GUARANTEEING SERVICE LEVEL AGREEMENTS FOR TRIANGLE COUNTING VIA OBSERVATION-BASED ADMISSION CONTROL ALGORITHM

Weerakkody Arachchige Chinthaka Rukshan Weerakkody

(189356P)

Thesis submitted in partial fulfillment of the requirements for the degree
Master of Science in Computer Science

Department of Computer Science and Engineering
Faculty of Engineering

University of Moratuwa
Sri Lanka

October 2022

DECLARATION

I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: ……. Date:…….
Name: Chinthaka Weerakkody

The above candidate has carried out research for the Masters dissertation under my supervision.

Signature of the supervisor: ……. Date: ……..
Name: Prof. Sanath Jayasena

Signature of the supervisor: ……. Date: ……..
Name: Dr. Miyuru Dayarathna

ACKNOWLEDGEMENT

My sincere appreciation goes to my family for the continuous support and motivation given to manage my MSc research work. I also express my heartfelt gratitude to Prof. Sanath Jayasena and Dr. Miyuru Dayarathna, my supervisors, for the supervision and advice given throughout this research period. I'm eternally grateful for Prof. Toyotaro Suzumura for helping us by providing necessary remote servers for the project. I am also thankful to WSO2. Telco (Pvt) Ltd, Virtusa (Pvt) Ltd, Macrometa and WSO2 Lanka (Pvt) Ltd for providing support to complete this project.

ABSTRACT

Increasingly large graph processing applications adopt the approach of partitioning and then distributed processing. However, maintaining guaranteed Service Level Agreement (SLA) on distributed graph processing for concurrent query execution is challenging because graph processing by nature is an unbalanced problem. We investigate on maintaining predefined service level agreements for commonly found graph processing workload mixtures. We develop a Graph Query Scheduler Mechanism (GQSM) which maintains a guaranteed service level agreement in terms of overall latency.

The proposed GQSM model is implemented using the queueing theory. Main component of GQSM is a job scheduler which is responsible for listening to an incoming job queue and scheduling the jobs received. The proposed model has a calibration phase where the Service Level Agreement data, load average curve data, and maximum load average which can be handled by the hosts participating in the cluster without violating SLA is captured for the graphs in the system. After completing the calibration phase the job scheduler is capable of predicting the load average curve for the incoming job requests. The scheduler checks whether the maximum load average extracted from the predicted load average curve exceeds the load average threshold values captured in the calibration phase. Based on the result the job scheduler accepts or rejects the job requests received.

Results show that SLA is successfully maintained when the total number of users is less than 6 in a JasmineGraph cluster deployed in a single host. For distributed clusters the number of users can go up to 10 without violating SLA. The proposed model is scalable and it can be applied to a distributed environment as well.

As future work, the proposed model can be extended to work with less initial calibration steps and the scheduling algorithm can be improved with intelligent workload management among hosts for more efficient resource consumption.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description
SLA	Service Level Agreement
CPU	Central Processing Unit
VM	Virtual Machine
HM	Host Machine
GPU	Graphics Processing Unit
GQSM	Graph Query Scheduling Mechanism
ESM	Elastic Switching Mechanism

LIST OF APPENDICES

CHAPTER 1: INTRODUCTION

This chapter gives an introduction to this research project. Then this chapter states the research problem, motivation behind addressing this research problem. Finally, it discusses the aims and objectives of this research.

1.1 Introduction

Recent computing applications prominently use graph databases for data representation. Some notable examples for these applications are online social networks, semantic web (DBPedia), computational biology, transportation systems and cyber security. Multiple graph/RDF (Resource Description Framework) database systems have been developed by both academia and industry in the recent for managing and mining large graph datasets. Some of the notable examples include Acacia [2][3][4], Titan [5], Oracle PGX [6], Neo4j [7], Microsoft Trinity [8], GraphChiDB [9] and AllegroGraph [10].

Graphs have interesting characteristics such as edge distributions following power-law and highly connected components. which makes it very challenging to handle when they are large with billions of vertices. The distributed graph databases need to handle these complexities to provide scalable execution given the limited amount of hardware resources on which they typically run.

Providing graph databases as services using cloud platforms is prominent today because of the resource limitations of the on-premise data centers. A notable example is Neo4j graph database cloud offering. With the sufficient hardware resources in place, it is important to maintain the service level agreements for job execution time in cloud-based graph database servers.

Graph databases are being used in mission critical applications such as cyber security and transportation systems. In such applications it is important to respond back to the queries within a shorter time period. For that it is necessary for the graph database to have an understanding of the underlying hardware resources available for the server to execute the requests. Based on the underlying hardware resource consumption the graph database should be capable of deciding whether the job requests can be served without delaying. If the graph database executes queries with low resources, then it takes a lot of time to execute the queries. In those situations, the external users who initiate the job requests will have to wait longer for the results. This leads to the requirement of having an intelligent job scheduler. A notable example of having such a job scheduler is Elastic Switching Mechanism (ESM) [12].

1.2 Research Problem

Current graph database servers do not consider the maintenance of service level agreement (SLA) for graph query execution time. Having SLA enforcement for graph database servers is important for them to be used as online transaction processing systems (OLTP).

1.3 Motivation

Social networks, electronic commerce systems, and smart cities produce huge amounts of data due to their massive scale. It is very important to analyze those data to identify trends in those areas. Graph databases can be used to easily represent the objects and their relationships in these data producers. Graph databases also make it easy to analyze those data as there are vast amounts of graph algorithms (PageRank, Connected Components, Triangle Counting.) exists to measure the properties of such graphs. Graph databases have multiple applications in cyber security [28][29], epidemiology [32], finance, metabolism [30][31] and scientific collaboration [33].

A distributed graph database will be serving multiple requests at a given time. Underlying resources of the server hosts are consumed when serving the requests it receives. If we consider a distributed graph database which accepts all the requests it receives, then there can be a huge number of requests to be served at times. At this time the system load goes up because of the resource consumption of the underlying hardware platform. Therefore, the performance of the graph database decreases. Due to that the response time for the job requests the graph database has received increases. As data analytics this is a pain point for them because they have to wait longer to get the results. To mitigate this problem in graph databases there should be a component which can take the decisions on accepting and rejecting job requests depending on the resource availability.

Most of the research carried out in graph databases is mainly focused on optimizing algorithms such as PageRank, Connected Components and Triangle Counting. There is less focus on an intelligent job scheduler which can serve requests efficiently within pre-agreed SLA bounds on query execution time using the underlying hardware resources. There are plenty of applications of graph processing where such enforcement of SLA bounds is critically important. For example, the graph processing in cyber security [28][29] should enforce a SLA because the output of the processing is being used to make decisions in the security concerns. Therefore, maintaining SLA for query execution time is important because it is the main contributor when calculating the latency.

Selecting a graph database for the implementations is a major challenge. Even Though many graph databases exist, all of them don't provide the necessary support for extending the existing system capabilities. JasmineGraph [20] is an open source distributed graph database system. This graph database server is mainly developed for research purposes and allows extending existing features as well as for introducing new features. Therefore, we have chosen JasmineGraph as the graph database server for the implementation of this project.

There are many popular algorithms which are being used in graph databases. One such algorithm is triangle counting. It is a widely used algorithm in many situations such as spam detection, discovery of hidden thematic structures in the World Wide Web and computer-aided design applications. Above examples depict that it is being used in time bound applications as well. It is important to enforce SLA for such time bound applications. Therefore, we have chosen triangle counting algorithm to be used in this project. Our main objective is to develop a job scheduler which guarantees service level agreements for triangle counting using observation-based admission control algorithm.

1.4 Aim and Objectives

Aim of this research is to develop a job scheduling algorithm which guarantees SLA for triangle counting jobs using observation based admission control. Main objectives of the project are,

1. Implement the basic system architecture of JasminGraph following Acacia's system architecture.
2. Implement graph triangle counting
3. Develop an approach to derive the service level agreement for triangle count jobs for a given graph via calibration
4. Develop a job scheduler algorithm which will guarantee service level agreements for triangle count jobs running in a cloud based distributed graph database system using observation based admission control.
5. Conduct performance evaluations in distributed compute clusters.

CHAPTER 2: LITERATURE REVIEW

This chapter provides a literature review which consists of several subsections which covers the areas of distributed graph processing, graph databases, elastic scaling systems, and hybrid cloud systems.

2.1 Distributed Graph Database Servers

Distributed graph processing systems can be used to execute user specified vertex functions in vertices distributed across multiple workers. In executing vertex functions, the vertices communicate with neighboring vertices iteratively to compute their local state. Hence this requires an efficient communication methodology between vertices to build a high efficient graph processing system. In this kind of distributed graph processing system, the network costs are the bottleneck for overall computation.

To overcome the above inefficiencies, an improved partitioning methodology is required for improving the locality of vertex communication. The two types of partitioning strategies: edge-cut and vertex-cut minimizes the number of edge or vertex spanning over multiple partitions. When it comes to real world application scenarios each vertex doesn't involve the same amount of communication overhead and also the network communication costs between each pair of workers are also not homogeneous.

Because of the low deployment costs and high scalability requirements currently it is common to run graph analytics in cloud environments. Modern cloud environments are geo-distributed. In a distributed environment like above the network link costs can differ by orders of magnitudes. These heterogeneous networks affect the overall communication costs and those should be considered when partitioning the graph.

GraphH is a distributed graph processing system which is designed to overcome the above-mentioned vertex traffic heterogeneity and network traffic heterogeneity problems [1]. GraphH is aware about the dynamic vertex traffic as well as the underlying network link costs. Taking this information into consideration the GraphH adaptively partitions the graph at runtime to minimize the overall communication costs.

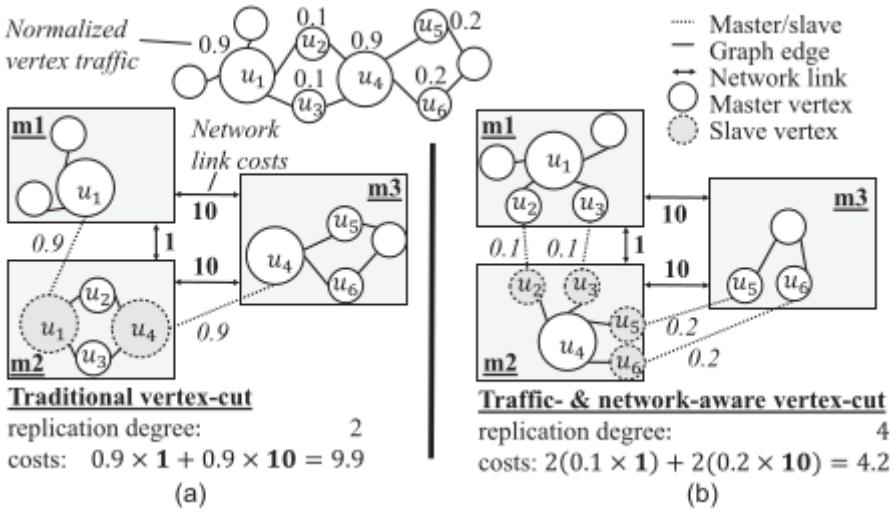


Figure 2.1: Traditional vs Traffic and network aware vertex cut algorithms [1]

They propose below three algorithms;

- H-adapt : To solve dynamic vertex traffic and network aware partitioning problem originally partitioned by H-load algorithm which uses to do the initial partition.
- Adaptive-∝: For online vertex traffic prediction
- Two new algorithms to solve two graph problems : Subgraph isomorphism and agent based Cellular automaton
- Evaluations on pagerank, subgraph isomorphism, and cellular automaton

This paper considers only vertex-cutting.

