that our instances would not become the target of an attack. Nevertheless, in recent years lucrative activities such as crypto mining have caused a spike in attacks on instances of popular databases that were found publicly exposed in the cloud[2,3]. Unfortunately, our publicly exposed instances were also attacked, even though we initially attributed the intermittently corrupted state inside the Redis instances to a bug in the Graphless program or the Redis configuration. After noticing that Graphless would reliably work with Neo4j, we inspected more closely the situation and we discovered Base64-encoded commands stored in our Redis instances, most likely to carry out remote code execution on our EC2 instances. To defend against this type of attack, we adopted two measures, i.e., securing the Redis instances with basic authentication and exposing a different port range (7379-7395 instead of 6379-6395) to prevent our MaaS shards to be easily detected by automated scans. Alternatively, we could have also restricted access to our EC2 instance within an Amazon Virtual Private

[1]a log aggregator for the Amazon ecosystem

[2]`https://www.trendmicro.com/en_us/research/20/d/more-than-8-000-unsecured-redis-instances-found-in-the-cloud.html`

[3]`https://www.trendmicro.com/en_nl/research/20/d/exposed-redis-instances-abused-for-remote-code-execution-cryptocurrency-mining.html`

Cloud (VPC), but this solution would have caused more billing costs due to additional infrastructure and a more complex setup as Amazon Lambda would require new VPC endpoints to be set up to work correctly inside a VPC.

Do not blindly trust claims taken out of context.
As reported by the original Graphless work, exchanging messages currently constitutes the bottleneck that makes Graphless itself perform worse in terms of processing time when compared to alternatives such as GraphMat. To address this concern, we also set up Amazon Lambda to run within an Amazon VPC so that function invocations could have access to Amazon ElastiCache and Amazon MemoryDB endpoints, as also mentioned to be a possible extension in the original Graphless work. Such solutions promise to unlock microsecond latency and scale with in-memory caching to offer ultra-fast performance, therefore we tried to use it for the Message MaaS layer presented in Section 3.3.2. Unfortunately, after some preliminary testing, we soon drew two important conclusions that determined the unfeasibility of this approach. First, this Redis-backed MaaS layer would entail higher monetary costs for the same performance, in turn receiving a fully-managed cluster. Second, Graphless executions would not terminate due to timeouts, with longer cold-start periods before the code inside an Amazon Lambda would be run as now Amazon would require to set up the VPC in addition to the function runtime and a greater variability when accessing the MaaS layer, with some queries taking up to approximately 3 seconds.

Take time to optimize your data model and queries.
After starting the implementation of the Graph MaaS backed by Neo4j, spending efforts in carefully mapping the data model as presented in Section 3.3.2 resulted in later benefits. After some preliminary testing, we noticed that all of the executions were resulting in timeouts and we, therefore, decided to investigate the root cause behind them. When analyzing the logs, we noticed a high latency in reading values from Neo4j, hence we generated a query plan for our reads by using the `EXPLAIN` command[1]. In this way, we noticed that, for example, retrieving vertices by their `id` property[2] would require a linear scan of the database. Therefore, we set up uniqueness constraints and indexes for efficiently querying our data, thus gaining read queries faster by two orders of magnitude in terms

[1]`https://neo4j.com/docs/cypher-manual/current/query-tuning/`

[2]here we refer to the vertex identifier stored as a property, *not* to the numerical identifier automatically assigned to Neo4j nodes

of wall-clock time. After this improvement, we noticed that our writes were significantly slower than their respective Redis counterparts, for which we instead tried to apply best practices from the Neo4j community[1]. In this case, using `UNWIND`, i.e., a command that allows to group multiple read or write operations into a single request, and batching, i.e., limiting the maximum size of `UNWIND` requests, allowed us to make our queries 20 times faster again in terms of wall-clock time.

