SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Enhancing Graph Processing Efficiency in Kubernetes: Towards Application-Aware Scheduling

Jacek Kuśnierz

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Enhancing Graph Processing Efficiency in Kubernetes: Towards Application-Aware Scheduling

Verbesserung der Effizienz der Graphverarbeitung in Kubernetes: Hin zu einer anwendungsbewussten Planung

Author:	Jacek Kuśnierz
Supervisor:	Matthijs Jansen, MSc.
Advisor:	Prof. Dr. Viktor Leis
Submission Date:	15.08.2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.08.2024

Jacek Kuśnierz

Acknowledgments

I would like to thank Emilia Majerz and Rafał Mucha for proof-reading and reviewing this work.

I would also like to thank OpenAI and Anthropic for creating tools for simplifying programming and data processing - without your help, with the multitude of technologies being used here, reaching the state this thesis is would not have been possible.

Abstract

This thesis focuses on enhancing the efficiency of graph processing within Kubernetes clusters through application-aware scheduling techniques. By delving into the unique characteristics of graph processing workloads and their interaction with Kubernetes environments, the research aims to develop novel scheduler-application interface that maximize resource utilization and minimize latency. Through empirical analysis and experimentation, the study seeks to optimize scheduling decisions to improve overall system efficiency for graph processing tasks in Kubernetes clusters.

Contents

1 Introduction

Businesses and governmental authorities systematically collect and analyze data to optimize the allocation of resources and improve decision-making processes. This has resulted in an increasing demand for sophisticated data collection and processing methodologies, as evidenced by multiple studies [HL11; Dom23]. The exponential growth in internet data generation further reflects this trend, with global data production escalating from 2 zettabytes in 2010 to 147 zettabytes in 2023 — representing a 74-fold increase. Consequently, the average internet user now generates approximately 102 MB of data per minute [Sta23].

A subset of this data is in the form of graphs. The concept of the graph was created to model relationships and interactions inside systems containing multiple actors, such as a circle of friends having specific relations with each other. At its core, a graph uses a structure made up of points, called nodes or vertices, connected by lines, known as edges. Both of these can be further annotated with extra data, such as providing the number called "weight" to edges, for example representing a distance between two points on the map.

The ability to model complex relationships in a way that is easy to understand makes graphs useful for multiple audiences. In computer science, it helps improve how search engines find the most relevant information quickly [KMK14]. In social sciences, researchers can understand how people connect and influence each other in society [Ahm+20]. In biology, it is used to study how different elements like genes or proteins interact, which can be crucial for understanding diseases and finding new treatments [Li+19].

The field is currently booming as more and more data is represented in the graph form [Sak+21]. As the volume of graph data is expected to grow substantially in the coming years, there is an increasing need for innovative techniques to effectively manage and process this expanding data. To ensure that graph processing remains efficient and scalable, researchers and developers must explore and implement new methodologies that can accommodate the complexities and demands of larger and more intricate graph datasets.

1.1 Context — Scalable Processing and Graphs

Since this graph data is already known to be big and growing, there are multiple solutions to process it at scale. One example of such would be a popular distributed computing platform, Apache Spark [Zah+16]. Spark was designed for a specific type of computation — namely Map Reduce [DG08] — on separable, tabular data. It can also process other data representations via plugins, such as graphs with GraphX [Gon+14]. However, such a plugin has to respect the basic API of Spark. This API limits the way it reads and processes the data, creating an extra overhead. On the other hand, the system is very **Scalable**horizontally when adding processing units.

A key aspect of the problem of "understanding the graph" is how the graph is represented when not in use: the default approach to store it is in CSV-like format. From this representation, the distributed platforms are creating a more efficient structure like Hash Map or Linked List, which takes extra time and space before the analysis can be started. It is a default behavior inside efficient databases like Neo4j [Neo24a], but such databases do not scale well, being optimized only for single-thread execution. The first property that we would want then is a **Graph Native** solution like Neo4j, so our solution does not introduce the overhead of translating this format for every analysis, unlike existing distributed platforms like Spark.

To scale up, one approach is to replicate the entire processing system using a lower-level system like Slurm [YJG03], which allows users to reserve hardware for their data processing needs. However, this approach requires users to set up their own environment, and it does not allow any communication or implicit data and resource sharing. Some more advanced solutions have been developed recently [Wen+22], which can allocate just enough memory for the workload instead of using full instances. However, the downside of these multitenancy solutions is that they still operate at a low level, requiring users to deploy their own processing instances. This makes scaling inefficient, as it involves replicating all components of the system. The second property required for creating an ideal solution is that the platform must share redundant parts of the system and cache data to reduce memory consumption and analysis time for **Multitenancy**.

Another category of solutions involves interactive systems, which offer rapid results, enabling users to iterate quickly and improve the feedback loop. Systems like Neo4j [Neo23b] and Dgraph [Lab23] exemplify this approach by supporting interactive execution. However, these systems centralize data across a few nodes, with each node limited to a single execution thread per analysis. This architecture presents scalability challenges. To increase computational power, these systems require replicated instances with all components of the system, which is not ideal for distributed computing involving massive datasets. By definition, big data cannot fit on a single machine and

Table 1.1: Comparison of desired properties on existing systems.

Component	Scalable	Multitenant	Interactive	Graph Native
Neo4j	✗	✓	✓	✓
Spark + GraphX	✓	partly	✗	✗
GraphScope	✓	✗	✓	✓
This Work	✓	✓	✓	✓

requires sharding. In the case of Neo4j, this necessitates additional analysis of the dataset to query multiple independent instances simultaneously [Neo20]. Despite these limitations, Neo4j on a single machine remains highly efficient for most tasks, except when dealing with extremely large datasets [MSB23]. The next property is that we need **Interactivity** like in Neo4j, as it supports a good user experience, making this database system widely used.

1.2 Problem Statement

None of the systems specialized for graphs support multitenant processing for data at scale. There is also the problem of having the graph modified when all users are working on the same instance together, leading to errors and confusion. We can see that other fields, like Machine Learning in the case of AWS SageMaker [Ama24], have the issue of multi-access tackled. However, it being closed-source complicates the insight.

Each of these systems has its own strengths but often lacks one or more key properties such as scalability, multitenancy, interactivity, or understanding graphs. To address this gap, we propose creating a system that encompasses all of these attributes.

Because it's not possible to create such a system from scratch, we have decided to extend an existing solution. We closely examine the distributed Graph Processing Platform, GraphScope [Fan+21], which is developed and used in production systems at Alibaba. GraphScope Flex, an evolution of GraphScope, achieves up to 2,400× performance gain in real-world applications [He+23] over state-of-the-art cybersecurity monitoring systems on Alibaba Cloud. From the performance and scalability perspective, this is the best option to develop from, especially when considering Table 1.1, which shows other systems are lacking the more complex properties we want.

Scalability provided by GraphScope is the hardest problem out of these four qualities we want to achieve. It also supports **interactive** execution out of the box. It also has the ability to do **native graph processing**, so it does not have the extra overhead of using a plugin over a different system. The only missing quality is the multitenancy and reuse of resources, which we are introducing in this work.

1.3 Research Questions and Methodology

Based on the above properties, we construct 4 research questions leading towards an ideal, integrated solution with all desired properties. Therefore, we ask:

1. **Research Question 1: How to design a multitenant graph processing platform based on the GraphScope?**

   Multitenancy is a feature that is well-supported in platforms from other fields, such as:

   - Cloud Computing Tools, e.g. AWS SageMaker [Ama24].
   - Database Systems, such as PostgreSQL [The24c].
   - Container Orchestration, like Kubernetes [Clo].

   but it has not yet been implemented in any existing graph processing platform. To address this gap, we conduct a literature review of the existing solutions and their architectures, and we adapt the best parts of these onto our system. We especially analyze the design of their resource sharing and access controls to transplant this into the system. We then implement an open-source solution, extending the existing GraphScope system to incorporate multiple users.

2. **Research Question 2: How to enable interactivity in a multitenant graph processing platform?**

   When extending GraphScope to a multitenant system, it is important to consider that traditional graph processing systems are typically batch-based, leading to potential delays as users wait in a queue for resource allocation. However, incorporating interactivity offers significant benefits. It enhances user experience by providing immediate feedback, which is crucial for rapid iteration and decision-making. Interactivity also increases productivity by reducing downtime and supports more agile development processes. With careful resource management, it is possible to maintain scalability while delivering the responsive interactions the users need. We conduct a literature review to see how interactivity is enabled in existing systems.

3. **Research Question 3: How to efficiently share a single graph processing platform between users?**

   The simplest approach to implementing multitenancy in a system is to create multiple instances of the system, as done with Slurm [YJG03]. However, this method is not scalable, as it requires replicating all resources for each user, leading to inefficiencies. To address this, we can examine existing systems, draw insights

to minimize non-shared components, and subsequently evaluate the impact of adding more clients. We draw inspiration from other High Performance Computing (HPC) domains, such as Machine Learning as a Service (MLaaS) [Wen+22], to design and implement a new system. This system, similarly, provides Graph Processing as a Service.