In H-load it first splits the graph into multiple partitions using a vertex-cut algorithm and then it maps the partitions to workers using a cost-efficient mapping. In the first stage of H-load it provides a clustered partitioning which consists of partition clusters as the output. These partition clusters consist of two partitions with many shared replicas and they are expected to reside in the same partition cluster. H-load also assumes that the cluster consists of worker clusters with low intra-cluster and high inter-cluster costs. They use a number of worker clusters to produce the partition clusters. In the second stage it assigns the partitions to workers. The minimal cost mapping of partitions to workers would assign two partitions with higher inter partition traffic to workers connected via a low-cost network link. H-adapt is an algorithm for dynamic edge migration. In this algorithm each worker locally reduces communication costs by dynamically migrating edges to other workers. In summary H-load mainly focuses on cost-efficient partitioning whereas the focus of H-adapt is on dynamic edge migration. None of those algorithms focuses on guaranteeing SLA for the jobs. GQSM

model which we are proposing mainly focuses on guaranteeing SLA for the jobs with observation-based admission control.

To evaluate the partitioning algorithm which has been proposed by these researchers they have implemented three graph algorithms PageRank, Subgraph Isomorphism and Cellular automation.

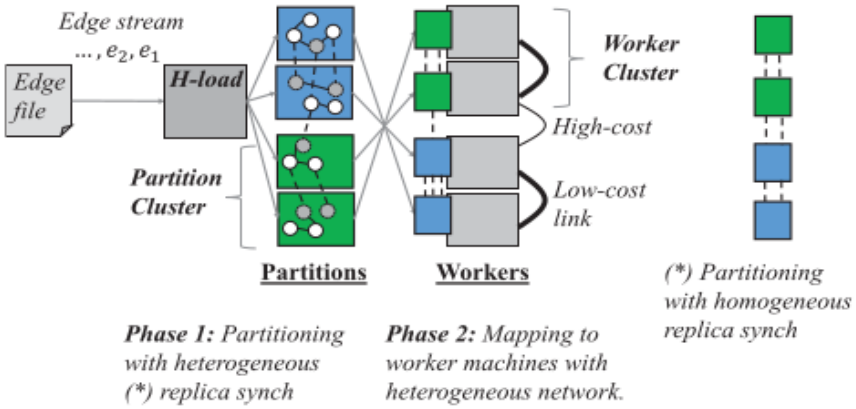


Figure 2.2: H-load approach overview [1]

Rishan *et al.* has focused on developing a partitioning framework called Surfer which improves the performance of partitioning the graphs [18]. They have used a tree structured multi-level graph algorithm partitioning for the framework where the root node represents the graph to be partitioned and the leaf nodes represents the graph partitions generated by the algorithm. They have defined an ideal partitioning sketch as a partitioning sketch with optimal bisections in the graph database at each level. The purpose of this optimal bisection is minimizing the number of cross partitioning edges between the partitions generated by the multi-level graph algorithm. They have observed that using this optimal bisection introduces relatively good partitioning quality, approaching global optimum.

A set of machines obtained from a cloud venter is used to carry out the graph partitioning. They have modeled these sets of machines as machine graph. Machine Graph is constructed by calibrating the network bandwidth between two machines in a set. The edge thickness between two machines in the machine graph represents the bandwidth between those two machines. Higher the thickness higher the bandwidth.
The framework partitions the machine graph and data graph with multi-level bisections They have used Metis graph partitioning algorithm to partition the machine graphs.

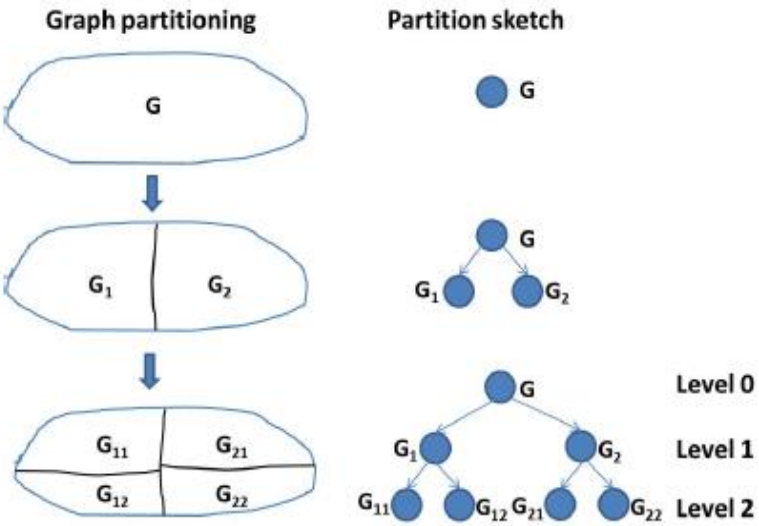


Figure 2.3: Correspondence between bisections and the partition sketch for the process of partitioning the graph [18]

Above mentioned researchers are focusing only on the graph partitioning problem. In our approach we are trying to develop a job scheduler which guarantees service level agreements for triangle counting using observation-based admission control algorithm

Most existing graph databases support either public clouds or private clouds. There is no proper graph database which supports hybrid clouds. Acacia [2] [3] [4] is a graph database which is an efficient and effective operation model for hybrid cloud graph databases. There are two broad categories of graph processing application types; first one is offline graph analytics which demand high throughput and second one is online graph processing that requires low latency. Acacia is focussing on the second category [4]. Acacia focuses more on graph storage aspects and secondary storage is also being used in Acacia. In distributed graph databases graph partitioning is a challenging issue. Acacia uses METIS [11] as the graph partitioner to reduce communication overhead between compute nodes.

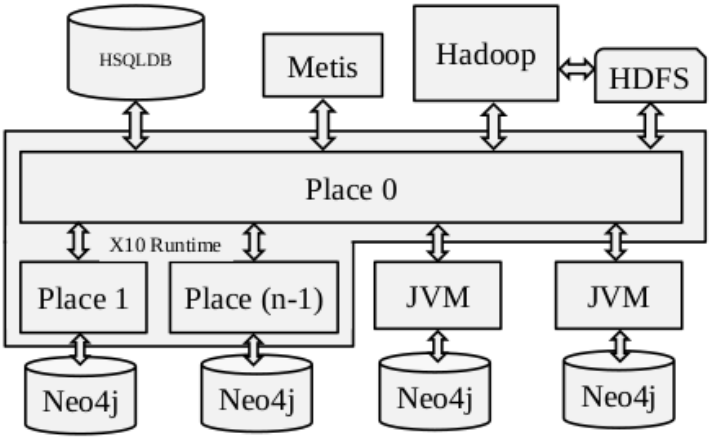


Figure 2.4: Acacia System Architecture [4]

Acacia uses a master-worker pattern which gives more control in system execution compared to other patterns (e.g., peer-to-peer). Acacia in general operates in a private cloud environment and starts using public cloud resources whenever there are resource limitations to maintain the SLA.

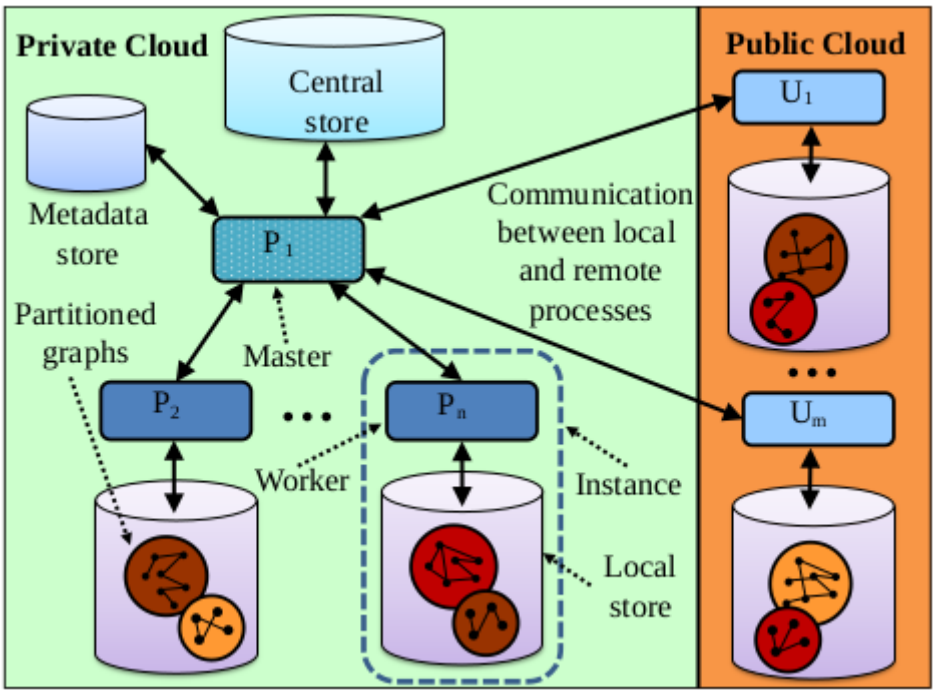


Figure 2.5: Overview of Acacia [4]

Assume that the local nodes have a limitation on its local data storage. When a new graph is uploaded to Acacia, it partitions the new graph into a number of partitions which is equal to the number of nodes in the Acacia cluster. Acacia keeps records on the local disk and CPU utilization. If that utilization violates the agreed SLA level,

then it considers migrating portions of subgraphs into the public cloud. Acacia then requests new Virtual Machine (VM) instances from the public cloud and initiates new workers in those public cloud instances. Then Acacia migrates the selected graph data to those new workers. When local graph data storage is increased by adding more volumes to the workers, then Acacia will adjust its partition distribution which leads to shutting down of public cloud workers.

This work differs from Acacia primarily by the fact that this project focuses on a job scheduling algorithm which guarantees service level agreements via observation-based admission control.

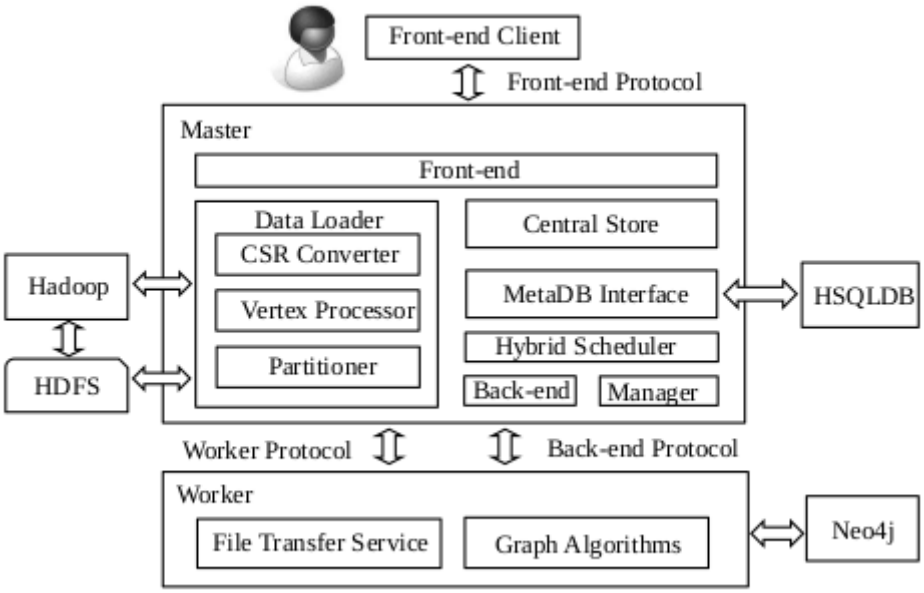


Figure 2.6: Components of Acacia [4]

JasmineGraph [20] is a graph database server very similar to Acacia. The main difference between the two servers is JasmineGraph is developed completely in C/C++ and Acacia was developed using X10 programming language. Figure 2.7 illustrates an overview of the design of JasmineGraph.