Thoroughly test concurrency protocols at scale.
Transitioning from Redis to Neo4j also required us to redesign some protocols, such as solving the problem of detecting the termination of a superstep, which we thoroughly describe in Section 3.3.2. The protocol that we design is correct, but its implementation required non-negligible testing and debugging efforts. A factor that accentuated this problem relied on deploying code to Amazon Lambda without being able to test it at scale with local runs, as simple graphs with up to 20 vertices were not large enough to allow errors to manifest themselves and, at the same time, real-world graphs such as *dota-league* require an amount of resources which is not always available on a machine owned by a researcher. By deploying such faulty code to Amazon AWS, we had to use techniques previously described in this chapter to interrupt Amazon Lambda invocations, thus preventing endless executions and high monetary expenditure. Therefore, for this challenge, we decided to run as concurrently as possible algorithms on simple graphs, e.g., assigning one **Worker** function to one vertex only for local runs, or combining local and cloud infrastructure when debugging real-world graphs, that is, running the functions on a local machine for better control and connecting to a MaaS layer hosted on Amazon EC2 to provide an adequate amount of resources.

[1]`https://neo4j.com/blog/cypher-write-fast-furious/`

6

Conclusion

This final chapter concludes the document by first analyzing how our findings answer the research questions we posed ourselves. Finally, we enunciate limitations currently existing in our work and indicate directions for future expansion and research.

6.1 Answers to the Research Questions

After validating the results obtained in the original work and extending Graphless with operational techniques, we confirm that this project is a viable option for graph processing. Its unique feature, i.e., being a serverless framework for graph processing, allows Graphless to be singled out as a valid choice for small and medium enterprises that are interested in running graph algorithms on their own graph data.

While other options available on the market often require either incurring billing or purchasing costs or expertise in setting up a clustered environment, Graphless offers an open-source implementation that can be set up with automated configuration on Amazon AWS and one-click deployments.

In Section 1.2, we pose some research questions regarding serverless technologies, hence giving motivation to the exploratory work we conduct in this work. Hereafter, we aim to provide answers based on our empirical findings:

RQ1. How to implement more efficient graph loading techniques?
By exploiting concurrency, we improve both loading time by half and resource utilization by one order of magnitude. The best performing concurrent technique, i.e., a thread pool consuming graph chunks from a work queue, is particularly attractive as it does not make any assumption on the data model used to store topological information in Graph MaaS. Nonetheless, by relaxing this constraint and assuming

that the value of a vertex is saved as a distinct property, we can achieve a 3-fold loading time improvement and we can request a minimal amount of resources for the **Main** function.

RQ2. Would different graph processing architectures based on serverless computing affect the processing time?
We propose a new architecture for Graphless in which the central memory component, i.e., MaaS, is split into two dedicated data stores, i.e., one to store messages and another to store graph and orchestration data. By lowering the rate at which the fan-out in the **Orchestrator** function happens and introducing retry logic at application level, we obtain a time penalty at processing time up to approximately 300 seconds. At the same time, we observe that network and MaaS instability cause the largest fluctuations in Graphless performance, with individual iterations remaining idle for approximately one minute each time a non-recoverable error occurs.

RQ3. How to conduct quantitative and repeatable experiments?
We provide an implementation of a Graphalytics driver to enable reliable and repeatable measurements. Our deliverable has minimal dependencies as it only demands an Amazon AWS to be correctly configured on the local machine, on top of what the core of Graphalytics already requires. This driver is capable, after specifying the path to the Graphless source folder, of independently running a Graphless execution and conveniently extracting features of interest, e.g., makespan, from its results.

6.2 Future Work

After our exploration of serverless graph processing techniques, our experience benefits from several lessons learned along the way, which we present in Chapter 5. On the other hand, we also see opportunities for Graphless to grow as a more mature tool for graph processing, which are described in the following subsections.

6.2.1 Extend Support for Graph MaaS Backends

With the newly proposed architecture for **RQ2**, we provide an implementation for a Neo4j Graph MaaS backend and empirical proof of its feasibility. When extending the project in this direction, we realize that splitting the data model and, subsequently, the MaaS central memory component allows Graphless to fulfill a non-functional requirement overlooked at the beginning, i.e., lower the operational efforts needed to set up an ETL pipeline to prepare the input for the serverless framework. End users of Graphless are likely to store graph vertices and edges in databases such as Neo4j or other ACID alternatives, whereas companies would arguably store graph data in Redis more rarely. By not limiting the whole MaaS to run on Redis or another message broker, i.e., technologies that can fulfill performance requirements to exchange messages, we enable users to potentially attach Graphless directly to their existing graph database, without needing to put in place an additional pipeline to copy the graph data to first Amazon S3 and then to a separate data store, i.e., Redis. Therefore, we expect room for growth for Graphless in supporting multiple Graph MaaS backends, as this would allow more and more companies to easily integrate this framework within their ecosystems.