4. **Research Question 4: How to share graph data between users, and what are the performance benefits?**

   Sharing in-memory data is a well-established technique across various fields, including Machine Learning [Kwo+23]. By caching models in RAM, this method significantly accelerates query processing by avoiding the repeated overhead of loading data from disk. This same principle can be effectively applied to graph databases. While earlier implementations have primarily focused on single-node configurations like Neo4j [Neo23b] or have been limited to single-session lifecycles as seen in Vineyard [Yu+23], the underlying logic remains consistent: performance gains are realized when data is cached in memory after its initial load.

In this study, we examine the performance improvements offered by our solution relative to the standard GraphScope system and other competitors like Neo4j. Our analysis centers on identifying and mitigating bottlenecks associated with data access and memory footprint.

To evaluate the research questions, we use the following benchmark scenarios:

1. Query processing time: Neo4j vs. scaling GraphScope if distributed approach gives meaningful improvements over single node optimized Neo4j.
2. Import time: how much can be saved by keeping the dataset for reuse instead of importing from the external source every time.
3. Deploy time: how long it takes to create a new system vs. attaching to an existing one and processing there.
4. Memory footprint: how we scale with n equivalent systems instead of sharing a single one between multiple clients.

1.4 Thesis structure

In Chapter 1, we introduce the problem and the possible solutions from similar fields. We also discuss the background and technical aspects of dealing with graphs, and also the shortcomings of existing approaches.

In Chapter 2, we describe in detail the technical background of the problem, as well as the architecture and design of technologies that will be useful for our system. We are focusing on multi-user parts of Spark and technical graph processing aspects of Neo4j and GraphX.

In Chapter 3, we present our design decisions, which lead to our final architecture of a multitenant system. We then describe the solution for multi-access, which introduces an extension to GraphScope, allowing for novel behavior in the graphs field.

In Chapter 4, we present the benchmark setup and evaluate our solution against the state of the art to get a precise picture of our improvement.

In Chapter 5, we conclude our findings and describe the possible further developments of the system.

2 State of the art

2.1 Graph Databases

Usually, the graph is a set of vertices, identified by IDs, and the set of edges, which are commonly a tuple of (source vertex, target vertex, weight/description). To be interoperable, it is usually stored in a tabular format containing said information. This can be later imported into one of the several databases specific for storing graphs. The most commonly used are Neo4j [Neo23b], OrientDB [Ori24], JanusGraph [The24b], and cloud-based services like Cosmos DB [Mic23] and Neptune [Ama23a]. While these databases can be distributed, the querying itself is usually limited to a single thread, which does not fully exploit the distributed nature of these systems.

Since we know that there is a storage that can hold this data and the databases themselves are quite limited, we can introduce something that can leverage the speed of access of the entity-aware database and power of distributed computing - Distributed Processing Platform.

2.2 Distributed Processing Platforms

Before dedicated solutions for distributed graph processing were developed, general-purpose platforms like Apache Spark [Zah+16] were adapted to work with graphs through tools like GraphX [Gon+14]. GraphX was the first tool on a popular platform that enabled distributed graph processing for the general public. Other systems back then, like Pregel [Mal+10] or A1 [Bur+20], were commercial, internal solutions, therefore limited to people inside the company. Widely used closed-source graph processing systems include Tigergraph [Tig] and Allegrograph [Inc24].

However, focusing on open-source alternatives, the very promising one that we see is Dgraph [Lab23], although the functionalities for the open-source version are still limited, and it uses JSON under the hood for synchronising the data, which is not optimal. There was a multitude of other systems on the way, such as Wukong [Shi+16] or Apache Giraph [Fou24], but these were purely academic and are abandoned at the time of writing.

2.2.1 Pregel: A Theoretical Foundation

Pregel is a distributed graph processing framework developed by Google, which fundamentally redefined how large-scale graph algorithms are implemented and executed in distributed environments. Its design is specifically tailored to handle graph datasets that are too large to fit on a single machine, making it a pioneering model for graph-parallel computation. Pregel's core concept is the *vertex-centric* programming model, where the computation is driven by the vertices of the graph rather than its edges. In this model, each vertex operates independently and concurrently, processing data, sending messages to its neighbouring vertices, and updating its state based on the messages it receives. This approach aligns with the natural parallelism in graph algorithms, where operations like graph traversal, shortest path calculation, and connectivity queries can be decomposed into parallel tasks that operate on individual vertices.

Pregel operates through a series of synchronized iterations called *supersteps*. During each superstep, vertices execute a user-defined function that processes incoming messages, updates the vertex state, and generates new messages to be processed in the next superstep. The process continues until a global termination condition is met, typically when all vertices are inactive and no messages are left to be processed. This structure ensures that the computation progresses in a lockstep fashion, providing a clear and predictable execution model that is crucial for large-scale distributed systems.

Pregel's vertex-centric model and superstep-based execution have profoundly influenced the design of subsequent graph processing frameworks and algorithms. It serves as the conceptual foundation for many modern systems like Apache Giraph, GraphX in Apache Spark, and Google's proprietary graph processing engines. These systems have adopted and extended Pregel's model, integrating it with their respective ecosystems to offer more flexibility, scalability, and efficiency.

For example, GraphX extends Pregel's principles to operate within the broader Apache Spark ecosystem, allowing seamless integration with Spark's data processing capabilities and enabling graph-parallel computations to be combined with other forms of data analytics available on Spark. This fusion of Pregel's model with the data flow paradigm of Spark enables users to leverage the best of both worlds: the simplicity and power of vertex-centric programming and the efficiency of distributed data processing.

Pregel has also inspired new research into optimizing graph processing in distributed environments, leading to advancements in handling large-scale graphs, improving fault tolerance, and reducing the computational overhead associated with graph-parallel processing. The introduction of abstractions like vertex programs, message-passing mechanisms, and superstep synchronization in Pregel has become the basis for developing more sophisticated algorithms that can efficiently process massive graphs across distributed systems.

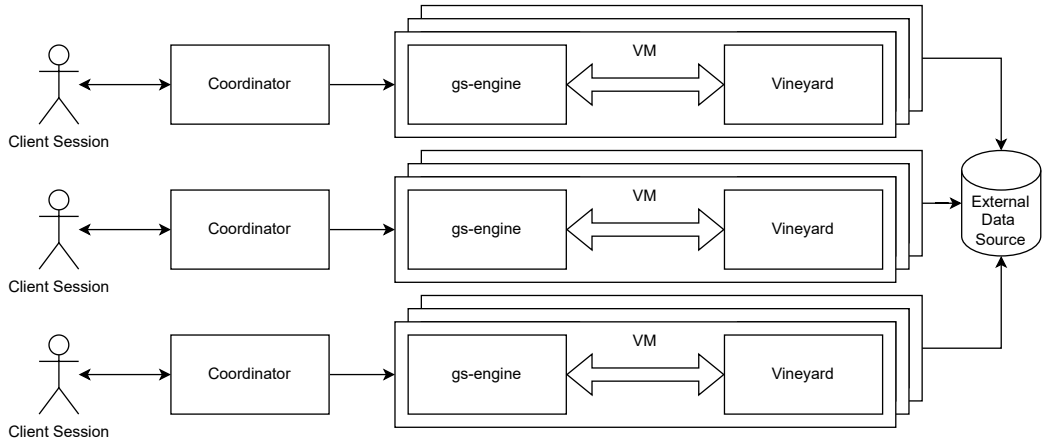

Figure 2.1: GraphScope architecture as it exists now for multiple users.

2.2.2 Apache Spark and GraphX

Apache Spark, a robust and widely used distributed data processing framework, plays a pivotal role in open-source graph processing, offering significant advantages in scalability and computational efficiency. Spark's graph processing capabilities are primarily realized through its GraphX [Gon+14] component, an API that provides a flexible and powerful platform for graph-parallel computations.

GraphX extends Spark's core functionalities to efficiently handle and analyze large-scale graphs. Traditional graph processing frameworks, such as Pregel [Mal+10], often require specialized environments, but GraphX integrates graph processing with Spark's existing data processing pipelines, allowing for seamless interoperability between graph and non-graph data.

Simulating Multitenancy using User Impersonation

Spark itself does not support multitenancy; each deployed cluster can serve only one user. Because of that, there is a workaround: user impersonation in Apache Spark can be utilized to create a pseudo-multitenant environment. This approach involves running Spark jobs from different clients under the same user identity, which includes sharing the same data and resources under the same name. This is available only in the case of Spark running on Yarn[Apab] clusters, and it is solving the problem of not having a real multitenancy on a singular cluster.