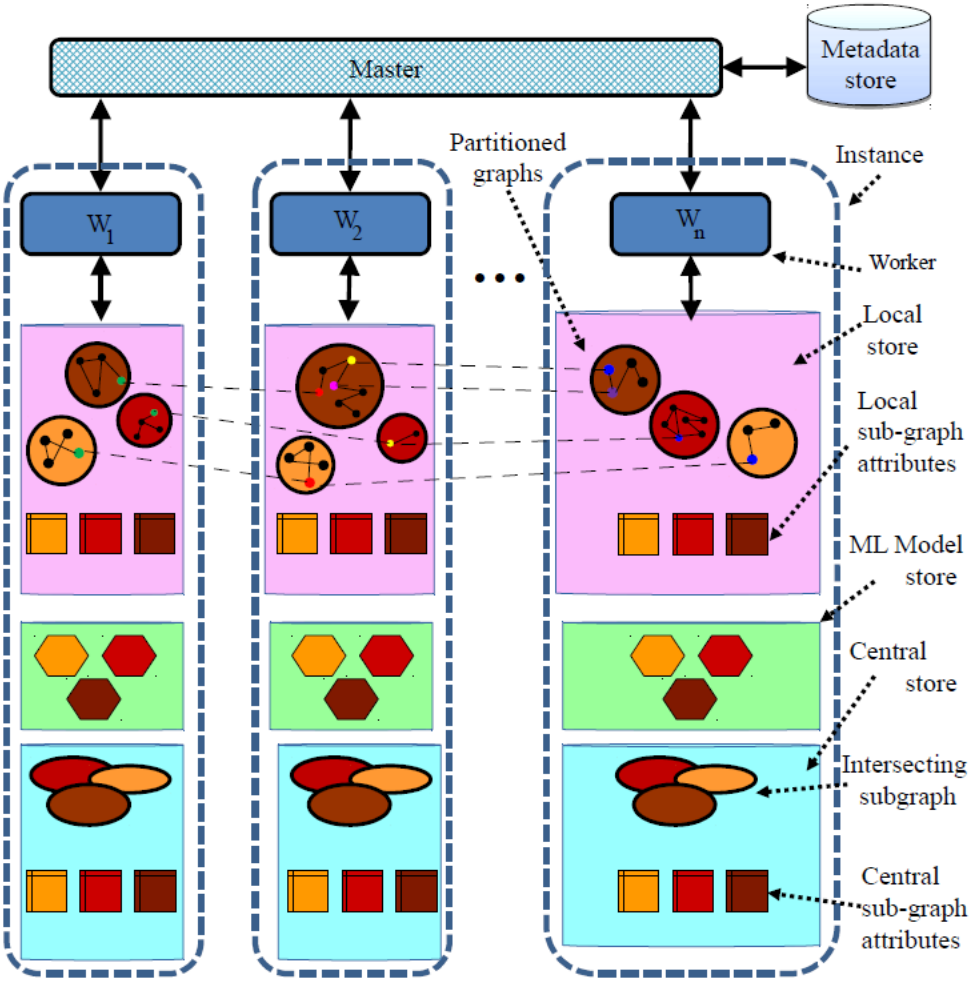


Figure 2.7: System Architecture of JasmineGraph

JasmineGraph has two main component categories called Master and Worker. System uses two types of communication protocols. One for communication between the external clients and the JasmineGraph system, and the other protocol is to communicate between Master and the Worker as well as between Workers.

Master is responsible for maintaining the status of the cluster, distributing work among workers and interacting with external users. External clients directly communicate with the JasmineGraph Master. A Frontend Service Session (FSS) is the connection which is used by the external clients with the JasmineGraph Master in order to do the communication. External clients use this FSS to achieve the tasks like upload a graph, check the uploaded graph list and shutdown the JasmineGraph cluster.

A worker is a standalone, non-distributed graph store. Workers are usually distributed among multiple hosts. Communication initiated from Master to the worker will be handled using the worker protocol. The communication initiated by workers to master is handled via Back-end protocol.

JasmineGraph uses a standalone SQLite database which is referred to as MetaDB of JasmineGraph. This MetaDB is used to store metadata information such as worker details and uploaded graph information.

JasmineGraph uses Metis Partitioner to partition the uploaded graph databases. After completing the partitions JasmineGraph uploads the partitions to the respective workers in the cluster. Each worker consists of two data stores: Local and Central stores. Partitioned subgraphs are stored in the local store. Information of the edges where the two vertices stay in two different local stores are maintained in the central store.

When a graph algorithm execution initiates, Master sends requests to all the workers in parallel and requests the workers to execute the respective graph algorithm for the specified dataset. When a worker completes its execution, it responds back with the response of the graph algorithm to the master. Master aggregates the results received by all the workers participating in the graph algorithm and then responds back to the external party who initiates the graph algorithm execution with the results.

Based on the above descriptions, Table 2.1 includes a comparison between GraphH, Acacia and JasmineGraph graph databases.

Table 2.1: Comparison Between GraphH and Acacia

Feature	GraphH	Acacia	JasmineGraph
Graph Partitioning	H-load	Metis	Metis
Cost Efficient Partitioning	Yes	No	No
Hybrid Cloud Support	No	Yes	Yes
Service Level Agreements for Triangle Counting Jobs	No	No. But when new graph is uploaded if the worker SLA is breaching due to new partition, those data will be automatically uploaded to public cloud. Also, later when there is sufficient space public cloud data will be migrated to private cloud and public cloud instance will be stopped if there are no other workers in that instance.	Yes, SLA guarantee will be available for triangle count jobs based on the resource consumption and the system load. New jobs will be accepted to the system after evaluating for sufficient resource availability.

Based on the above comparison we can observe that those three distributed graphs have their own specific advantages. But none of the graph databases guarantees service level agreements for triangle count jobs. Our approach will first check whether a particular job request can be accepted in to the graph database without violating the service level agreement of already running jobs as well as the new job requests.

2.2 Elastic Scaling

Elastic scaling is widely associated with private cloud systems. Elastic scaling is being widely used as a means of guaranteeing better performance of online services. One performance factor which is trying to achieve is the SLA of online services. Based on the system load of any given point it is feasible to use elastic scaling to scale up and down the resources. This approach can be used in an online services system to maintain the SLA of its services.

Data Stream Processing is one such example area where it is used to process streams of data to extract information of real-world events. There are multiple stream processing systems available which operate on various software/hardware environments. WSO2 Stream Processor [13] server is an example stream processing engine. It is a stream processor which is capable of operate on multiple clouds.

stream processing applications are more often deployed in private clouds. But face resource limitations due to unexpected loads on those systems. In these situations, there are multiple workarounds such as elastically scaling into an external cluster, load shedding and approximate query processing. However, most of them have multiple limitations such as not using the network efficiently.

There has been multiple work carried out in the area of elastic scaling of event processing systems. Implementing a distributed stream processor based on MapReduce on top of IaaS cloud platform such that it can be scaled up and down is one solution [14]. They introduce two main load balancing strategies. First one is always-on in which load balanced between local and cloud and second one is adaptive load balancing in which stream processing will move to cloud when the capacity in the local environment is not sufficient.

Another group of researchers has discussed how elasticity of data streams can be achieved using Cloud computing [15]. They mentioned elastic scaling based on input data rate and complexity of the operation. As most operators in stream computing are stateful, it is difficult to easily split them up or migrate them.

Stormy [16] is a system developed as “stream processing as a service” concept. It has been built to support scalability, elasticity and multi-tenancy. One major drawback of Stormy is that it assumes a query can be completely executed in one node.
But none of the above solutions have investigated reducing amount of data sent to public clouds in such elastic scheduling scenarios.

Elastic Switching Mechanism (ESM) [12] is a mechanism developed to reduce the overall average latency cost of event processing jobs considering the cost of operating system. ESM consists of two types of switching.

1. Data switching - Part of data is sent to public cloud
2. Query switching - A selected query is sent to the public cloud based on the characteristics of the query

ESM with compression can be used to handle out-of-order events more efficiently compared to techniques without compression. Data field compression technique in this ESM for compressing data sent from private cloud to public cloud.

ESM is designed to operate between private and public cloud. When required copies of the same stream processing engine which runs on private cloud is run on the public cloud. There is a profiler in the system which is responsible for gathering statistics of system performance. A scheduler is built in the system which will do the elastic scheduling functions based on the performance information provided by Receiver. A predefined value for average latency is used as the performance parameter. Time spent by an event in the stream processing engine is considered as the latency here. Average latency is being used as the performance metric due to the introduction of additional latency when switching to public cloud.

Publisher is responsible for emitting data stream to public cloud whenever required. Data stream compression is done by the compressor. Compression handler deployed in public cloud will intercept the messages sent to public cloud with compression enabled. ESM operations within the specified QoS constraints is shown in Figure 2.7 in private cloud and hybrid cloud scenario.

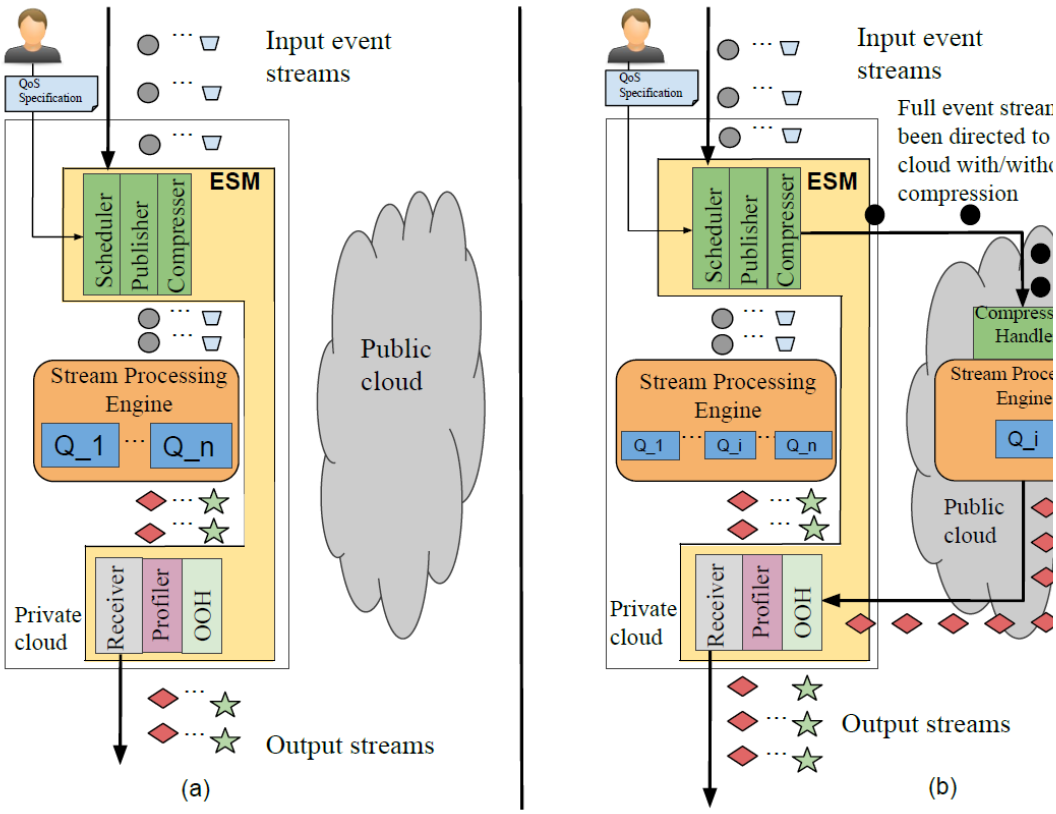


Figure 2.8: ESM operation. a) Private cloud only mode operation. b) Hybrid cloud mode of operation with data switching and compression [12]

Aforesaid solutions are focusing on data stream processing, but not on job scheduling algorithms in graph databases. In our approach we will be developing a job scheduler which guarantees service level agreements for triangle counting using observation-based admission control algorithm.

2.3 VM Load Balancing

VMs are deployed on top of host computers. When a load is imposed in a certain VM, that consumes the underlying resources of the host computers which the VM is deployed. If a certain host has multiple VMs and imposing a considerable load would result in high resource consumption of the host. If the VMs are used to deploy a system which should guarantee a SLA for its services, there is a high possibility of breaching the SLA. Therefore, a system which can intelligently manage the VM load can maintain a SLA for the provided services.