6.2.2 Try New Message Brokers

In the context of **RQ2**, decoupling the messages component in the data model enables swapping Redis with other message brokers, for which we identify two use cases. First, alternative middleware could offer better throughput, e.g., Hoang [46] shows that Rocket-Buf can achieve better performance than Redis. Second, a company might already have a cluster running a specific message broker, e.g., RabbitMQ or Kafka [47], hence installing Redis on a new set of machines might imply an economic investment in terms of human workforce.

6.2.3 Improved Resilience Logic

As we elaborate in Section 4.4.2, even a single failing Amazon Lambda invocation causes the whole execution to be delayed by approximately one minute. Therefore, a limitation of the current implementation relies in the fact that many isolated failures distributed over an execution affect the framework more severely than all the same failures happening within a short time frame. To address this shortcoming, we see two possible remediations. First, the logic to retry operations at an application level has room for improvement, as it now awaits for a short random amount of time before attempting again, in most cases without applying a specific behavior depending on the exact error cause. Then, Amazon AWS might offer more granular configuration options in terms of retry or a longer maximum timeout period, in which case further research and investigation would be needed to reach an optimal trade-off between reduced waiting time and MaaS stability.

6.2.4 Better Idempotence

Finally, we observe that Graphless does not implement idempotent for any arbitrary read-write operation to MaaS. For example, if a **Worker** invocation crashes after processing half of its workload, some messages might be sent twice as the **Worker** would restart with no awareness of what has already been processed. To address this, we should either design a different data model or store additional metadata in MaaS, as sending duplicate messages could cause some algorithms, e.g. PageRank, to produce slightly incorrect results.

6.2.5 Parallel Runs

In the original implementation, Graphless was thought to be run in complete independence. Now, with the central memory component split into Graph and Message MaaS, users could potentially use their own graph data store for Graph MaaS as discussed in Section 6.2.1. Nevertheless, it might be interesting to run simultaneously multiple algorithms, or instances of the same algorithm, on the same graph. For example, n users could run in parallel the Single Source Shortest Path (SSSP) algorithm to determine different n spanning trees starting from different n source vertices. Even though the current Graphless implementation does not support this, parallel runs could run on the same graph by using different properties and run identifiers to be processed independently from each other.

References

[1] Lucian Toader, Alexandru Uta, Ahmed Musaafir, and Alexandru Iosup. **Graphless: Toward Serverless Graph Processing**. In Alexandru Iosup, Radu Prodan, Alexandru Uta, and Florin Pop, editors, *18th International Symposium on Parallel and Distributed Computing, ISPDC 2019, Amsterdam, The Netherlands, June 3-7, 2019*, pages 66–73. IEEE, 2019. iii, 1, 3, 5

[2] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael J. Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and Peter A. Boncz. **LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms**. *Proc. VLDB Endow.*, **9**(13):1317–1328, 2016. iii, 1, 5, 17

[3] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. **The future is big graphs: a community view on graph processing systems**. *Commun. ACM*, **64**(9):62–71, 2021. 1

[4] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Ro-

dric Rabbah, Aleksander Slominski, and Philippe Suter. **Serverless Computing: Current Trends and Open Problems**. In Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya, editors, *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017. 1

[5] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. **Serverless Computing: A Survey of Opportunities, Challenges, and Applications**. *ACM Comput. Surv.*, 2022. 1

[6] Erwin Van Eyk, Lucian Toader, Sacheendra Talluri, Laurens Versluis, Alexandru Uta, and Alexandru Iosup. **Serverless is More: From PaaS to Present Cloud Computing**. *IEEE Internet Comput.*, **22**(5):8–17, 2018. 1

[7] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan S. Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. **Future Computer Systems and Networking Research in the Netherlands: A Manifesto**. *CoRR*, **abs/2206.03259**, 2022. 1

[8] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. **One Trillion Edges: Graph Processing at Facebook-Scale**. *Proc. VLDB Endow.*, **8**(12):1804–1815, 2015. 1, 15

[9] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. **GraphMat: High performance graph analytics made productive**. *Proc. VLDB Endow.*, **8**(11):1214–1225, 2015. 1, 16

[10] @Large Research. **Graphless**. `https://github.com/atlarge-research/graphless`, 2021. 5