2.3 Neo4j

2.3.1 System Architecture

Neo4j is a native graph database built with an architecture optimized for handling graph structures. The core components of Neo4j architecture include:

- **Native Graph Storage:** Neo4j uses a native graph storage format, which stores graph data (nodes, relationships, and properties) directly, allowing for efficient graph traversal and querying. This contrasts with non-native graph databases that store graph data in relational or key-value stores, resulting in less efficient data retrieval.
- **Kernel:** The Neo4j kernel is responsible for managing transactions, data consistency, and ACID (Atomicity, Consistency, Isolation, Durability) compliance. It also provides low-level APIs for data manipulation.
- **Cypher Query Engine:** Cypher [Neo24d] is Neo4j's declarative graph query language, designed specifically for graph operations. The query engine parses, plans, and executes Cypher queries, optimizing them for performance. Cypher is so popular that it was integrated into several other frameworks, like Spark's GraphX.
- **Indexing and Caching:** Neo4j supports multiple indexing options (e.g., B-tree, full-text search) to speed up query execution. It also includes caching mechanisms to store frequently accessed data in memory, reducing I/O operations.
- **High Availability and Clustering:** Neo4j provides clustering support for high availability and horizontal scaling. It uses a master-slave architecture where write operations are handled by the master node, and read operations can be distributed across slave nodes — this feature is only available in Enterprise Edition, which for the purpose of this work was not accessible due to not being open-source.

Thanks to the above, Neo4j is a very powerful tool, which rivals other state-of-the-art solutions [MSB23].

2.3.2 Query Submission and Execution

Submitting a query in Neo4j involves interacting with the Cypher query engine. The process is as follows:

1. **Query Submission:** Users submit Cypher queries via Neo4j web interface, REST API, or Bolt protocol. A typical Cypher query consists of patterns that describe the nodes and relationships to be matched, along with optional clauses for filtering and projection.

2. **Parsing:** The Cypher query engine parses the query into an abstract syntax tree (AST), which represents the logical structure of the query.

3. **Planning:** The engine generates an execution plan from the AST, choosing between different strategies (e.g., index scans, label scans) based on query complexity and available indexes.

4. **Execution:** The execution plan is run by the query engine, traversing the graph and fetching the required data. The engine optimizes traversal operations using its native graph storage format.

5. **Result Delivery:** The results are returned to the user in the requested format, such as JSON, CSV, or visual graph representation.

2.3.3 Neo4j Graph Data Platform

The Neo4j Graph Data Platform is a new development that offers a flexible and scalable architecture designed to handle complex data relationships through a property graph model. This model represents data as nodes and relationships, optimized for efficient graph traversals. It integrates tools like the Cypher query language, Neo4j Browser, and graph-based applications to enable diverse use cases such as fraud detection and network monitoring. This architecture serves as an inspiration for modernizing our own technology, where we aim to redesign and extend these concepts using newer technologies to build a more advanced, adaptable platform.

2.4 GraphScope

The development that we focus on is GraphScope [Fan+21], which took the idea that we can use Vineyard [Yu+23] — ephemeral storage optimized for graph processing workload characteristics — between the executors. It also does not need dedicated cluster managers like Yarn [Apab] or Mesos [Apaa] used in Spark. Instead, it delegates all the work for scheduling the resources, as well as networking to the well-established Kubernetes [Clo] Platform as shown in Figure 2.1. It optimizes the entire flow from the job submission, reading the data, distributing the data to intermediate memory, distributed processing, up to collecting and returning the result. It keeps the data flow

```python
import graphscope

# Create a session
sess = graphscope.session()

# Load a simple graph
graph = sess.g().add_vertices(
    {
        'vertex': [(1, "A"), (2, "B"), (3, "C"), (4, "D")],
    },
    id_field="id"
).add_edges(
    {
        'edge': [(1, 2), (1, 3), (2, 4), (3, 4)],
    },
    src_label="vertex",
    dst_label="vertex",
)

# Run PageRank on the graph
pagerank_result = graphscope.pagerank(graph)

# Print the results
for node_id, rank in pagerank_result:
    print(f"Node {node_id} has PageRank: {rank}")

# Close the session
sess.close()
```

```python
from pyspark.sql import SparkSession
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.graph import PageRank

# Create a Spark session
spark = SparkSession.builder.appName("PageRank").getOrCreate()

# Load a simple graph
edges = spark.createDataFrame([
    (1, 2),
    (1, 3),
    (2, 4),
    (3, 4),
], ["src", "dst"])

# Run PageRank on the graph
pagerank = PageRank(resetProbability=0.15, maxIter=10)
model = pagerank.fit(edges)

# Print the results
ranks = model.vertices
ranks.show()

# Stop the Spark session
spark.stop()
```

Table 2.1: Comparison of GraphScope (left) and Spark GraphX (right) code for computing PageRank.

very similar to one known from Spark as seen in a comparison of sample analysis as seen on listings in Figure 2.1. The general flow of processing the analysis can be viewed in Figure 2.2. We describe in depth the components and interplay between them below.

Noteworthy to mention again is that GraphScope uses C++ stored procedures to process its queries, leading to great processing time improvements. The downside to this is that the procedures need to be compiled first, which adds some extra time; although once compiled, they are very efficient and can be reused in further executions, even by other sessions. This is not the case for datasets — they need to be loaded over again by a new session, even if they are the same as the other session has already requested. It is caused by the fact that they are marked by session ID, which makes this dataset unique for this particular client.

2.4.1 GraphScope — and its good parts

Each of these systems has its own strengths, but often lacks one or more key qualities such as scalability, multitenancy, interactivity, or design specialization. To address this gap, we propose creating a system that encompasses all of these attributes. We decided to closely examine the distributed Graph Processing Platform, GraphScope [Fan+21], which was developed for production use at Alibaba. GraphScope Flex, an evolution of GraphScope, achieves up to 2,400× performance gain in real-world applications [He+23] over state-of-the-art cybersecurity monitoring systems.

Scalability provided by GraphScope is the hardest problem out of these four qualities we want to achieve. It also supports interactive execution out of the box. It understands the format of the graph well, so it does not have the extra overhead of using a plugin over a different system, such as in the case of GraphX on Spark.

2.4.2 GraphScope — the bad parts

One missing part is support for multitenancy. Extending this system by adding support for multiple concurrent users will result in a universal system for scalable multitenant graph processing and querying with strong data locality provided by the current system. When focusing on GraphScope, we already see a problem with static deployment, which allows only a single user session. While the processes and algorithms inside this single session are greatly optimized, the overhead of provisioning and running multiple copies of the same software is not an efficient use of computing resources.

Another issue arises from a design decision in Vineyard, which GraphScope uses as its working storage. While the colocation of compute and storage improves read times, Vineyard's authors designed the storage to be immutable. This allows for querying and transformations within the connected worker but creates a challenge when trying

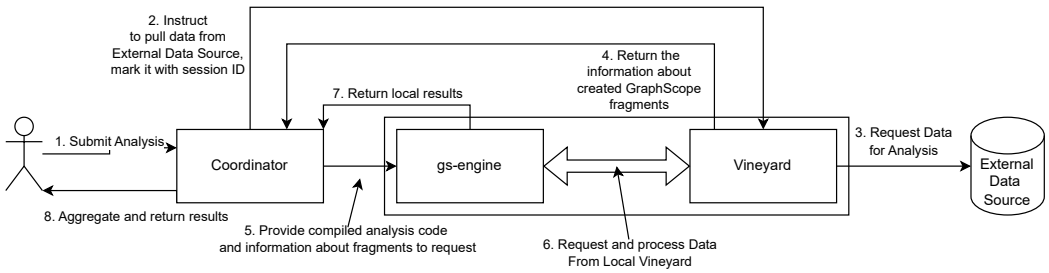

Figure 2.2: GraphScope data processing flow on the example of a single session.

to modify the original graph in place. Currently, GraphScope does not offer a way to perform such in-place modifications.

2.4.3 GraphScope and Multitenancy

GraphScope has the same problem as Spark: each cluster is deployed only to support a single user session connected to it. Similar to Spark, it creates a Coordinator (equivalent to Spark's Master) and Executors separate for each user, as well as imports the dataset directly to Vineyard, which allows multiple instances to potentially access the same data. Yet, there is currently no mechanism implementing this multi-access.

2.4.4 Vineyard

GraphScope uses Vineyard for fast-access storage. Vineyard by design allows the definition of new data types of storage and methods of access, optimizing the data access time. This is a step up from traditional stores of source data like HDFS [Apa23] and s3 [Ama23b], which only provide file-like access to data.

Internally, Vineyard uses distributed key-value store etcd [CNC23], known from Kubernetes, to keep track of its resources and their locations. Vineyard provides predefined types and methods of accessing them, as well as an interface to create a domain-specific implementation to read and write any format. This allows more flexibility in interacting with the data than the classical file-based system, relational databases or key-value stores, as it can take into account the data type and the way data is read.

However, GraphScope Executor needs to be on the same machine as where the data processing is taking place; because of its implementation, Vineyard only serves data over UNIX sockets [The23].