Li *et al*. gave a system review of the VM load rebalancing (VMrB). Frequent VM spawning and shutting down can cause the imbalance in load across host machines [17]. In cloud environments many VMs are created in a single HM or physical machine. To avoid HMs overloading and underloading the cloud computing load balancing technology distributes the VM across a set of coordinating HMs. VM instances get dynamically deployed when many users request the services. Here users expect a quick response time, service quality and user satisfaction which will result in less attention to load balancing after a certain time period. In offline VMrB recompute the VM placement on HMs for better load balancing. This team discusses on VMrB problem.

VMrB consists of two aspects. Any resource requested by VMs should be evenly spread across different cooperating HMs which is considering the perspective from the entire system. This is called inter-HM load balancing. The second aspect is to consider the perspective from a individual HM which needs to balance the amount of residual resources in each resource type. This is called intra-HM load balancing. Most researches focus only on either inter-HM load balancing or intra-HM load balancing. This research is focusing on both the load balancing aspects.

Challenge with inter-HM load balancing is to manage multiple types of resources (CPU, Memory and Storage). It is difficult to find an optimal solution with an optimal balance in each resource dimension. MOVMrB is a VMrB solution which optimizes the inter-HM and intra-HM load balancing. In addition, the research design a hybrid live VM migration technique which focus on reducing migration costs using an interval optimization method. This methodology discourages the useless VM migrations which decreases the number of VM migrations.

Existing VM migration algorithms use two methodologies to migrate VMs. The first one is the serial migration where the VMs are migrated one after the other when there is a set of VMs to be migrated. Second methodology is parallel migration where multiple VMs are migrated concurrently. When there are a larger number of VMs to migrate at the same time the parallel migration becomes faster than the serial migration.

In terms of migration time the sequential migration is faster than the concurrent migration. The migration time increases considerably for each node when increasing the concurrency granularity because the network acts as the bottleneck for migration. In the proposed hybrid migration strategy the information about the network for migration is being used. In the proposed hybrid migration strategy, each link carries out two simultaneous live VM migrations. Multiple VM migrations are being executed parallely on different transfer routes.

CHAPTER 3: SOLUTION VIA GRAPH QUERY SCHEDULING MECHANISM

This chapter focuses on the solution provided via Graph Query Scheduling Mechanism (GQSM) which is used for building a model to guarantee service level agreements for triangle counting via observation-based admission control algorithms.

3.1 Methodology

This section mainly focuses on the model which will be used to implement GQSM. Each section below focuses on a specific area of the GQSM model.

3.1.1 Job Classification

This project focuses on maintaining Service Level Agreements for jobs. One key decision to make is whether to provide SLA for each job executing in the system or only to maintain SLA for a selected set of jobs. Following are a few important aspects to consider when deciding that.

- When a job is scheduled in the system it will consume resources which are necessary to run that job.
- Accepting each job for processing will result in high resource utilization when there are a considerable number of jobs running in the system. Each job running in the system will consume the resources it needs to run the job. When there are a lot of running jobs in the system then the resource usage of that environment is high and there is no control over the resource utilization in the system. This will lead to overloading the system at some point.
- Due to high resource consumption, there won't be sufficient resources to allocate for the jobs which are running at the moment as well as the jobs which are planned to onboard in the future will also be affected by the low resource availability.
- Due to lack of sufficient resources, all the jobs which are executing in the system will be impacted. Lack of resources will lead the jobs to run slower. Hence it will take more time to complete a given single job.

Above factors show that maintaining SLA for each job in the system is not practical. Therefore, the proposed model will maintain SLA only for a specified set of job requests. This specific set of job requests will be given a higher precedence over the other job requests and the system will allocate resources on priority basis for this set of jobs to achieve the SLA.

System needs to use some sort of a parameter for deciding whether to give higher precedence for a given job. A parameter called ‘job priority’ is used to identify the priority of the job request. Job priority will be assigned to each triangle count job request. Based on the job priority all the triangle count job requests receiving to the system will be categorized into two main categories. Those are,

- High Priority Jobs
  JasmineGraph server maintains an SLA value for high priority jobs.
  These jobs will be given the precedence which will make all the low priority jobs preempted if any of them are running.

- Low Priority Jobs
  JasmineGraph server doesn’t guarantee a SLA for low priority jobs.
  Resources for low priority jobs will be allocated when no high priority jobs are running.

User who initiates the triangle count job has the ability to provide a priority for the job request. A default low priority value will be assigned to all the triangle count job requests if a priority value is not provided by the user. Planned model of the job scheduler handles only jobs with two priority levels, high priority and low priority. Handling jobs with multiple priority levels is not in the scope of this project and it is a future work which is planned to do.

3.1.2 Calibration

The main purpose of the calibration phase is to calculate the best-case SLA for a given graph. System also captures the resource usage information and the maximum resource consumption each server can handle without compromising the SLA. It is not practical to maintain the same exact SLA value which was obtained during the calibration run. But if an upper bound can be derived based on the SLA obtained then it can be used as the guaranteed SLA value.

Two approaches can be used to derive the guaranteed SLA using the collected SLA during the calibration phase. First approach is to add a constant time period for the collected SLA and derive the guaranteed SLA. There are complex graph datasets which take a lot of time to execute triangle count algorithm as well as there are other simple graph datasets which don't take much time. Therefore, deciding a constant time period which matches both these graph types is hard.

Second approach is to add a certain percentage of the collected SLA as a margin and then derive the guaranteed SLA. A larger margin value will be added for the graphs which takes a longer time to execute whereas a small margin value will be added for

the graphs which takes comparatively shorter to execute. This approach can be used to derive more reasonably guaranteed SLA. When using this approach, a proper value for the percentage has to be derived. Selecting a higher percentage value will result in a high variance in the response times of high priority jobs which is not a good approach for a SLA. Using a lower value will provide a guaranteed SLA value which is closer to the collected SLA. It is hard to maintain the same exact SLA value as the collected SLA. Therefore, using a percentage value like 5% will be more reasonable than selecting a higher or lower percentage value. Equation 1 below can be used to obtain guarantee SLA value.

Guaranteed SLA = Collected SLA from Calibration * 1.05	……. (1)

There are two types of calibrations available in JasmineGraph. Type of calibration used is decided by the state of the JasmineGraph cluster at that moment. Below is a detailed discussion on each of the calibration types available in the system.

3.1.2.1 Initial Calibration

This is the calibration which will be conducted for the first few graphs of the JasmineGraph cluster after the initial setup. During initial calibration an uploaded graph is selected and then initiates a triangle count job for that graph. Load average data will be collected periodically until the triangle count job execution completes. No other job requests should be served by the JasmineGraph cluster throughout the calibration of the triangle count algorithm for the selected graph. The main objective is to capture the resource consumption only for the triangle count job execution for that graph and also to capture the response time to derive the SLA.

After successful execution of the triangle count job, the system persists the SLA value derived based on the triangle count job response time. The system also persists the load average data collected periodically. Collected periodic data will be used for analyzing the load average curves for that graph.

3.1.2.2 Auto Calibration

Auto calibration of a graph begins when a triangle counting is initiated for an uncalibrated graph along with triangle count for calibrated graphs. Auto calibration allows uncalibrated graphs to be calibrated by running them along with the calibrated graphs. All the job requests should be for the same calibrated graph in order to begin the calibration of the uncalibrated graph. Figure 3.1 illustrates the flow chart for the auto calibration process.

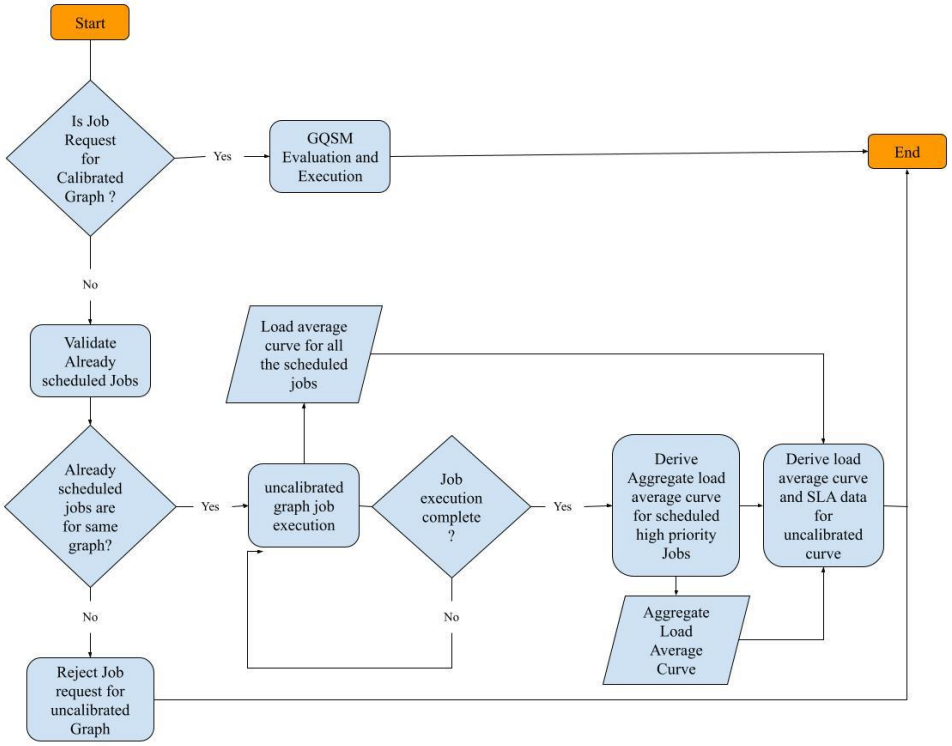


Figure 3.1: Flow chart for auto calibration process

Once a job request is received, the JasmineGraph server first checks whether the graph in the job request is a calibrated graph. If it is a calibrated graph then the server pushes that job request to the job request queue. For those requests the GQSM model which will be introduced in the next section gets applied and the scheduling decision will be taken by the GQSM.

If the graph in the job request is not a calibrated graph, then the server validates whether all the running jobs are for the same graph dataset. If all the running jobs are for the same graph dataset, then the server schedules the triangle count job for uncalibrated graph and the server periodically captures the load average information until the job execution completes. Upon the completion of the job execution, the server gets all the running calibrated jobs and constructs the load average curve data and then derives the actual load average curve for the calibrated jobs.

To obtain the load average curve data for the uncalibrated graph, the server calculates the difference of the curve data it captured after scheduling the uncalibrated graph and the curve data it constructed by aggregating all the scheduled calibrated graph.

After obtaining the guaranteed SLA value and resource usage information then in the calibration phase itself the cluster is subjected to a load test. During this load test the cluster is loaded with triangle count requests for the graph which was calibrated earlier and the system tries to identify the maximum resource consumption it can allow

without exceeding the guaranteed SLA for that graph. This will help the graph database to identify the maximum resource consumption it can allow such that it can accept multiple job requests in parallel. This resource consumption information is persisted in the system to be used in the job scheduler algorithm.

For a particular graph, SLA value varies depending on the underlying resource allocation for the server and the number of workers used to run the graph. This means the SLA values for a particular graph varies from environment to environment.

For a given environment for a given graph, best-case SLA can be collected when all the resources are exclusively available for the execution of the triangle count algorithm of that particular graph. All the resources are available for allocation means that there are no other tasks running in the system. Hence during the calibration of a particular graph no other tasks should run in the system. If a certain graph is in the calibration phase, then no other tasks will be accepted into the system. After completing the calibration task, the cluster will accept the jobs as usual.

There are two types of data which are being collected during the calibration phase. Those are the SLA value and the resource consumption values. SLA value means the time it takes to run the triangle count algorithm and return results for a particular graph during the calibration phase.

There are multiple metrics which can be used to measure the resource consumption such as CPU usage, Memory Usage and Disk Usage. Using multiple resource consumption metrics makes the analysis difficult as there are a huge number of factors to be analyzed. Therefore, it is ideal to use a minimum number of factors to analyze resource consumption. In a Linux environment there is a single metric which can be used to analyze resource consumption which is Load Average [19]. Load Average of a Linux environment denotes the average system load on the server for a defined period of time. Therefore, in this project we use the Load Average (workload) as the main factor to measure the resource utilization of the servers.