[11] Maarten Van Steen. **Graph theory and complex networks**. *An introduction*, **144**, 2010. 7

[12] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. **Everything you always wanted to know about multicore graph processing but were afraid to ask**. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 631–643. USENIX Association, 2017. 7

[13] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. **One Trillion Edges: Graph Processing at Facebook-Scale**. *Proc. VLDB Endow.*, **8**(12):1804–1815, 2015. 8, 15

[14] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. **Pregel: a system for large-scale graph processing**. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010. 8

[15] Jeffrey Dean and Sanjay Ghemawat. **MapReduce: simplified data processing on large clusters**. *Commun. ACM*, **51**(1):107–113, 2008. 8, 15

[16] Renzo Angles. **A Comparison of Current Graph Database Models**. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *Workshops Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012, Arlington, VA, USA, April 1-5, 2012*, pages 171–177. IEEE Computer Society, 2012. 9, 16

[17] Diogo Fernandes and Jorge Bernardino. **Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB**. In Jorge Bernardino and Christoph Quix, editors, *Proceedings of the 7th International Conference on Data Science, Technology and Applications, DATA 2018, Porto, Portugal, July 26-28, 2018*, pages 373–380. SciTePress, 2018. 9, 16

[18] Diogo Fernandes and Jorge Bernardino. **Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB**. In Jorge Bernardino and Christoph Quix, editors, *Proceedings of the 7th International Conference on Data Science, Technology and Applications, DATA 2018, Porto, Portugal, July 26-28, 2018*, pages 373–380. SciTePress, 2018. 10

[19] Tobin J. Lehman and Michael J. Carey. **A Study of Index Structures for Main Memory Database Management Systems**. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 294–303. Morgan Kaufmann, 1986. 10

[20] Theo Härder and Andreas Reuter. **Principles of Transaction-Oriented Database Recovery**. *ACM Comput. Surv.*, **15**(4):287–317, 1983. 10

[21] Jinesh Varia, Sajee Mathew, et al. **Overview of amazon web services**. *Amazon Web Services*, **105**, 2014. 11, 12

[22] Russ Cox, Robert Griesemer, Rob Pike, Ian Lance Taylor, and Ken Thompson. **The Go programming language and environment**. *Commun. ACM*, **65**(5):70–78, 2022. 12, 22, 23

[23] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. **Apache Spark: a unified engine for big data processing**. *Commun. ACM*, **59**(11):56–65, 2016. 15

[24] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. **GraphX: Graph Processing in a Distributed Dataflow Framework**. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 599–613. USENIX Association, 2014. 15

[25] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. **GraphLab: A New Framework For Parallel Machine Learning**. In Peter Grünwald and Peter Spirtes, editors, *UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, July 8-11, 2010*, pages 340–349. AUAI Press, 2010. 16

[26] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. **Distributed GraphLab: A Framework for Machine Learning in the Cloud**. *CoRR*, **abs/1204.6078**, 2012. 16

[27] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. **PGX.D: a fast distributed graph processing engine**. In Jackie Kern and Jeffrey S. Vetter, editors,

Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015, pages 58:1–58:12. ACM, 2015. 16

[28] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sánchez. **Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling**. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, pages 1–14. IEEE Computer Society, 2018. 16

[29] Robert Ryan McCune, Tim Weninger, and Greg Madey. **Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing**. *ACM Comput. Surv.*, **48**(2):25:1–25:39, 2015. 16

[30] Bradley R. Bebee, Rahul Chander, Ankit Gupta, Ankesh Khandelwal, Sainath Mallidi, Michael Schmidt, Ronak Sharda, Bryan B. Thompson, and Prashant Upadhyay. **Enabling an Enterprise Data Management Ecosystem using Change Data Capture with Amazon Neptune**. In Mari Carmen Suárez-Figueroa, Gong Cheng, Anna Lisa Gentile, Christophe Guéret, C. Maria Keet, and Abraham Bernstein, editors, *Proceedings of the ISWC 2019 Satellite Tracks (Posters & Demonstrations, Industry, and Outrageous Ideas) co-located with 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, October 26-30, 2019*, **2456** of *CEUR Workshop Proceedings*, pages 189–192. CEUR-WS.org, 2019. 16

[31] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. **Jupyter Notebooks - a publishing format for reproducible computational workflows**. In Fernando Loizides and Birgit Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7-9, 2016*, pages 87–90. IOS Press, 2016. 16