The granular approach to data access involves breaking down data into smaller,

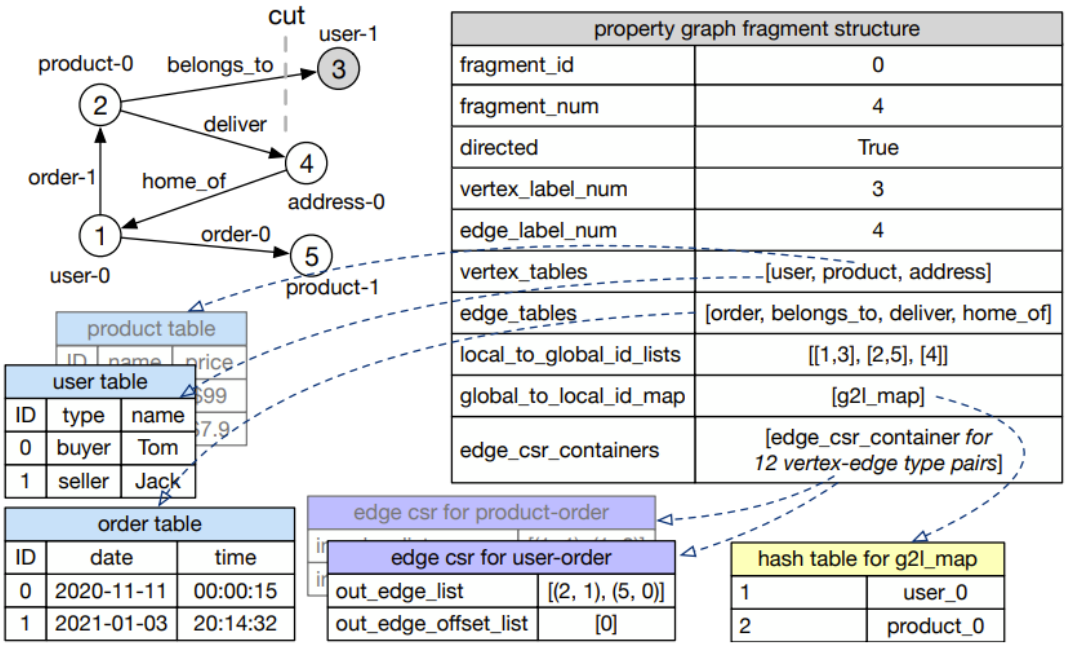

Figure 2.3: GraphScope Fragment, sourced from GraphScope Paper [Fan+21].

more manageable pieces that can be accessed and processed independently. This method is particularly useful when working with graphs or network data, where the relationships between nodes (such as connections, edges, or paths) are often complex and interdependent.

2.4.5 GraphScope Fragment

To address the challenges of graph data management, GraphScope introduced Fragment, visible in Figure 2.3, a special Vineyard data type designed to facilitate efficient access to specific portions of a graph. Graphs are usually stored in tabular formats on disk, such as in multiple CSV files. While this format is effective for storage, it poses significant challenges for random access and retrieval of graph partitions.

Fragment tackles this issue by parsing the graph data stored in these files and intelligently distributing its components across a Vineyard cluster. This distributed approach allows for more efficient data management and enables methods that provide quick access to any desired part of the graph.

2.4.6 GraphScope Flex

GraphScope Flex [He+23] was introduced to resolve the problem of having a generalized engine, which includes all the tools up front. Since the field is burgeoning, the selection of tools is growing with every passing day. That's why the innovation allows building a dedicated processing engine docker image suited for a particular workload. It is composed of multiple smaller blocks, which operate on different layers, in a fashion similar to the ISO OSI model of a network. This allows the creation of customized processing units that can replace standard executors for even more optimized workloads.

2.4.7 Integration with Kubernetes

GraphScope has two modes of operation with Kubernetes: standalone and Helm Chart-based. The first one needs access to the Kubernetes Kubeconfig configuration file, which it then uses to deploy the required resources via standard *kubectl* calls. When the new session is created in GraphScope, it first deploys the coordinator, which then provisions a Vineyard cluster and executor pods. Afterwards, it is the gateway to communicate from the client session to the rest of the deployed resources. Every client session has its own coordinator deployed.

The main difference in chart deployment is that GraphScope can be deployed first via Helm, and such a provisioned environment has the coordinator waiting for connections, cutting on waiting time for a client. The client only provides an address, and the session simply connects to an existing coordinator, instead of creating new resources. The problem is, that only a single client session can be connected at the same time, which only saves time on provisioning resources — the data cannot be shared, as it was not designed to do so.

To enable more efficient memory sharing, let's take a look at the development from a similar field: Inference Server for Machine Learning.

2.5 Inspiration from Machine Learning: The Inference Server

An inference server is a specialized system designed to host Machine Learning models and provide real-time predictions or batch processing capabilities. These servers are integral to deploying Machine Learning models in production environments, allowing for efficient management, scaling, and execution of models across different applications. The inference server handles various tasks, including loading models, processing incoming data, performing inference, and returning the results to the user [HP18].

The primary benefit of an inference server is its ability to offer low-latency predictions, making it suitable for applications requiring real-time responses. Additionally, it supports large-scale inference workloads, which is essential for applications that demand high throughput [Bai+20].

2.5.1 NVIDIA Triton

Triton Inference Server [NVI24], developed by NVIDIA, addresses the need for efficient and scalable deployment of Machine Learning models in production environments. It optimizes inference performance by enabling concurrent model execution, which allows multiple models to run simultaneously on the same GPU. Such an approach greatly reduces the time needed to load and execute the models, as they are managed intelligently by the internal scheduler and kept warm for inference requests.

2.5.2 Inference, but for Graphs?

As we see in the case of Triton, the need for fast-access frameworks has already arisen in the field of Machine Learning and is also present in the case of graph processing. Although said optimization qualities are present only in proprietary platforms [LL21; Tig], but not in an open-source, academic environment. GraphScope is quite fitting for creating such a scalable platform for graphs with a few adjustments.

3 Design and implementation

3.1 Researching existing state

We benchmark the current state of execution on GraphScope using WikiTalk [Les12] dataset on a subset of LDBC workloads to see how it behaves in practice. The results of the benchmark are visible in Figure 3.1. It currently has three distinct steps for every single user session analysis:

1. **Compilation** — majority of time spent executing when starting fresh, the new code submitted from the user needs to be compiled first,
2. **Data Loading** — part of analysis startup is in loading the dataset from the remote store, the problem is so severe that GraphScope developed its own data format [Li+23] for exporting graphs for reuse between sessions,
3. **Execution** — running the algorithm compiled previously on the dataset unpacked in memory.

Based on the experiments with a small dataset in Figure 3.1, we can see the parts in GraphScope, that can be easily shared between the users. Enabling sharing of dataset and in this specific case, algorithm compilation, produces the multitenant system that we want to achieve.

Based on the research questions and provided state of the art, we can ask for the following qualities of the final system, as it should:

1. Provide a graph processing platform for multiple concurrent users efficiently,
2. Have a good degree of interactivity with a good feedback loop, like a Machine Learning inference server,
3. Manage resources between multiple users,
4. Serve the cached in-memory source graph for multiple clients.

We will approach the solution in the following iterations:

1. First, we will take the original system from Figure 2.1, and combine all the deployments to use the same Vineyard cluster as in Figure 3.2.

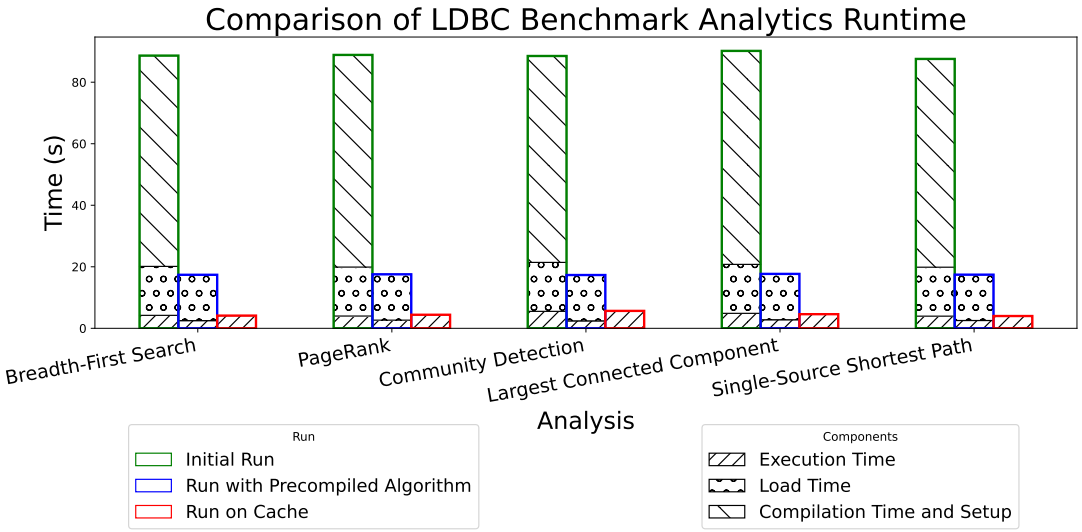

Figure 3.1: Difference between starting a new system (green) versus reusing the compiled stored procedure (blue) and dataset (red) in GraphScope [Fan+21]

2. To further improve the setup, we will allow the reuse of the same dataset, identified by external source name, across multiple clients (Figure 3.3), reducing the cold start for the analytics.