3.1.3 GQSM Model

Based on the above discussed factors we have come up with the following model for job processing. The model which will be discussed below is implemented using queuing theory. We name this proposed queuing mechanism as Graph Query Scheduling Mechanism (GQSM). Figure 3.2 below depicts how GQSM is modeled overall in JasmineGraph.

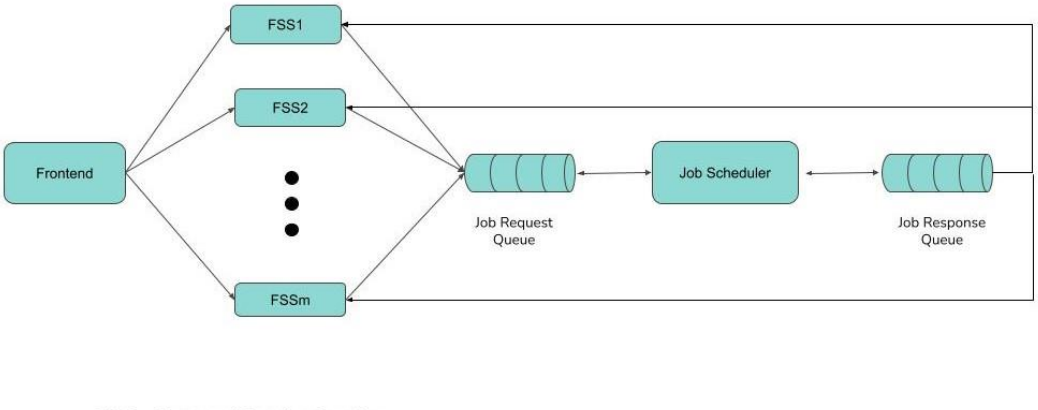


Figure 3.2: Graph Query Scheduling Mechanism (GQSM)

Frontend Service Session (FSS) in JasmineGraph is a connection established with the JasmineGraph server which is used by external users to communicate with the server. Each job is added to JasmineGraph via this created FSS. As soon as the job request is submitted to JasmineGraph, the response time calculation begins. All the job requests received by the JasmineGraph Frontend gets pushed into a queue called *Job Request Queue* immediately.

Job Scheduler is the component which is working with the job requests received to the system. It is responsible for taking the following key decisions on the job requests received to the system.

- Identify the job request type (high priority job or low priority job)
- Schedule jobs
- High priority job execution start time
- Drop high priority job

Once every Δt seconds the Job Scheduler keeps checking the job request queue. By the time it checks if there are jobs available, the job scheduler picks them up for scheduling. A job can be in the job queue for a maximum of Δt seconds and that time is also included in the response time.

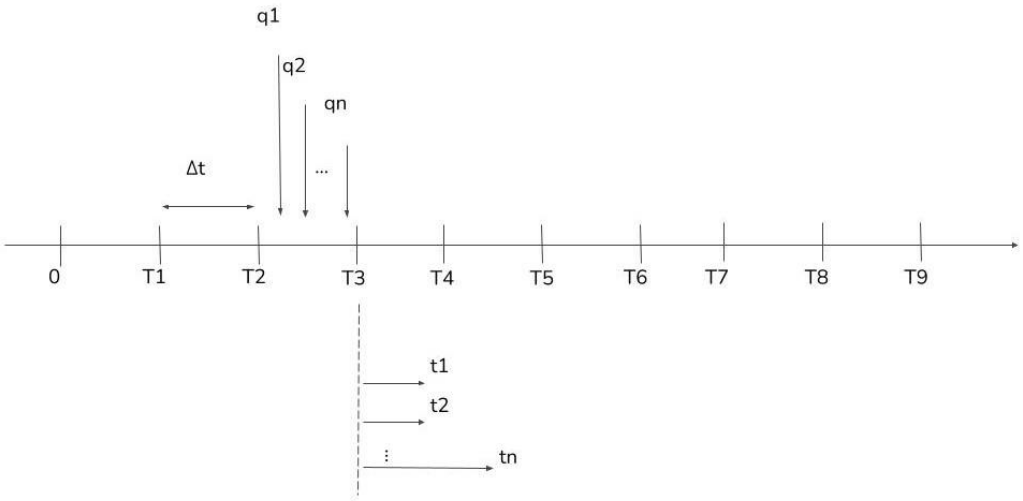


Figure 3.3: Job Request Pickup and Scheduling by Job Scheduler

Job Scheduler should be aware about the resource usage of the system. Based on the resource usage the scheduler should be capable of deciding which jobs are going to be accepted into the system without affecting already executing high priority jobs.

Furthermore, it should have the capability to decide whether a certain job can be accepted for future execution without affecting the SLA. When a new job starts execution, the scheduler should be capable of deriving the overall resource consumption of the system for all the running jobs. With that prediction the scheduler will have an understanding of the resource usage of the currently running jobs for the whole job execution time. That will be a key factor when deciding whether to accept or reject new job requests.

Figure 3.4 below depicts the modeled high-level architecture of the Job scheduler. Also, a description on the job scheduler is available below.

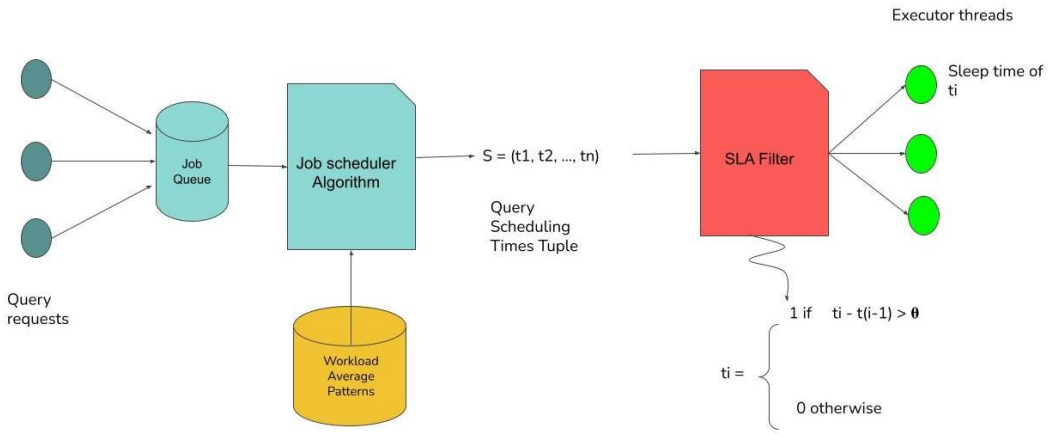


Figure 3.4: Job Scheduler High Level Architecture

Once the Job Scheduler picks up the available set of Job requests, each job request will go through validation and evaluation phases. During the validation phase the scheduler checks whether the job request is valid. These validations mainly include the request parameter validations like the Graph ID and requested Job category. A flow chart of the job scheduler is available in Figure 3.5.

Once the request parameters are successfully validated then the job request goes through an evaluation phase where the job scheduler checks whether the job request can be accepted into the system for execution. The main objective of this evaluation is to categorize the job requests into high priority jobs and low priority jobs.

Scheduler identifies the low priority job requests. System don't have to guarantee a SLA for these low priority job requests. Therefore, the system can accept the low priority jobs for later execution. First it checks the priority of each job request received. If a particular job request is a low priority one, then the scheduler will schedule the job request if there are no other high priority jobs scheduled in the system. If there are already scheduled high priority jobs, then the scheduler will accept that low priority job request for later execution.

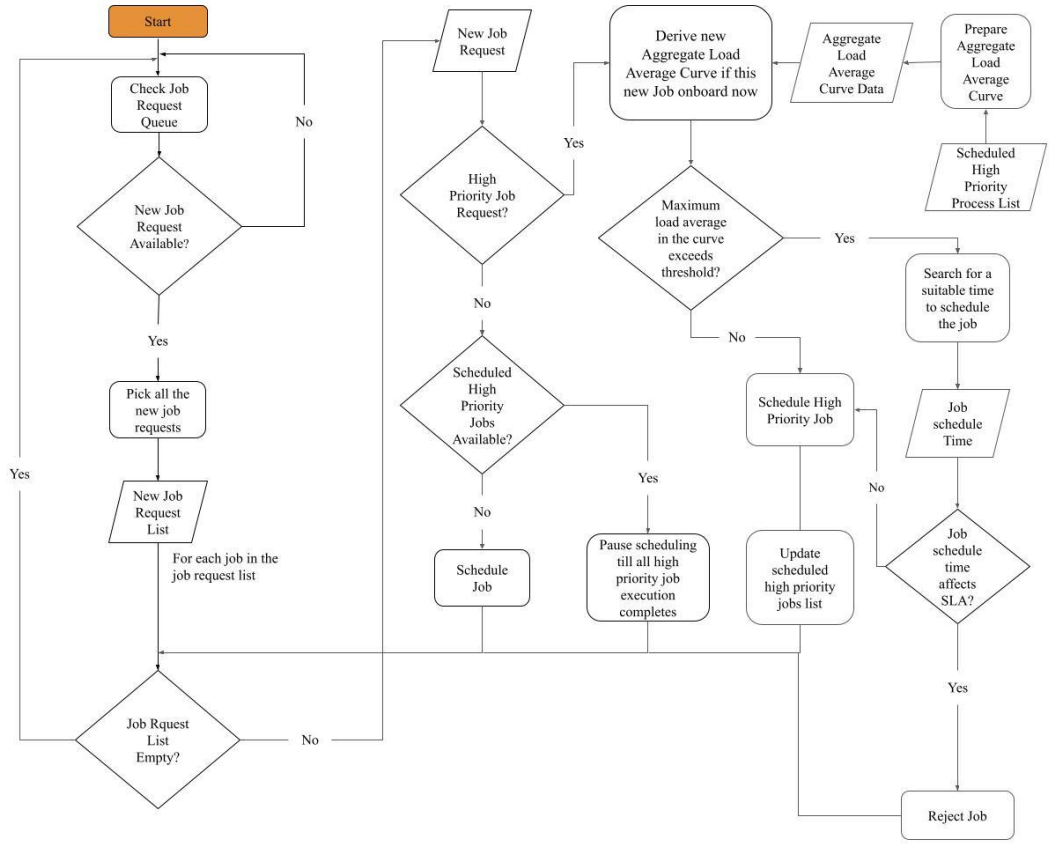


Figure 3.5: Flow Chart of the Scheduler

If the job request is a high priority job request, then the Job scheduler will go through the evaluation process in Algorithm 1 below.

The Job Scheduler gets the list of high priority job requests. JasmineGraph cluster also maintains the details of the high priority jobs which are already scheduled. First the Job Scheduler gets the details of all the scheduled high priority jobs and then builds the expected aggregate load average curve for each host by retrieving the load average data from the performance database. Table 3.1 illustrates how the expected aggregate load average curve data is derived for a single host based on the load average curve information retrieved from the database for each scheduled high priority job.

Table 3.1: Deriving Aggregated Load Average Curve

Job Request	Elapsed Time			
	0	**5**	**...**	**n**
J1	L10	L15	...	L1n
J2	L20	L25	...	L2n
J3	L30	L35	...	L3n
...	...	...	...	...
Jk	Lk0	Lk5	...	Lkn
Aggregated Load Curve	**L10+L20+L30 +...+Lk0**	**L15+L25+L35 +...+Lk5**		**L1n+L2n+L3n +...+Lkn**

Figure 3.6 below is an example of an aggregation of two curves. The figure illustrates how to derive the aggregate curve of two scheduled job requests for the same graph ID.