[32] Pieter Cailliau, Tim Davis, Vijay Gadepally, Jeremy Kepner, Roi Lipman, Jeffrey Lovitz, and Keren Ouaknine. **RedisGraph GraphBLAS Enabled**

Graph Database. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 285–286. IEEE, 2019. 16

[33] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. **Formal foundations of serverless computing**. *Proc. ACM Program. Lang.*, **3**(OOPSLA):149:1–149:26, 2019. 16

[34] M. Garrett McGrath and Paul R. Brenner. **Serverless Computing: Design, Implementation, and Performance**. In Aibek Musaev, João Eduardo Ferreira, and Teruo Higashino, editors, *37th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 405–410. IEEE Computer Society, 2017. 16

[35] Scott Hendrickson, Stephen Sturdevant, Edward Oakes, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. **Serverless Computation with OpenLambda**. *login Usenix Mag.*, **41**(4), 2016. 16

[36] Zhitao Mao, Ruoyu Wang, Haoran Li, Yixin Huang, Qiang Zhang, Xiaoping Liao, and Hongwu Ma. **ERMer: a serverless platform for navigating, analyzing, and visualizing Escherichia coli regulatory landscape through graph database**. *Nucleic Acids Research*, 2022. 17

[37] Youngbin Kim and Jimmy Lin. **Serverless Data Analytics with Flint**. In *11th IEEE International Conference on Cloud Computing, CLOUD 2018, San Francisco, CA, USA, July 2-7, 2018*, pages 451–455. IEEE Computer Society, 2018. 17

[38] Arjun Singhvi, Sujata Banerjee, Yotam Harchol, Aditya Akella, Mark Peek, and Pontus Rydin. **Granular Computing and Network Intensive Applications: Friends or Foes?** In Sujata Banerjee, Brad Karp, and Michael Walfish, editors, *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*, pages 157–163. ACM, 2017. 17, 23

[39] Erwin Van Eyk, Joel Scheuner, Simon Eismann, Cristina L. Abad, and Alexandru Iosup. **Beyond Microbenchmarks: The SPEC-RG Vision for a Comprehensive Serverless Benchmark**. In José Nelson Amaral, Anne

Koziolek, Catia Trubiani, and Alexandru Iosup, editors, *Companion of the 2020 ACM/SPEC International Conference on Performance Engineering, ICPE 2020, Edmonton, AB, Canada, April 20-24, 2020*, pages 26–31. ACM, 2020. 17

[40] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. **SeBS: a serverless benchmark suite for function-as-a-service computing**. In Kaiwen Zhang, Abdelouahed Gherbi, Nalini Venkatasubramanian, and Luís Veiga, editors, *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*, pages 64–78. ACM, 2021. 17

[41] Pascal Maissen, Pascal Felber, Peter G. Kropf, and Valerio Schiavoni. **FaaSdom: a benchmark suite for serverless computing**. In Julien Gascon-Samson, Kaiwen Zhang, Khuzaima Daudjee, and Bettina Kemme, editors, *DEBS '20: The 14th ACM International Conference on Distributed and Event-based Systems, Montreal, Quebec, Canada, July 13-17, 2020*, pages 73–84. ACM, 2020. 17

[42] Bogdan Nicolae. **Understanding Vertical Scalability of I/O Virtualization for MapReduce Workloads: Challenges and Opportunities**. In Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, Stephen L. Scott, and Josef Weidendorfer, editors, *Euro-Par 2013: Parallel Processing Workshops - BigDataCloud, DIHC, FedICI, HeteroPar, HiBB, LSDVE, MHPC, OMHI, PADABS, PROPER, Resilience, ROME, and UCHPC 2013, Aachen, Germany, August 26-27, 2013. Revised Selected Papers*, **8374** of *Lecture Notes in Computer Science*, pages 3–12. Springer, 2013. 19

[43] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. **Understanding Real-World Concurrency Bugs in Go**. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 865–878. ACM, 2019. 22, 23

[44] Rob Pike. **Go at Google**. In Gary T. Leavens, editor, *SPLASH'12 - Proceedings of the 2012 ACM Conference on Systems, Programming, and Applications: Software for Humanity, Tucson, AZ, USA, October 21-25, 2012*, pages 5–6. ACM, 2012. 31