3. As the last step of improving the platform, we will add the multiple IDs on Coordinator, thus reducing the footprint of every consecutive client (Figure 3.4).

3.2 Combining the Vineyard

The design that we deal with initially visible in Figure 2.1 has a problem: all the components of the system need to be replicated for every extra user in the system.

The low-hanging fruit then is to deduplicate the storage: we can fit more than one dataset in the single Vineyard, and since GraphScope by default marks all data downloaded for the analysis with the session's unique UUID4, we can assume that these identifiers will not repeat on the same instance of Vineyard. We will then be able to get rid of repetition in this area, resulting in the new architecture visible in Figure 3.2, being the Intermediate Step 1.

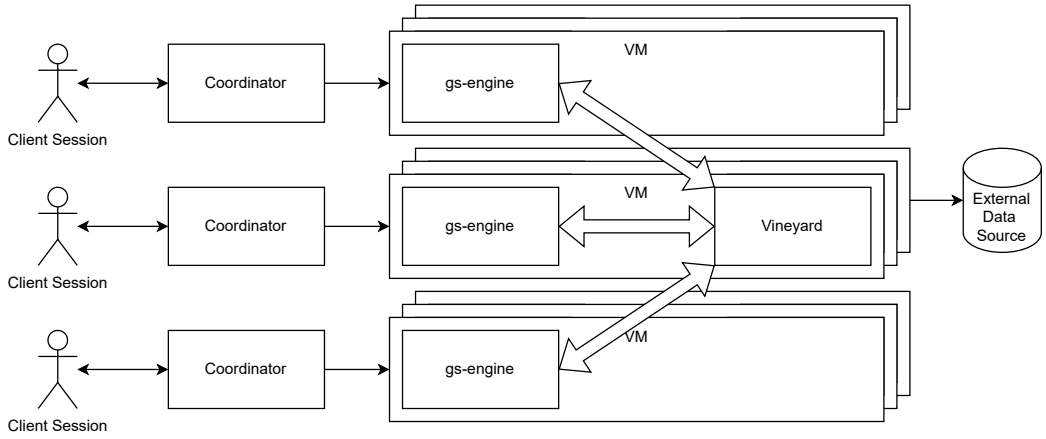

Figure 3.2: GraphScope architecture after combining the clients to use a single Vineyard instance.

Table 3.1: Memory footprint for different parts of the system and Neo4j with Spark as a reference.

Component	**Memory Footprint**	**Number Per Client**
Coordinator	120 MiB	1
Executors	2.5 MiB	NUM_NODES
Vineyard	40 MiB	NUM_NODES
Vineyard etcd	11 MiB	1
Neo4j	348 MiB	1
Spark Master	181 MiB	1
Spark Worker	126 MiB	NUM_NODES

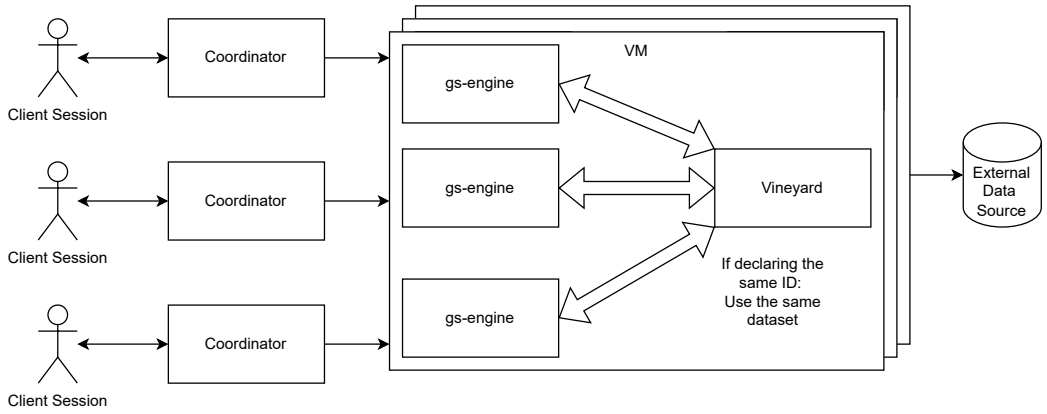

Figure 3.3: GraphScope modified to reuse the same dataset.

3.3 Sharing the loaded dataset

```
import graphscope
sess = graphscope.session(addr=f'{ip_address}:{port}')
```

Listing 3.1: GraphScope session connection configuration.

With the storage now combined into a single space, the same dataset might be repeated within the storage. This is because each session will request its own copy of the dataset, leading to additional cold start times and extra network traffic when starting analysis referring to the same data source. The solution to this challenge is simplified by the Vineyard's limitation of not supporting writes after a dataset is loaded [The23]. Therefore, we don't need to worry about multiple access, as there is no way to modify existing data.

To implement this functionality, we need to provide a repeatable session. The session ID is autogenerated whenever the GraphScope client connects to the coordinator, and it blocks further connections until it is removed. In Vineyard, resources marked with session ID *X* will exist until the session disconnects: this part of GraphScope also needs to be prevented from executing so as not to interrupt the other clients using the same dataset. So in general, the steps needed to implement a reusable dataset are as follows:

1. Allow providing session ID in connection configuration.
2. Prevent resources from being deleted on disconnect.
3. Discover the GraphScope Fragments2.3 already allocated on Vineyard.

To develop this functionality, we needed to rework the connection string. This included rewriting the logic for the session ID to be only created when it was not supplied. As of current implementation, each session ID is stored in a coordinator specific for the user that created it.

After following these instructions, we end up with the situation presented in Figure 3.3, which represents Intermediate Step 2.

```
import graphscope
sess = graphscope.session(addr=f'{ip_address}:{port}', session_id="xxx")
```

Listing 3.2: GraphScope session connection configuration after reaching Intermediate Step 2.

3.4 Adding multiple clients on a single coordinator

We were able to deduplicate the Vineyard and data storage, now the last part to optimize is the coordinator. As of now, it is limited by what is provided in the connection command, and once the connection succeeds, no more clients are allowed to connect. We can unlock the multiple users by allowing multiple sessions with multiple (not necessarily unique!) session IDs to connect, allowing the same behavior as in Neo4j data platform [LL21], but open-source. Therefore, to differentiate between the sessions and avoid conflicts, the steps are as follows:

1. Refactor the code in coordinator to accept a list of session objects instead of a singular one there was so far.

2. Add a hidden session ID to identify requests from multiple clients uniquely, since we removed the uniqueness of the session ID.

After implementing these steps, the whole setup simplifies to the final architecture (Fig. 3.4), which is the most optimal given the provided stack. This was not implemented in this project, as the timescale was too short, so further suggestions are presented in Future Work section.

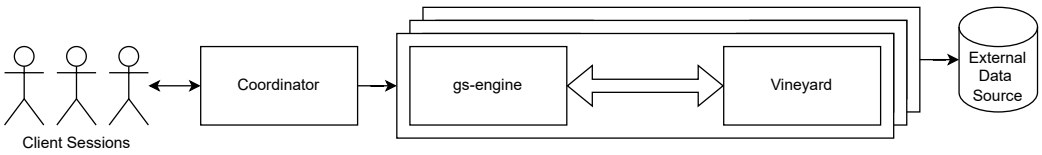

Figure 3.4: GraphScope optimized for multitenant use.

4 Evaluation and comparison with state of the practice

Table 4.1: System Configuration for benchmarking.

Component	**Configuration details**
Virtualization	QEMU 1:4.2-3ubuntu6.29
VMs	9 spread across 3 physical nodes, 1 for Kubernetes master and 8 for worker nodes
vCPU	4 cores of Intel(R) Xeon(R) Silver 4210 per VM
DRAM	16 GiB DDR4 per VM
OS	Ubuntu 20.04.6 LTS
Platform	Kubernetes v1.27.16
Container Engine	containerd 1.7.19
Data Storage	Hadoop 3.3.3
PVC backend	NFS 1:1.3.4-2.5ubuntu3.7
Neo4j	5.22
GraphScope	0.28.0

4.1 Hardware Setup

To evaluate our design, we deploy a Kubernetes cluster utilizing the Continuum framework [Jan+23], enabling rapid iteration and efficient infrastructure setup. The cluster consists of eight Ubuntu-based virtual machines (VMs) for worker nodes and one identical for a single master node, each provisioned with 4 virtual CPU cores and 16 Gigabytes of RAM. These VMs were distributed across three distinct physical nodes to ensure a robust and diversified testing environment.

The hardware for this setup is hosted by AtLarge Research [AtL24], where this thesis research is conducted. A detailed schematic of the hardware and network configuration is shown in Figure 4.1, illustrating the system architecture and the distribution of computational resources across the physical nodes.