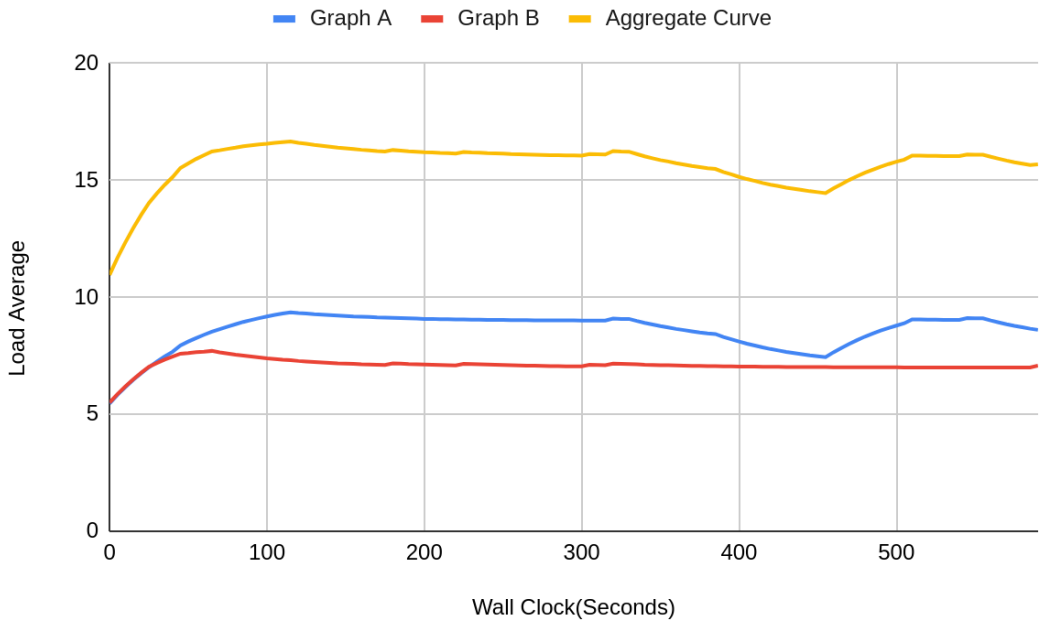


Figure 3.6: Deriving aggregate curve for two high priority jobs

After deriving the expected aggregated load average curve for all the hosts, the job scheduler then starts processing new job requests one by one. For each new job request first the scheduler obtains the expected load average curve from the performance database for all the hosts. Then for each host job scheduler derives a curve after aggregating the previously derived aggregate load average curve for the selected host and the expected load average curve retrieved from the database for the selected host

for the selected graph. This newly derived curve is the aggregated load average curve for that selected host if the new job is scheduled now.

If the maximum load average of this newly derived graph data doesn't exceed the load average threshold for the system the selected host the new job request can be scheduled without affecting the SLA. The load average curves will be derived for each and every host participating in the JasmineGraph cluster. Then the maximum load average value of the graph derived for each host will be checked against the load average threshold of the respective host. If the respective load average threshold value is exceeded by at least one of the load average curves, the job request will get rejected.

If the maximum load average doesn't exceed the threshold value for each and every host, then the job can be scheduled immediately. Now there is a new job in the scheduled state. This will change the aggregate load average curve for all the hosts. Therefore, the job scheduler has to update the aggregate load average curve for all the hosts before processing the next job request.

If the maximum load average exceeds the threshold value in one or more hosts, then the job scheduler has to find a suitable time to schedule the job. For those hosts which exceed the load average threshold following steps are followed.

- The scheduler gets the aggregated load average curve derived based on scheduled high priority jobs.
- Calculates the load average curve data from the database for that specific host for that graph.
- Derives new load average curves if the job schedules for a future time and checks if the maximum load average data exceeds the load average threshold.
- For all the hosts which exceeded the load average threshold initially follow above steps and get the appropriate times to schedule the job.
- Out of the obtained times from all the hosts use the maximum time as the schedule time because if the scheduler schedules the job at that time none of the hosts won't exceed the load average threshold.
- There is a waiting time for the job request which will be included in the response time. Therefore, the response time will be increased and it can affect the SLA value for that job request. Therefore, the job scheduler checks whether the waiting time obtained from the above step is less than 10% of the agreed SLA. If it is less than 10% then the job scheduling proceeds. If the waiting time is more than 10%, then the job will be rejected.
- Again, derive the aggregated load average curve data for all the hosts based on the new scheduled job list.

After a job request is scheduled then the system will wait for results of that job and then will publish the result of that job to the job response queue. After submitting the job requests each FSS will keep monitoring the job response queue for the respective

job response. Once the job response is available in the job queue the FSS will pick the job response and inform the external user about the result of the request initiated.

3.2 Data Collection Model

The test data needs to be collected from the JasmineGraph cluster for two main environment types.

- Single server environment

  In this environment all the workers of JasmineGraph will be running inside a single physical server. All the workers will be consuming the resources of the same server. As all the workers are running on a single host there will be a high demand for the limited resources available in the underlying server. Therefore, we can use the experiment results which will be capturing for analyzing how the Job Scheduler behaves in an environment with resources which has a high demand.

- Distributed worker environment

  Workers of Jasminegraph will be running on two or more geographically distributed servers. Each worker will be consuming resources of the respective server which it is being deployed to.

There are multiple scenarios the experiments can run. These scenarios are modeled so that various workloads will be imposed on the server. This will be helpful in order to get some rich set of experiment results for analysis. Below are the two main scenarios that are planned to run and gather the experiment results.

1. Single Graph Single Job Request

   Use a single graph and load the environments with a single job request in order to gather the results for an environment with lower load.

2. Single Graph Multiple Job Requests

   This scenario can be used to capture and analyze the experiment results for an environment with higher load using the same job request.

When running the experiments, it is important to capture all the data required for analyzing the workload patterns in each of the servers. During the experiments we need to analyze how the predicted workload prior scheduling the job requests compares to the actual workload of the environment when the job requests are scheduled. To do

that comparison, the following data is planned to be collected from each of the servers participating in the experiments.

- Workload Average

  Workload average in each of the servers participating in the experiments will be collected once every ‘n’ seconds for the period of running the triangle count algorithm for a given graph. Value of ‘n’ should not be a large value because then only a few load average data can be collected when running the triangle count algorithm. Therefore, ‘n’ has to be a value which provides a good set of experiment data for analysis.

  Collecting load average in the JasmineGraph cluster is achieved through the support of underlying C language APIs [22]. After starting the JasmineGraph cluster the system keeps logging the load average of the system. The job scheduler triggers once every two seconds and picks the jobs in the queue. For making the load average changes visible in the system and to collect sufficient data for the load average curve the system logs load average information once every 5 seconds into the log file. This is helpful for analyzing how actually the system load average behaves when scheduling triangle count jobs.

  When the workers are distributed across multiple hosts, then during the JasmineGraph startup process the Master appoints a host manager for each of the hosts participating in the cluster. This host manager is responsible for collecting all the statistics and logging all the statistics for that respective host.

- Resource consumption

  Resource consumption measures like CPU usage, memory usage will be collected from each of the servers participating in the experiments. These metrics won’t be used for the job scheduler algorithm to maintain the SLAs for the job requests. Rather these statistics can be used to analyze how actual underlying resources are being consumed by the workers while achieving the assigned task.

  To collect those performance data we have integrated nmon [21] which is a tool that can be used to capture performance data. With this integration the administrator who starts the JasmineGraph cluster has the capability to collect nmon dumps based on the requirement.

Table 3.2 below illustrates the format which will be used to capture the experiment results for each of the servers for a given scenario.

Table 3.2: Experiment Results Capturing Format

	Server 1		Server 2	
Wall Clock (seconds)	**Expected Workload**	**Actual Workload**	**Expected Workload**	**Actual Workload**
n				
2n				
3n				
4n				
…				

CHAPTER 4: IMPLEMENTATION OF GQSM

This chapter focuses on the implementation details of the GQSM and also the performance statistic collection framework.

4.1 Job Scheduler Implementation

Job scheduler responsible for monitoring the job request queue and doing the job scheduling if there are any new job requests in the queue. Therefore, the job scheduler is implemented to work in a separate thread. After initializing the job scheduler, it keeps monitoring the job request queue as long as the JasmineGraph cluster runs. Another queue was introduced to segregate the job requests and responses. That queue is called the job response queue and it is used to push the results of the job requests. After submitting job requests to the job request queue, the FSS keeps monitoring the job response queue for the result.

A job ID is used to correlate the job request and job response. Before the job request is submitted to the job request queue, the FSS sets the job ID for each job request. After submitting the job request to the job request queue, the FSS keeps monitoring the job response queue until it finds a job response with the submitted job ID. Once the job response is found, then the FSS sends back the response to the external user.

The GQSM implementation is illustrated in the pseudocode in Algorithm 1. The method accepts the job requests and the scheduled high priority job list in the JasmineGraph cluster as input parameters and then returns back a list of scheduled times for each request in the job queue.

Algorithm 1 : Scheduling Triangle Count Jobs in JasmineGraph

Input : scheduledJobsList {}, queuedJobsList {},
loadAverageThreshold
Output : jobScheduleTimeList {}
Description :

```
1:  aggregatedLoadAverageList ←{}
2:  hostLoadAverageList ← {}
3:  loadAverageList ← {}
4:  newAggregatedLoadAverageMap ←map (hostId,loadAverageList)
5:  hostList ←{}
6:  graphLoadAverageMap← map (hostId,loadAverageList)
7:  aggregatedLoadAverageMap ←map (hostId,
                                   aggregatedLoadAverageList)
8:  for each job in scheduledJobsList
9:      graphId ←job.graphId
10:     graphLoadAverageMap ←getLoadAverageMap(graphId)
11:     adjustLoadAverageMap(graphLoadAverageMap,job)
12:     constructAggregateLoadAverageMap(graphLoadAverageMap,
                    aggregatedLoadAverageMap);
13: for each queueJob in queuedJobsList
14:     hostScheduleTime ←{}
15:     requestGraphId ←queueJob.graphId
16:     graphSLA ←getGraphSLA(requestGraphId)
17:     graphLoadAverageMap ←getLoadAverageMap(graphId)
18:     newAggregatedLoadAverageMap ←
          deriveAggregatedLoadAvgMap(aggregatedLoadAverageMap,
            graphLoadAverageMap )
19:     hostIdkeySet ← getKeys(newAggregatedLoadAverageMap);
20:     for each hostId in hostIdkeySet
21:       hostLoadAverageList ←
                    newAggregatedLoadAverageMap[hostId]
22:       if max(hostLoadAverageList) < loadAverageThreshold then
23:         hostScheduleTime.push(0)
24:       else
25:         time ←
              getPossibleOnboardingTime(aggregatedLoadAverageList,
                                tmpAggregatedLoadAverageList)
26:         hostScheduleTime.push(time)
27:       if max(hostScheduleTime) * 0.1 < graphSLA then
28:         jobScheduleTimeList.push(max(hostScheduleTime))
29:         updateAggregateLoadAverageMap(
                        max(hostScheduleTime),
                            newAggregatedLoadAverageMap,
                                aggregatedLoadAverageMap)
30:       else
31:         jobScheduleTimeList.push(-1)
32: return  jobScheduleTimeList
```

Based on the schedule time suggested by the GQSM, the job scheduler starts scheduling the triangle count jobs in the job queue. The schedule time can be 0, -1 or a positive value. If the schedule time is 0, the job has to be immediately scheduled to start its execution.

If the schedule time is -1 then the job is a rejected job. The job scheduler prepares a job response message and sets the corresponding job ID of the job request and pushes it to the job response queue. Therefore, the external user who initiated the job request will be informed about the job rejection. If the schedule time is positive value that means the job is accepted into the system for later execution. In this case the job scheduler creates a new thread and assigns the job execution for that thread. Then the scheduler sleeps the job execution thread for a time period of schedule time returned by the GQSM. This will hold the execution of the job for a time period of scheduled time returned by GQSM. Once that time period is over the thread starts its execution and returns the triangle count result. Appendix B contains a snippet of the developed code of job scheduler.

4.2 Performance Statistic Collection

GQSM makes decisions solely based on the performance statistics collected during the calibration phase. These statistics include load average information, job response times, memory usage and CPU usage. Therefore, it is essential to store and maintain these performance statistics properly inside the JasmineGraph server.

JasmineGraph only had the metadata datastore to store metadata information about the JasmineGraph. These metadata include worker information, graph information and graph partition information. Metadata information differs from the performance statistic information. Therefore, storing performance statistics along with the metadata information inside the same metadata datastore is inappropriate. Hence maintaining a separate place to store performance related statistics was a major requirement.