[45] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. **COSE: Configuring Serverless Functions using Statistical Learning**. In *39th IEEE Conference on Computer Communications, INFOCOM 2020, Toronto, ON, Canada, July 6-9, 2020*, pages 129–138. IEEE, 2020. 34

[46] Huy Hoang. *Building a Framework for High-performance In-memory Message-Oriented Middleware*. Master's thesis, University of Waterloo, 2019. 55

[47] Philippe Dobbelaere and Kyumars Sheykh Esmaili. **Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper**. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*, pages 227–238. ACM, 2017. 55

Appendix A

Reproducibility

A.1 Abstract

To reproduce the results we presented in this thesis, hereafter we provide a set of instructions on how to run the experiments using Amazon Web Services (AWS) or in a local environment.

A.2 Artifact check-list (meta-information)

- **Program:** Graphless extensions (go), Graphalytics driver (Java)
- **Compilation:** GNU make, go, JDK, Apache Maven
- **Binaries:** `main`, `orchestrator`, `worker`
- **Data set:** `https://ldbcouncil.org/benchmarks/graphalytics/`
- **Run-time environment:** go, JDK, Amazon AWS
- **Hardware:** local machine, Amazon Lambda, Amazon EC2 `r4.8xlarge`, Amazon S3
- **Metrics:** Loading time, processing time, makespan
- **Output:** Graphalytics, Amazon CloudWatch, Amazon S3
- **Experiments:** Graphalytics
- **How much disk space required (approximately)?** 3 GB
- **How much time is needed to prepare workflow (approximately)?** 30-60 minutes (including setting up an AWS account)
- **How much time is needed to complete experiments (approximately)?** 20 minutes to run BFS on a small graph, e.g., `example-directed`
- **Publicly available?** Yes
- **Code licenses?** Yes, Apache 2.0 for the Graphalytics driver for Graphless

A.3 Description

A.3.1 How to access

To get the code from GitHub, run the following command on your Command Line Interface (CLI):

```
$ git clone https://github.com/jmasic/graphless
```

The repository contains:

- Artifacts from the previous work
- **A3:** an implementation supporting Neo4j, and full support for local runs
- **A4:** an implementation with optimized Neo4j queries, improved loading, and full support for local runs
- **A5:** an implementation supporting Neo4j, with split memory component, further improved loading, full support for local runs, and backend choice done via configuration
- An LDBC Graphalytics driver for Graphless

A.3.2 Hardware dependencies

In this thesis work we use:

- a local machine on which we execute Graphalytics and local runs
- two Amazon EC2 instances, one for Graph MaaS and one for Message MaaS. The precise specifications can be found in Section 4.2.

When an Amazon EC2 instance is created, the only port open is by default 22 (SSH). For running MaaS, the following ports have to be opened:

- **Neo4j:** 7687 (Bolt) and, optionally, 7474 (UI reachable via HTTP)
- **Redis:** 7379-7395, i.e., one port per shard

A.3.3 Software dependencies

- `GNU make`, for executing Makefile targets
- `go` $\geq$ 1.17.x, for building Graphless and local runs
- `Apache Maven` $\geq$ 3.8.x, for building Graphalytics
- `JDK` $\geq$ 17, for running Graphalytics
- `Python` $\geq$ 3.9.x, for transforming LDBC graph datasets in input suitable for Graphless
- `AWS CLI` $\geq$ 2.4.x, for deploying the cloud stack and triggering Graphless executions
- `docker`, for local runs
- `github.com/mailru/easyjson` (go package), for making changes to JSON binaries used in function invocations

A.3.4 Data sets

In this work we used graphs available at `https://ldbcouncil.org/benchmarks/graphalytics/`:

- `example-directed`, for testing. Size: 10 vertices and 17 edges; and
- `example-undirected`, for testing. Size: 9 vertices and 12 edges; and
- `dota-league`, for testing and evaluation. Size: 61,170 vertices and 50,870,313 edges.

A.4 Installation

A.4.1 Graphless

Initialize the Graphless go module locally with:

```
$ go mod init "github.com/devLucian93/thesis-go"
```

A.4.1.1 MacOS Installation

Build Graphless for MacOS with[1]:

```
$ make clean-local build-local
```

[1]we use the suffix "`-local`" for the MacOS targets as we run local tests on a MacBook

A.4.1.2 Linux Installation

Build Graphless for Linux with:

```
$ make clean build
```

A.4.1.3 Amazon Installation

Execute the steps described in Section A.4.1.2.