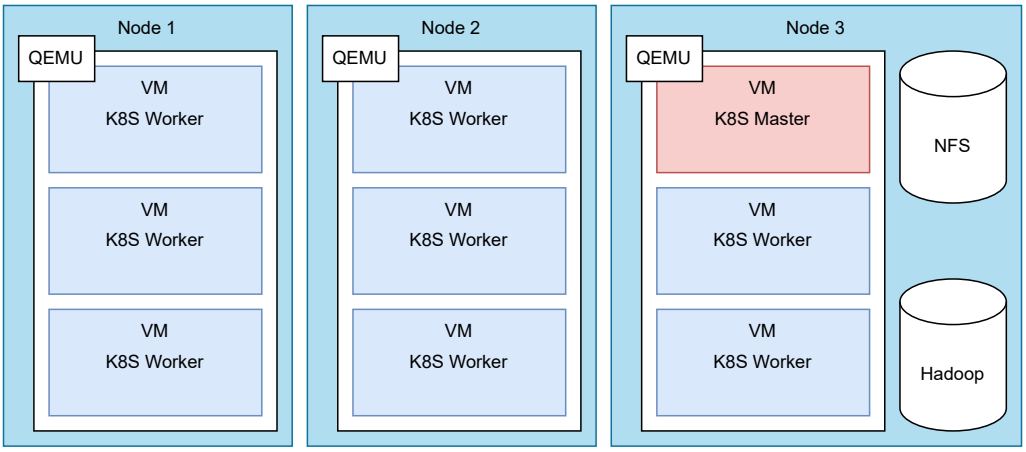

Figure 4.1: Hardware layout of the system used for benchmarking.

4.1.1 Software Setup

For testing the efficiency of resource consumption, we use Kubernetes [Clo] Cluster running on QEMU [QEM24], made reproducible by Continuum [Jan+23]. To facilitate extra storage needs of Neo4j, we create Persistent Volumes in NFS [Sun89], provided via a Kubernetes operator [Kub24c], as well as a separate single node Hadoop [Apa23] deployed using Docker Compose [Doc24a; Ran24] running on a bare metal host as the source storage for GraphScope and Spark.

To gather accurate information about resource consumption on the Kubernetes cluster, we leverage data provided by the Kubernetes Metrics Server [Kub24b]. This data is then aggregated using custom-written code to reflect the actual usage of RAM and CPU for each part of the system. This monitoring setup is crucial for understanding how each component contributes to overall resource usage, enabling us to identify bottlenecks and compare the real usage of resources.

Since we pull massive amounts of data over the network with deployed Docker [Doc24d] images, we use Docker Registry [Doc24b] with pull-through cache [Doc24c] to optimize our deployment time.

One of the key goals is to create a reproducible benchmarking setup. To deploy each part of the system, we use Helm Charts [The24a; Neo24b; Bit24; Gra24; Kub24a] with our own set of values for each of the releases.

For evaluating the setup we use the small graph network WikiTalk [Les12], composed of 2.4 million nodes and 5 million edges, which contains the data for all Wikipedia discussions until 2008.

To test the scalability of the system, we increase the number of replicas of Spark workers and Neo4j deployment, although the latter can be tested only for memory consumption as we are missing the license for clustering. We also deploy a new GraphScope deployment for each client session, reusing the same Vineyard storage in most cases.

4.1.2 Versions

On the cluster outfitted with the auxiliary software as described above, we then run different types of graph processing platforms:

1. Spark with GraphX plugin
2. Neo4j
3. Standard GraphScope (Figure 2.1) — with separate Vineyard deployment.
4. Intermediate Step 1 GraphScope (Figure 3.2) — with shared Vineyard deployment.
5. Intermediate Step 2 GraphScope (Figure 3.3) — modified with shared Vineyard session resource identifiers.
6. Final GraphScope (Figure 3.4) — with single coordinator deployment and all qualities described above.

4.2 Evaluation results

4.2.1 Scaling and base memory usage

The first test we perform measures the impact of adding an extra client to the setup. Since we have 8 nodes, GraphScope was configured to deploy Vineyard and executors on all 8 of them with every consecutive deployment. Spark was upscaled with an extra worker for every scale above 2, and Neo4j was deployed anew, to show the standard clustering recommended in the documentation [Neo24c].

We performed the preliminary upscaling test to assess the memory gains: we can see in Figure 4.2 that we save a lot of space just by combining the Vineyards, and even more by merging the Coordinators with the final design. The default setup scales worse than Spark and about the same as Neo4j. The difference is, that Neo4j is a single container containing everything from database to management, and Spark is a heavy container, weighing much more than the lightweight Executor container from GraphScope from Table 3.1.

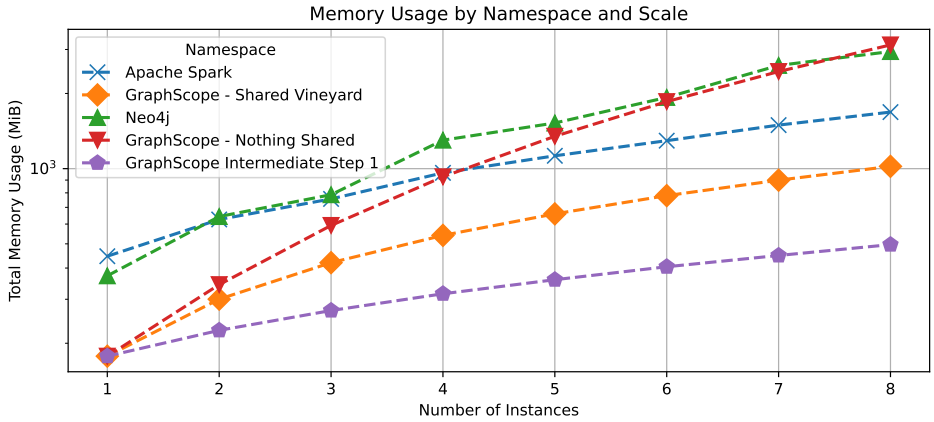

Figure 4.2: Memory scaling before and after tuning GraphScope.

We can see that even the default GraphScope with combined Vineyard can scale better than the other platforms, mainly because it's split, so it can save on memory. Instead of deploying 8 extra Vineyard instances in a new cluster. This can consume up to $8 \cdot 40 = 320MiB$ of extra memory space. Instead, it reuses the first deployed Vineyard and does not therefore occupy extra space.

Another interesting matter is theoretical Final GraphScope, which could cut off as much as 120 MiB per user working on the cluster.

Spark scales better than Neo4j, since we scale up only the worker, which consumes much less space.

4.2.2 CPU efficiency

To compare the efficiency of neo4j and GraphScope, we have employed Single-Source Shortest Path (SSSP) algorithm as our reference. Since we are not able to scale Neo4j without commercial license, we have to run it on a single instance. The configuration used was 4 Gigabytes of RAM and 2 CPUs per instance.

We see on the Figure 4.3 that given the scaling and extra resources, GraphScope will win over Neo4j for the same example query: the difference will get bigger when we add extra nodes for GraphScope, which makes the GraphScope a faster solution when we can scale up. The up-and-coming Neo4j Fabric[Neo23a] Neo4j might also scale better, but we were still able to achieve a good result, given that GraphScope will scale even better for bigger graphs.

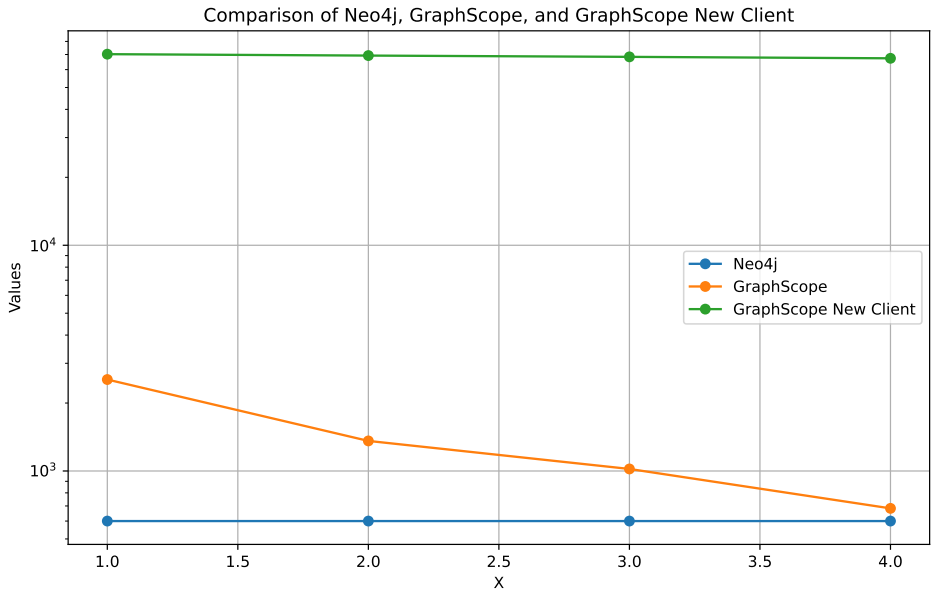

Figure 4.3: CPU scaling between tuned Graphscope and Neo4j

5 Conclusion and future work

In this work, we show that optimizing the new academic platforms can bear fruit: better scalability and less resource consumption on standby with increased multitenancy are quite easy to achieve if ideas from similar fields are applied. We were able to produce a querying server that rivals the established graph database Neo4j with pure processing performance and graph processing framework GraphX in scalability. We have also shown, that the solution achieves better memory scalability than the other existing products, achieving up to 83% memory reduction compared to state of the art. Unfortunately, due to the short timeframe of the project, we were not able to finish every planned improvement.