A new datastore called performance datastore was introduced to overcome this issue. As the name implies it is intended to store all the data related to the performance aspect of JasmineGraph. Figure 4.1 illustrates the updated architecture of JasmineGraph after introducing the performance datastore.

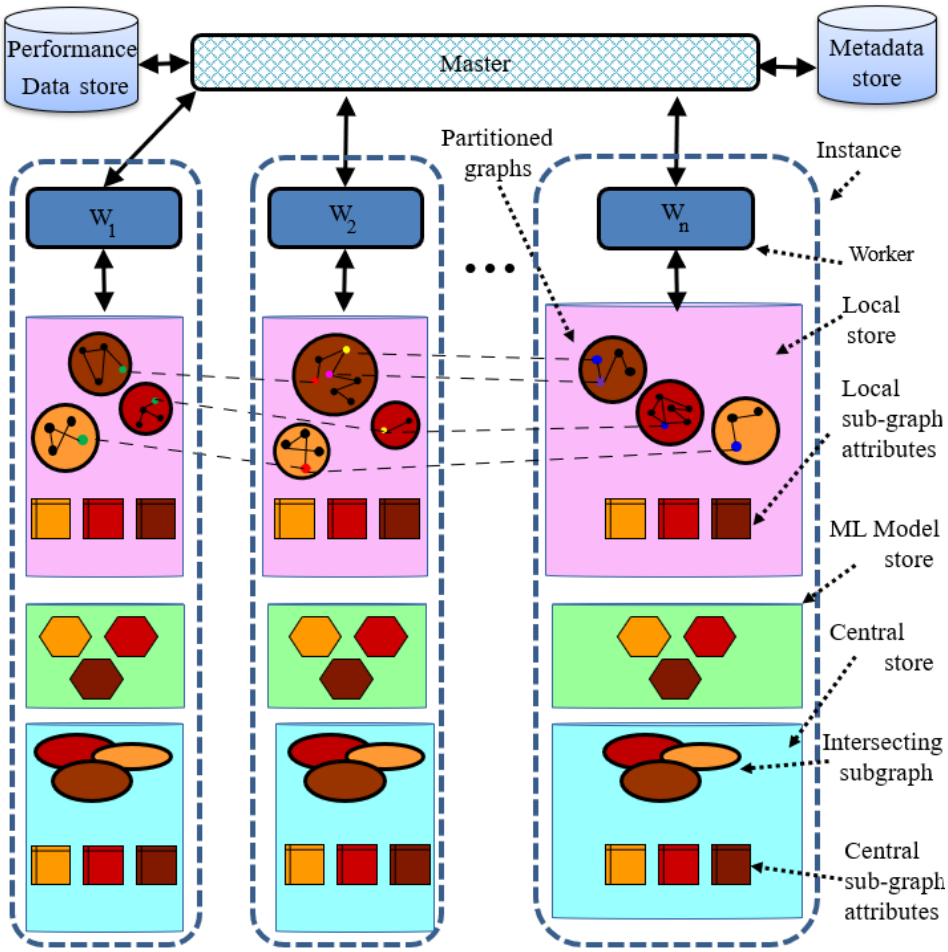


Figure 4.1: New JasmineGraph architecture

One of the main statistics collected during the calibration phase is the load average curve data. That curve data is being collected periodically. To achieve this in the JasmineGraph with the initiation of calibration phase, a new thread called the statistics data collection thread is spawned and assigns the task of periodic collection of load average data to that new thread. It collects the load average data periodically till the calibration phase completes. Then it processes the collected load average data and stores them in the performance database.

Figure 4.2 below depicts the high-level flow of the calibration phase implemented.

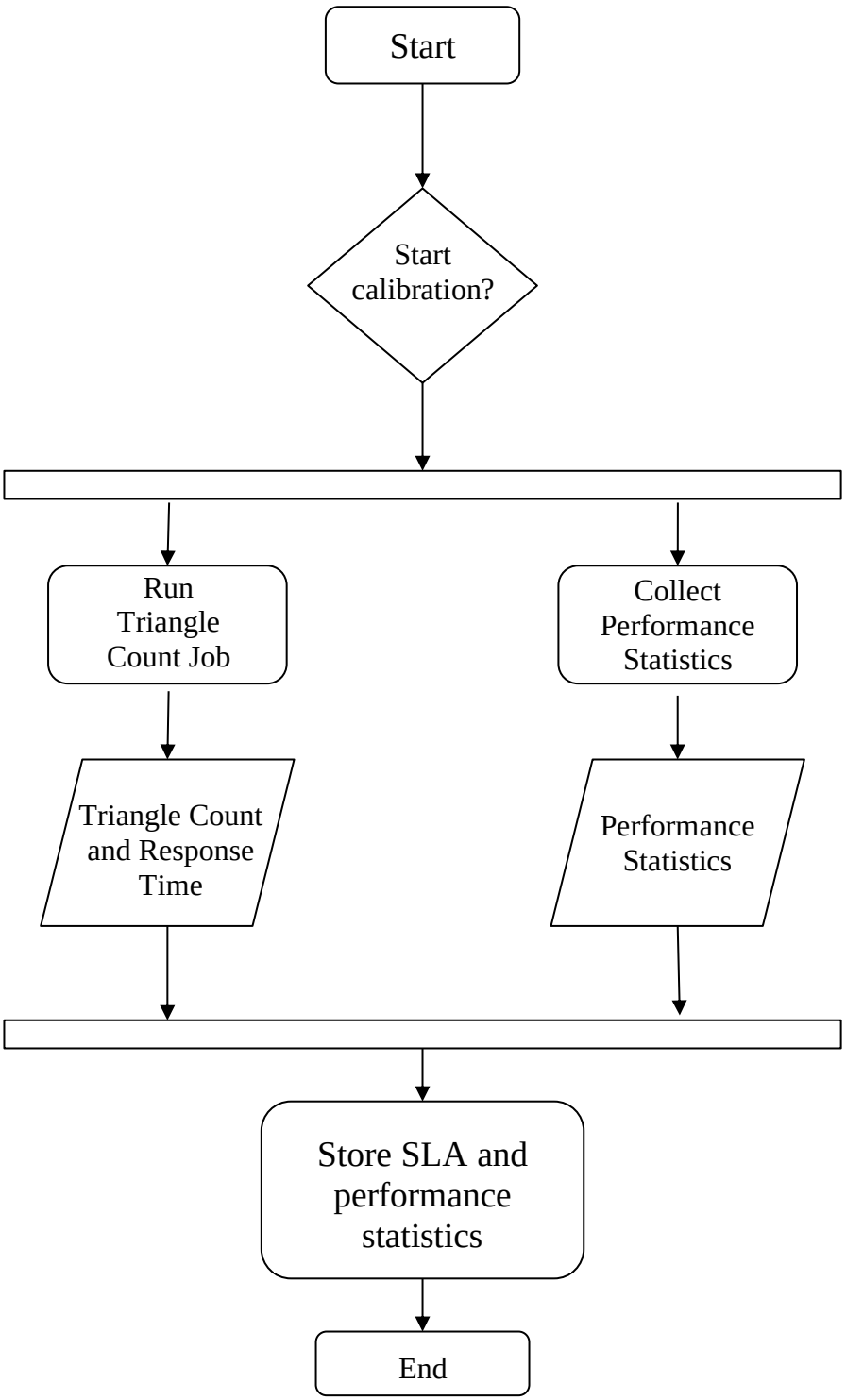


Figure 4.2: Flow chart for job calibration

Performance evaluation requires the load average information to be collected in some central place. This is achieved by logging the load average information to log files periodically. Once the job execution completes the log files can be processed to get the actual performance statistics related to the execution. If the job execution fails, the log file will be discarded.

A new thread is spawned with the initiation of the JasmineGraph server. That newly initiated thread is responsible for logging the load average information into the log files periodically. During experiments it was observed that there is no considerable resource consumption incurred due to this newly initiated thread. This thread terminates only when the server shuts down. Appendix C includes a snippet of the implementation of performance statistic collection.

CHAPTER 5: EVALUATION

This chapter focuses on evaluation of the proposed GQSM model which was modeled and implemented as described in the previous chapter.

5.1 Experimental Setup

To collect data and to evaluate models JasmineGraph cluster was used in two different environments. First one is on a single server environment where the JasmineGraph cluster will be started in a single host. Second one is in a distributed environment where some of the workers will be started in a different host than the Master.

5.1.1 Computing Infrastructure Setup

It is critical to use near production servers to run the experiments. In this experiment setup we used two servers.

JasmineGraph multiple dependencies to be installed in each host participating in the cluster. Therefore, the initial host preparation requires a considerable effort to install all the dependencies. To mitigate this issue JasmineGraph has been dockerized [34]. In a docker image all the dependencies are installed and for hosts to run docker images only docker engine installation is sufficient. Using the JasmineGraph docker image, it is possible to start up a full JasmineGraph cluster using docker containers. In a containerized cluster Master and each worker runs in its own containers. Figure 5.1 illustrates how the JasmineGraph cluster runs in a containerized environment.

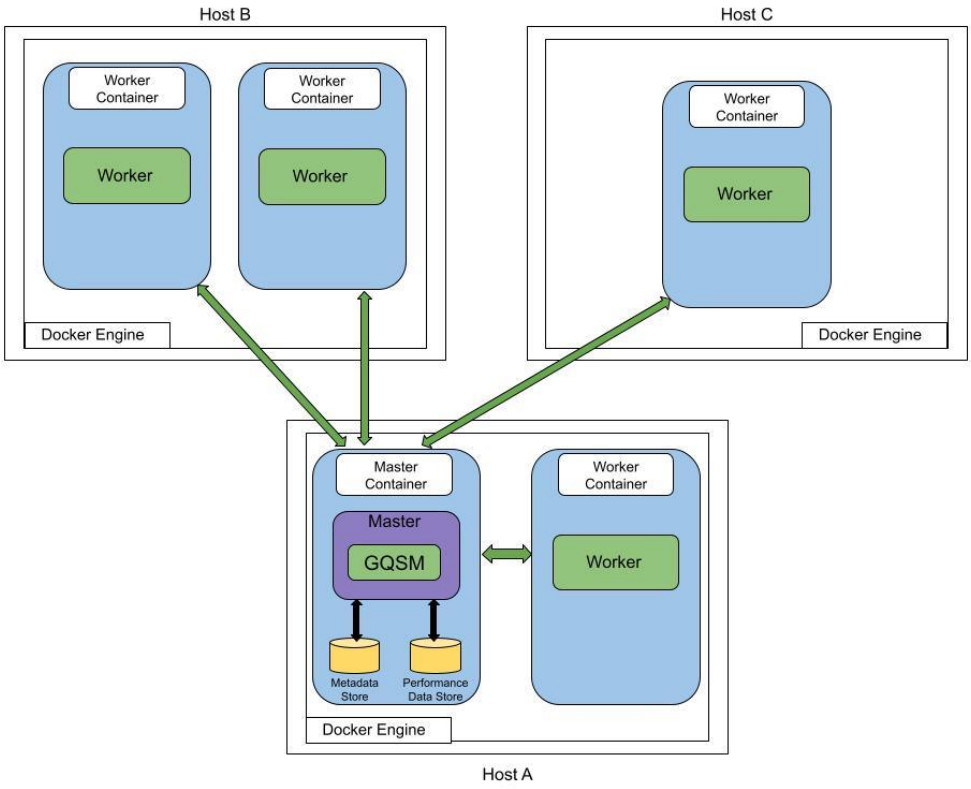


Figure 5.1: JasmineGraph dockerized environment

During the initial startup of the cluster, the JasmineGraph Master gets started in the selected host using the Docker image. Then the Master container issues commands to the docker engines installed in each of the worker hosts to start a new container with worker profile using the JasmineGraph docker image available in those hosts.

A Jmeter [23] was used to generate workload in the environment and was deployed in a separate VM. Depending on the testing scenarios the Jmeter script was changed.