Deploy the cloud stack, i.e., the Amazon Lambda functions, to AWS:

```
$ bash deploy.sh -t false
```

Run on Amazon EC2 either of the following scripts depending on the technology implementing the MaaS component:

- `neo4j_script.sh` to install and configure Neo4j:

  ```
  $ ./neo4j_script.sh
  ```

- `redis_script.sh` to install Redis:

  ```
  $ ./redis_script.sh 16 7379 redis
  $ ./redis_script.sh 16 7379 start
  ```

A.4.2 Graphalytics

Download the Graphalytics framework:

```
$ git clone https://github.com/ldbc/ldbc_graphalytics
```

Build the Graphalytics framework[1]:

```
$ mvn clean install -DskipTests=true
```

Build the Graphalytics Graphless extension, available in the provided sources:

```
$ mvn clean package
```

A.5 Experiment Workflow

We describe two different setups, one for local experimentation and the other for experimentation on Amazon AWS.

[1]this might not be necessary if you have access to the LDBC council Graphalytics Maven repository

A.5.1 Local Experiment Workflow

We run local experiments on a MacBook, with the MaaS components executing as Docker containers to remove the burden of installing them as separate applications on our machine.

We run Neo4j containers on Docker with the following command:

```
$ docker run --rm --name neo4j \
    -p 7687:7687 -p 7474:7474 \
     --env NEO4J_AUTH=neo4j/n \
    neo4j:4.4.6
```

We run Redis containers on Docker with the following command:

```
$ docker run --rm --name redis \
    -p 6379:6379 \
    redis:6.2.6-alpine3.15
```

Before starting Graphless, the `local_payload.json` file can be modified as specified in the `README.md` file of the `A5` implementation.

To start Graphless locally, run the following command from the desired Graphless implementation folder (`A3`, `A4`, or `A5`):

```
$ ./bin-local/main_function/main_function local
```

A.5.2 Amazon AWS Experiment Workflow

Before starting Graphless, the `main_payload.json` file can be modified as specified in the `README.md` file of the `A5` implementation.

To start Graphless on Amazon AWS, run the following command from the desired Graphless implementation folder (`A3`, `A4`, or `A5`):

```
$ bash start.sh --payload main_payload.json
```

For accessing the logs on Amazon AWS, we use Amazon CloudWatch or, alternatively, we install `awslogs`[1] for bulk downloads from Amazon CloudWatch. The latter solution enables us to store the logs on our machine and process them with scripts to extrapolate insights such as ingestion delay or the error penalty due to Amazon Lambda retries.

A.6 Evaluation

First, configure the Graphalytics Graphless driver as described in the `README.md` file.

For evaluating Graphless on Amazon AWS, run the following command from the Graphalytics Graphless driver folder:

[1]`https://github.com/jorgebastida/awslogs`

```
$ ./init.sh $path_to_graph aws compile
```

Alternatively, for evaluating Graphless on Amazon AWS, run the following command from the Graphalytics Graphless driver folder:

```
$ ./init.sh $path_to_graph local compile
```

When running these commands on a terminal, we advise to redirect the output into a separate file as the Graphalytics logger, i.e., Log4j, might print in colors which have low contrast with the CLI background.

Finally, when the execution of Graphalytics terminates, you can find the outputs in the `graphalytics-1.4.0-graphless-0.1-SNAPSHOT/report` folder. Since the contents of this folder are erased at every Graphalytics run, we advise to make a copy at a separate location to avoid losing test results.

Appendix B

Additional Experiments

In this chapter, we present additional experiments which are omitted from Chapter 4 for brevity.

In Figure B.1, we can see the processing time on the y-axis, in seconds, for all the algorithms, on the x-axis, that we take into consideration for the other experiments. Instead of the `Neo4j` column, there is a new variant `Redis_w/o`, which represents executions with Redis used both as Graph MaaS and Message MaaS, with no throttling while the **Orchestrator** invokes the **Worker** functions at the beginning of a new superstep.

In Figure B.2, we instead visualize the makespan on the y-axis, in seconds, and keep the same x-axis and variants as in Figure B.1.

For running these additional experiment runs, we use the same setup as described in Section 4.3.2.