The main contribution of this work is reducing the analysis time when reusing the same GraphScope instance. We did it by removing the cold start associated with algorithm compilation and data import — we keep once imported graph in memory between sessions, allowing for faster data access. The standard loading time of a graph can take hours according to the official documentation of GraphScope, and we were able to remove the loading time by redirecting newly allocated sessions to resources already provisioned by the previous analyses. Multiple users can now access the same graph instance loaded in memory, saving the space allocated inside Vineyard. We were able to reduce the time of similar analysis by over 28 times while compacting the memory space required by the number of concurrent clients connected to the same data store.

We can now answer all the questions stated at the beginning of the work.

1. **Research Question 1: How to design a multitenant graph processing platform based on the GraphScope?** After conducting a literature review, we have found out we can achieve multitenancy in a way similar to how Spark does it: by faking our ID, so the system thinks we are the previous user. This work added an option to take over existing session ID, allowing new GraphScope session to talk with cached data in the store. This also required preventing Vineyard from deleting the data on session close, since Vineyard would unload the graph when this particular session would disconnect.

2. **Research Question 2: How to enable interactivity in a multitenant graph processing platform?** We have kept interactivity by extending only the parts

below the GraphScope client layer, which kept the same system intact. The changes introduced did not change the default behavior of GraphScope, which has a very fast feedback loop for the user. Because we have a rapid way to access every part of the data thanks to Vineyard and GraphScope Fragment on it, we get very similar performance to Neo4j, while benefitting from scalability.

3. **Research Question 3: How to efficiently share a single graph processing platform between users?** After conducting the literature review, we have decided to keep the executor units the same as in Spark - this way, the coordinator, which does not do much heavy work, can work the different sets of Executors for each user, making them not interfere with each other.

4. **Research Question 4: How to share graph data between users, and what are the performance benefits?** When we shared the graph data with the reuse of session ID as described in answer to the first point, we were able to achieve almost no loading time for repeating data sources — the data was already located inside Vineyard, and it was only the matter of changing the session ID so it would match the one that imported the data originally. Since the data is already there, we were able to reduce cold starts related to loading it, as well as enable smaller analyses on big sets of data to be performed very fast, in an interactive way thanks to cache (similar to inference servers in Machine Learning).

5.1 Future work

There are multiple ways to extend the achieved setup:

1. Remove requirement for the coordinator, as suggested by the unimplemented final design, saving up over 120 MiB of memory for every consecutive user, as well as allowing for reuse of compiled algorithms,

2. Add data-source awareness inside GraphScope, which can reuse the source if it was already loaded by another user,

3. Create more monitoring and visibility of internal caching behavior in GraphScope and Vineyard, to be able to better use cache,

4. Adjust Vineyard to garbage collect leftover graphs, as they are not removed after the changes that were introduced and may cause out-of-memory errors.

List of Figures

List of Tables

Bibliography

[Ahm+20] S. Ahmadian, N. Joorabloo, M. Jalili, M. Meghdadi, M. Afsharchi, and Y. Ren. "A temporal clustering approach for social recommender systems." In: *Proceedings of the 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. ASONAM '18. Barcelona, Spain: IEEE Press, 2020, pp. 1139–1144. ISBN: 9781538660515.

[Ama23a] Amazon Web Services. *Amazon Neptune*. Graph database service. Accessed: 2024-06-26. 2023. URL: https://aws.amazon.com/neptune/.

[Ama23b] Amazon Web Services. *Amazon Simple Storage Service (S3)*. Accessed: 2024-06-26. 2023. URL: https://aws.amazon.com/s3/.

[Ama24] Amazon Web Services. *Amazon SageMaker Documentation*. Accessed: 2024-07-14. 2024. URL: https://docs.aws.amazon.com/sagemaker/.

[Apaa] Apache Software Foundation. *Apache Mesos*. https://mesos.apache.org/. Accessed: 2024-06-26.

[Apab] Apache Software Foundation. *Apache Yarn*. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/index.html. Accessed: 2024-06-26.

[Apa23] Apache Software Foundation. *Apache Hadoop*. Software available from Apache Software Foundation. Accessed: 2024-06-26. 2023. URL: https://hadoop.apache.org/.

[AtL24] AtLarge Research. *About Us*. https://atlarge-research.com/about.html. Accessed: 2024-08-10. 2024.

[Bai+20] J. Bai, F. Lu, K. Zhang, J. Wang, and Q. Hou. "Fast and Scalable Inference with TensorRT." In: *Proceedings of the International Conference on Machine Learning*. ICML. 2020, pp. 493–502.

[Bit24] Bitnami Helm Charts Maintainers. *Helm Install for Spark*. https://registry-1.docker.io/bitnamicharts/spark. Accessed: 2024-08-12. 2024.

[Bur+20] C. Buragohain, K. M. Risvik, P. Brett, M. Castro, W. Cho, J. Cowhig, N. Gloy, K. Kalyanaraman, R. Khanna, J. Pao, M. Renzelmann, A. Shamis, T. Tan, and S. Zheng. "A1: A Distributed In-Memory Graph Database." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, May 2020. DOI: 10.1145/3318464.3386135. URL: http://dx.doi.org/10.1145/3318464.3386135.

[Clo] Cloud Native Computing Foundation. *Kubernetes*. https://kubernetes.io. Accessed: 2024-06-26.

[CNC23] CNCF - Cloud Native Computing Foundation. *etcd - A distributed, reliable key-value store for the most critical data of a distributed system*. Accessed: 2024-06-26. 2023. URL: https://etcd.io/.

[DG08] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters." In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[Doc24a] Docker Inc. *Docker Compose: Define and run multi-container applications with Docker*. https://docs.docker.com/compose/. Accessed: 2024-08-12. 2024.

[Doc24b] Docker Inc. *Docker Registry*. https://docs.docker.com/registry/ (Accessed: 2024-08-12). 2024.

[Doc24c] Docker Inc. *Docker Registry Pull Through Cache*. https://docs.docker.com/registry/recipes/mirror/ (Accessed: 2024-08-12). 2024.

[Doc24d] Docker, Inc. *Docker: A platform for developers and sysadmins to build, run, and share applications with containers*. https://www.docker.com. Accessed: 2024-08-12. 2024.

[Dom23] Domo. *Data Never Sleeps 11.0*. https://www.domo.com/learn/infographic/data-never-sleeps-11. Accessed: 2024-07-09. 2023.

[Fan+21] W. Fan, T. He, L. Lai, X. Li, Y. Li, Z. Li, Z. Qian, C. Tian, L. Wang, J. Xu, et al. "GraphScope: a unified engine for big graph processing." In: *Proceedings of the VLDB Endowment* 14.12 (2021), pp. 2879–2892.

[Fou24] A. S. Foundation. *Apache Giraph*. https://giraph.apache.org/. Accessed: 2024-06-26. 2024.

[Gon+14] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. "GraphX: graph processing in a distributed dataflow framework." In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 599–613. ISBN: 9781931971164.

[Gra24] GraphScope Helm Chart Maintainers. *Helm Upgrade for GraphScope*. `https://github.com/alibaba/GraphScope`. Accessed: 2024-08-12. 2024.

[He+23] T. He, S. Hu, L. Lai, D. Li, N. Li, X. Li, L. Liu, X. Luo, B. Lyu, K. Meng, S. Shen, L. Su, L. Wang, J. Xu, W. Yu, W. Zeng, L. Zhang, S. Zhang, J. Zhou, X. Zhou, and D. Zhu. *GraphScope Flex: LEGO-like Graph Computing Stack*. 2023. arXiv: `2312.12107 [cs.DC]`.

[HL11] M. Hilbert and P. López. "The World's Technological Capacity to Store, Communicate, and Compute Information." In: *Science* 332.6025 (2011), pp. 60–65. DOI: `10.1126/science.1200970`.

[HP18] J. Huang and J. Peng. "GPU-accelerated Inference for Deep Learning." In: *Proceedings of the IEEE* 106.6 (2018), pp. 1031–1045.

[Inc24] F. Inc. *AllegroGraph 8.0 - Neuro-Symbolic AI Platform*. `https://allegrograph.com`. Accessed: 2024-06-26. 2024.