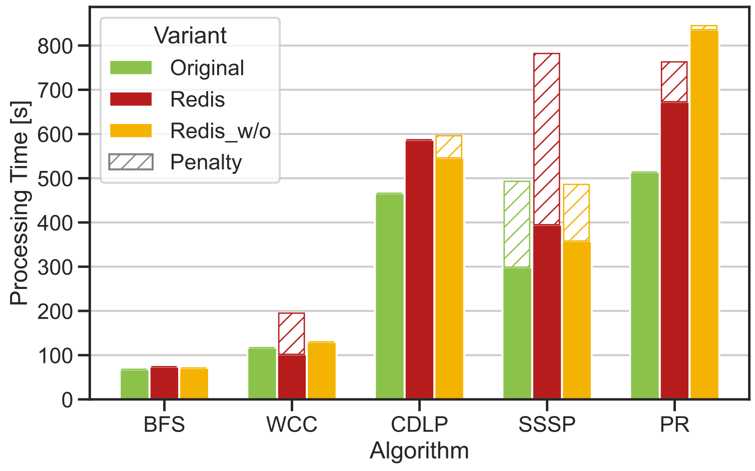

Figure B.1: Processing comparison with correction among Original, Redis and Redis without throttling.

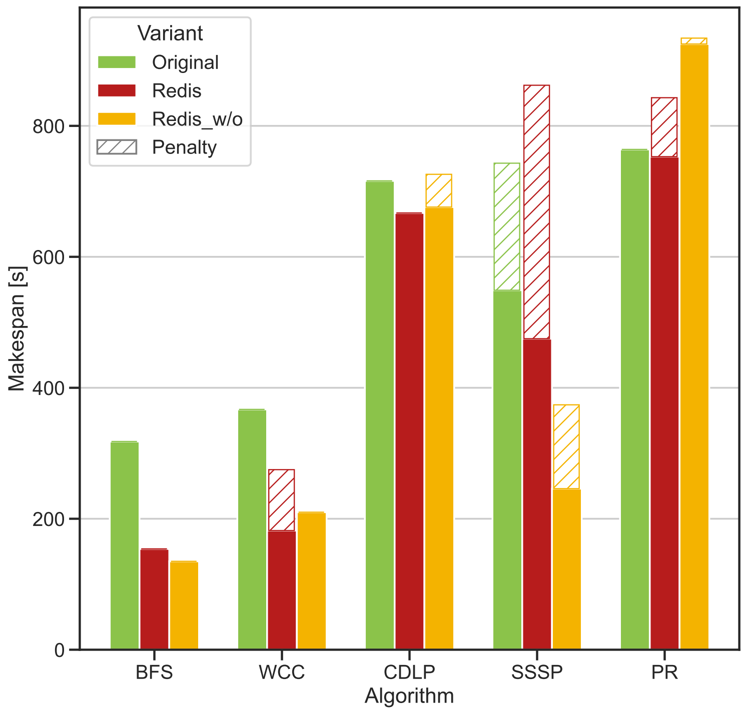

Figure B.2: Makespan comparison with correction among Original, Redis and Redis without throttling.

Appendix C

Results in Numbers

In this chapter, we provide numerical figures for the bar plots we present in Chapter 4. In Table C.1, we show the results we obtained for Section 4.4.1. In Table C.2, we instead show the results we obtained for Section 4.4.2, where "T" and "P" stand respectively for processing time and penalty.

Implementation	**Redis**		**Neo4j**	
	Time [s]	Memory [GB]	Time [s]	Memory [GB]
Sequential (Baseline)	238	0.35	153	0.54
Barrier	152	2.79	61	3.71
Thread Pool	80	3.11	56	5.03
Reset Values	-	-	38	0.06

Table C.1: Loading metrics (Figures 4.2 and 4.3).

Algorithm	**Original**		**Redis without throttling**		**Redis**		**Neo4j**	
	T [s]	P [s]	T [s]	P [s]	T [s]	P [s]	T [s]	P [s]
BFS	68	0	71	0	74	0	113	16
WCC	117	0	130	0	195	93	147	0
CDLP	466	0	596	50	587	0	622	0
SSSP	493	194	486	128	782	387	974	320
PR	514	0	845	9	763	90	836	0

Table C.2: Processing metrics (Figures 4.4, 4.6, and B.1).