[Jan+23] M. Jansen, L. Wagner, A. Trivedi, and A. Iosup. "Continuum: Automate Infrastructure Deployment and Benchmarking in the Compute Continuum." In: *Proceedings of the First FastContinuum Workshop, in conjuncrtion with ICPE, Coimbra, Portugal, April, 2023*. 2023. URL: `https://atlarge-research.com/pdfs/2023-fastcontinuum-continuum.pdf`.

[KMK14] M. Kusumoto, T. Maehara, and K.-i. Kawarabayashi. "Scalable similarity search for SimRank." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 325–336. ISBN: 9781450323765. DOI: `10.1145/2588555.2610526`. URL: `https://doi.org/10.1145/2588555.2610526`.

[Kub24a] Kubernetes Metrics Server Maintainers. *Helm Upgrade for Metrics Server*. `https://kubernetes-sigs.github.io/metrics-server`. Accessed: 2024-08-12. 2024.

[Kub24b] Kubernetes Metrics Server Maintainers. *Kubernetes Metrics Server*. `https://github.com/kubernetes-sigs/metrics-server`. Accessed: 2024-08-12. 2024.

[Kub24c] Kubernetes NFS Subdir External Provisioner Maintainers. *Kubernetes NFS Subdir External Provisioner*. `https://github.com/kubernetes-sigs/nfs-subdir-external-provisioner`. Accessed: 2024-08-12. 2024.

[Kwo+23] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. "Efficient Memory Management for Large Language Model Serving with PagedAttention." In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. Koblenz, Germany: Association for Computing Machinery, 2023, pp. 611–626. ISBN: 9798400702297. DOI: 10.1145/3600006.3613165. URL: https://doi.org/10.1145/3600006.3613165.

[Lab23] D. Labs. *Dgraph: An Open Source Distributed Graph Database*. https://dgraph.io. 2023.

[Les12] J. Leskovec. *Wiki-Talk: Wikipedia Talk Network Dataset – KONECT, July 2012*. https://snap.stanford.edu/data/wiki-Talk.html. Accessed: 2024-08-15. 2012.

[Li+19] Y. Li, H. Kuwahara, P. Yang, L. Song, and X. Gao. "PGCN: Disease gene prioritization by disease and gene embedding through graph convolutional neural networks." In: *bioRxiv* (2019). DOI: 10.1101/532226. URL: https://www.biorxiv.org/content/early/2019/01/28/532226.

[Li+23] X. Li, W. Zeng, Z. Wang, D. Zhu, J. Xu, W. Yu, and J. Zhou. "Enhancing Data Lakes with GraphAr: Efficient Graph Data Management with a Specialized Storage Scheme." In: (2023). DOI: 10.48550/ARXIV.2312.09577. arXiv: 2312.09577. URL: https://doi.org/10.48550/arXiv.2312.09577.

[LL21] L. Lazarevic and W. Lyon. "Overview of the Neo4j Graph Data Platform." In: (2021). Neo4j Developer Blog. URL: https://neo4j.com/developer-blog/overview-of-the-neo4j-graph-data-platform/.

[Mal+10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. "Pregel: a system for large-scale graph processing." In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 135–146. ISBN: 9781450300322. DOI: 10.1145/1807167.1807184. URL: https://doi.org/10.1145/1807167.1807184.

[Mic23] Microsoft Corporation. *Azure Cosmos DB*. Cloud database service. Accessed: 2024-06-26. 2023. URL: https://azure.microsoft.com/en-us/services/cosmos-db/.

[MSB23] J. Monteiro, F. Sá, and J. Bernardino. "Experimental Evaluation of Graph Databases: JanusGraph, Nebula Graph, Neo4j, and TigerGraph." In: *Applied Sciences* 13 (May 2023), p. 5770. DOI: 10.3390/app13095770.

[Neo20] I. Neo4j. *Sharding the LDBC Social Network - Graph Database & Analytics*. Accessed: 2024-07-15. 2020. URL: https://neo4j.com/fosdem20/.

[Neo23a] Neo4j, Inc. *Getting Started with Neo4j Fabric*. https://neo4j.com/blog/getting-started-with-neo4j-fabric/. Accessed: 2024-08-15. 2023.

[Neo23b] Neo4j, Inc. *Neo4j*. Graph database management system. Accessed: 2024-06-26. 2023. URL: https://neo4j.com/.

[Neo24a] Neo4j. *Understanding Data on Disk*. Accessed: 2024-07-14. 2024. URL: https://neo4j.com/developer/kb/understanding-data-on-disk/.

[Neo24b] Neo4j Helm Chart Maintainers. *Helm Repository for Neo4j*. https://helm.neo4j.com/neo4j. Accessed: 2024-08-12. 2024.

[Neo24c] Neo4j Inc. *Quickstart: Neo4j Cluster Server Setup on Kubernetes*. Accessed: 2024-08-15. 2024. URL: https://neo4j.com/docs/operations-manual/current/kubernetes/quickstart-cluster/server-setup/.

[Neo24d] Neo4j, Inc. *Cypher Query Language*. https://neo4j.com/developer/cypher/. https://neo4j.com/developer/cypher/. 2024. URL: https://neo4j.com/developer/cypher/.

[NVI24] NVIDIA Corporation. *Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution*. Available at https://github.com/triton-inference-server/server. 2024. URL: https://github.com/triton-inference-server.

[Ori24] OrientDB Ltd. *OrientDB: A NoSQL, Open Source Multi-Model Database Management System*. https://www.orientdb.org/. Accessed: 2024-05-20. 2024.

[QEM24] QEMU Developers. *QEMU: A generic and open source machine emulator and virtualizer*. Accessed: 2024-08-12. 2024. URL: https://www.qemu.org.

[Ran24] C. Rancavil. *Hadoop Single Node Cluster*. https://github.com/rancavil/hadoop-single-node-cluster. Accessed: 2024-08-12. 2024.

[Sak+21] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. Aref, M. Arenas, M. Besta, P. A. Boncz, K. Daudjee, E. D. Valle, S. Dumbrava, O. Hartig, B. Haslhofer, T. Hegeman, J. Hidders, K. Hose, A. Iamnitchi, V. Kalavri, H. Kapp, W. Martens, M. T. Özsu, E. Peukert, S. Plantikow, M. Ragab, M. R. Ripeanu, S. Salihoglu, C. Schulz, P. Selmer, J. F. Sequeda, J. Shinavier, G. Szárnyas, R. Tommasini, A. Tumeo, A. Uta, A. L. Varbanescu, H.-Y. Wu, N. Yakovets, D. Yan, and E. Yoneki. "The future is big graphs: a community view on graph processing systems." In: *Commun. ACM* 64.9 (Aug. 2021), pp. 62–71. ISSN: 0001-0782. DOI: 10.1145/3434642. URL: https://doi.org/10.1145/3434642.

[Shi+16] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. "Fast and Concurrent RDF Queries with RDMA-based Distributed Graph Exploration." In: *12th USENIX Symposium on Operating Systems Design and Implementation*. OSDI '16. GA: USENIX Association, Nov. 2016, pp. 317–332. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/shi.

[Sta23] Statista. *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025*. https://www.statista.com/statistics/871513/worldwide-data-created/. Accessed: 2024-07-09. 2023.

[Sun89] Sun Microsystems, Inc. *NFS: Network File System Protocol Specification*. RFC 1094. 1989. URL: https://tools.ietf.org/html/rfc1094.

[The23] The Vineyard Authors. *NO instant remote data accessing*. https://v6d.io/notes/architecture.html (Accessed: 2024-07-14). 2023.

[The24a] The Helm Authors. *Helm: The Kubernetes Package Manager*. Accessed: 2024-08-12. 2024. URL: https://helm.sh.

[The24b] The JanusGraph Authors. *JanusGraph: an open-source, distributed graph database*. https://janusgraph.org. Version 0.6.3, Apache Software License 2.0. 2024.

[The24c] The PostgreSQL Global Development Group. *PostgreSQL 16 Documentation*. Accessed: 2024-08-15. 2024. URL: https://www.postgresql.org/docs/.

[Tig] I. TigerGraph. *TigerGraph: Native Parallel Graph Database for Enterprise Applications*. https://www.tigergraph.com/. Accessed: 2024-05-20.

[Wen+22] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding. "MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters." In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 945–960. ISBN: 978-1-939133-27-4. URL: https://www.usenix.org/conference/nsdi22/presentation/weng.

[YJG03] A. B. Yoo, M. A. Jette, and M. Grondona. "SLURM: Simple Linux Utility for Resource Management." In: *Job Scheduling Strategies for Parallel Processing*. Ed. by D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4.

[Yu+23] W. Yu, T. He, L. Wang, K. Meng, Y. Cao, D. Zhu, S. Li, and J. Zhou. "Vineyard: Optimizing Data Sharing in Data-Intensive Analytics." In: *Proc. ACM Manag. Data* 1.2 (June 2023). DOI: 10.1145/3589780. URL: https://doi.org/10.1145/3589780.

[Zah+16] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. "Apache Spark: a unified engine for big data processing." In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: https://doi.org/10.1145/2934664.