5.1.2 Resource and Workload Combinations

Table 5.1 below lists the hardware specification of the two servers which we used to run the experiments. These two servers were used exclusively for the experiments described in this thesis. Processes and the server loads were frequently monitored when running experiments to make sure that the experiment results are not affected by any other process running in the servers. Using the two servers, different combinations of JasmineGraph clusters were spawned. For each of those clusters different workloads were imposed to collect experiment data.

Table 5.1: Combination of hardware

Server	CPU Cores (Hardware Threads)	Memory (GB)
Server 1	80	62
Server 2	40	62

Various cluster setups were used for running experiments. Table 5.2 lists the details of the cluster setup which were used to impose the workloads. This table also lists the number of workers spawned in each of the servers during each cluster setup.

Table 5.2: JasmineGraph cluster setup

Server	Cluster 1	Cluster 2	Cluster 3	Cluster 4
Server 1	4	2	8	4
Server 2	0	2	0	4

Figure 5.2 below illustrates the master worker distribution of cluster 1. Similarly Figure 5.3, Figure 5.4 and Figure 5.5 illustrates the master worker distribution of cluster 2, cluster 3 and cluster 4 respectively.

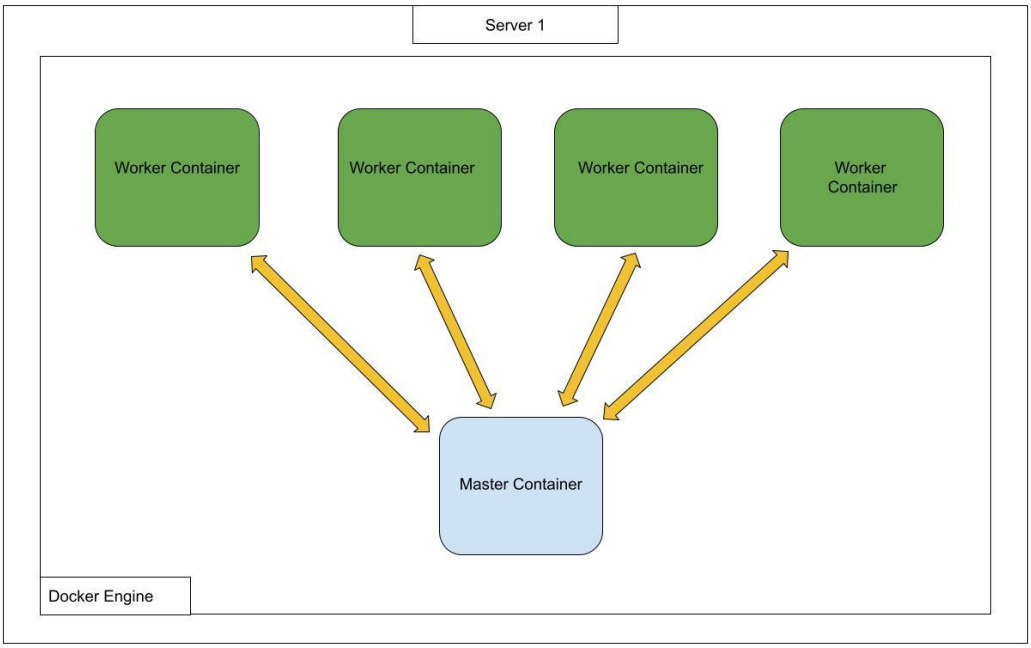


Figure 5.2: Master worker distribution of cluster 1

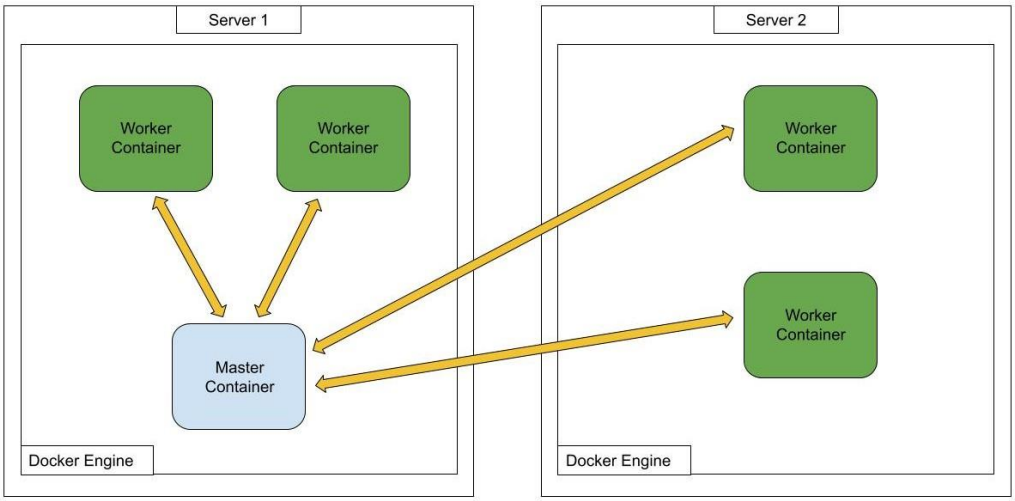


Figure 5.3: Master worker distribution of cluster 2

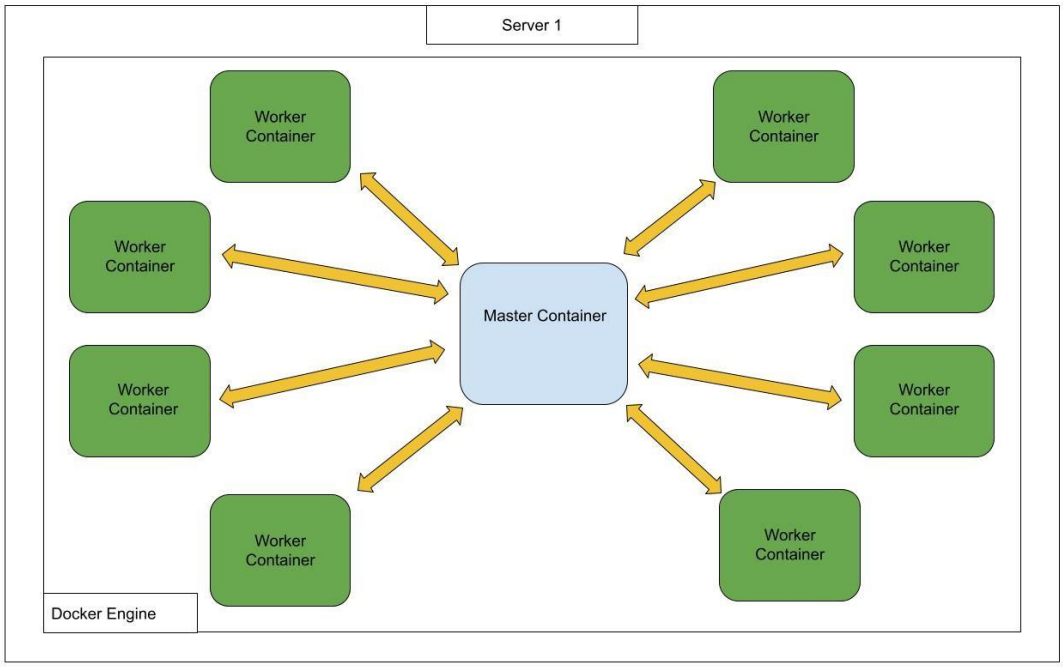


Figure 5.4: Master worker distribution of cluster 3

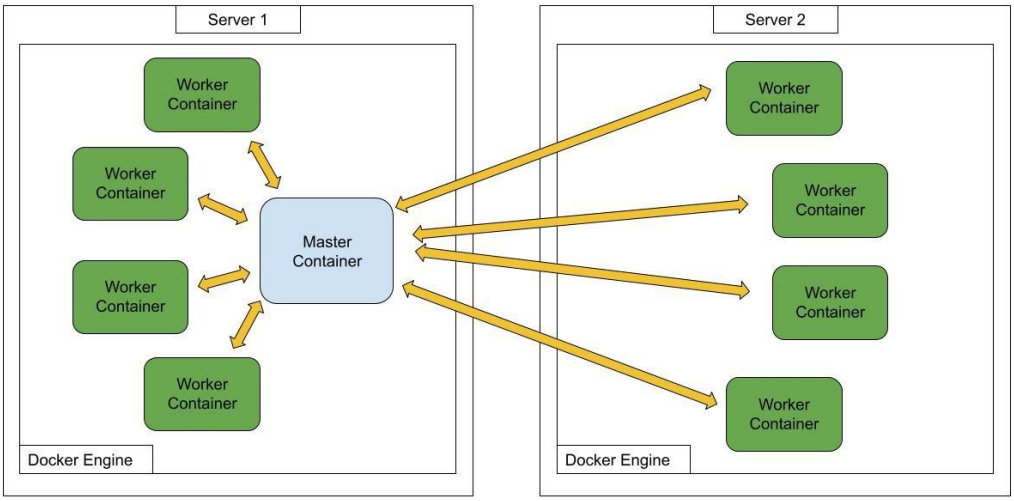


Figure 5.5: Master worker distribution of cluster 4

During each cluster setup the load test was carried out using 16,8,4 and 1 user in order. One user is sending one triangle count job request at a time. Table 5.3 includes the list of graph data sets and the statistics of each graph which were used for the experiments. The triangle count information of those graph datasets are available in the datasource and that information is used in this research to prove that the implementation of the proposed model provides accurate data. These were the best publicly available datasets for the experiments. Two of the graph datasets have slight variations in the triangle count which returns from JasmineGraph compared to the reported triangle count in the sources. Triangle count was verified for several other graph datasets as well and the results showed that the triangle count reported by JasmineGraph matches with the triangle count in the sources. Therefore, the differences of those two datasets were ignored.

Below two factors were considered when choosing graph datasets for running experiments in this research.

- Number of triangles

  The number of triangles in the graph database should be neither too low nor too high. If the number of triangles is lower, then it won't take a considerable amount of time to run the triangle count algorithm. When the execution time is less the quantity of load average data is also low. Therefore, the GQSM model will not have sufficient data to predict the behavior of that respective graph. If the number of triangles is too high then the system takes too long to run the algorithm.

- Time takes to run triangle count algorithm

  Reasonable set of statistics should be available to analyze the results properly. A given graph should run a considerable time to collect a good set of results.

Table 5.3: List of graph data sets for experiments

Graph	Edge Count	Vertex Count	Triangle Count - Reported by Source	Triangle Count - Returned by JasmineGraph
Youtube [24]	2,987,624	1,134,890	3,056,386	3,056,386
Epinions [25]	508,837	75,879	1,624,481	1,624,481
Web Google [26]	5,105,039	875,713	13,391,903	13,391,901
Astro Physics [27]	198,110	18,772	1,351,441	1,354,586

5.1.3 Data Collection

First, the system has to go through the calibration phase to find out the guaranteed SLA for each of the graph datasets chosen. One of the cluster combinations was selected at a time and then each graph was uploaded into the cluster setup and executed the triangle count algorithm against the selected graphs. No other jobs should run during this calibration phase. This is a onetime task per cluster per graph. Table 5.4 includes the SLA values obtained for each cluster after completing the calibration phase.

Table 5.4: SLA values for graphs

Graph	Cluster 1 SLA (seconds)	Cluster 2 SLA (seconds)	Cluster 3 SLA (seconds)	Cluster 4 SLA (seconds)
Youtube	599	750	461	606
Epinions	685	1134	570	568
Web Google	589	596	344	341
Astro Physics	205	403	108	498

During the calibration phase one other data which was collected was the load average data per graph per host per cluster. These data are automatically captured and recorded by the system. Load average information has to be captured periodically such that it can easily identify the maximum load average from the load average curve. Therefore, the period for capturing load average data was set to 5 seconds during the calibration phase. This load average data is used by the job scheduler to derive the aggregate load average curve when scheduling jobs in the system. Figure 5.6 illustrates the load average curve for Youtube graph database in cluster 1 setup. Appendix A includes the load average curves for Epinion, Google and Astro-Physics graphs.

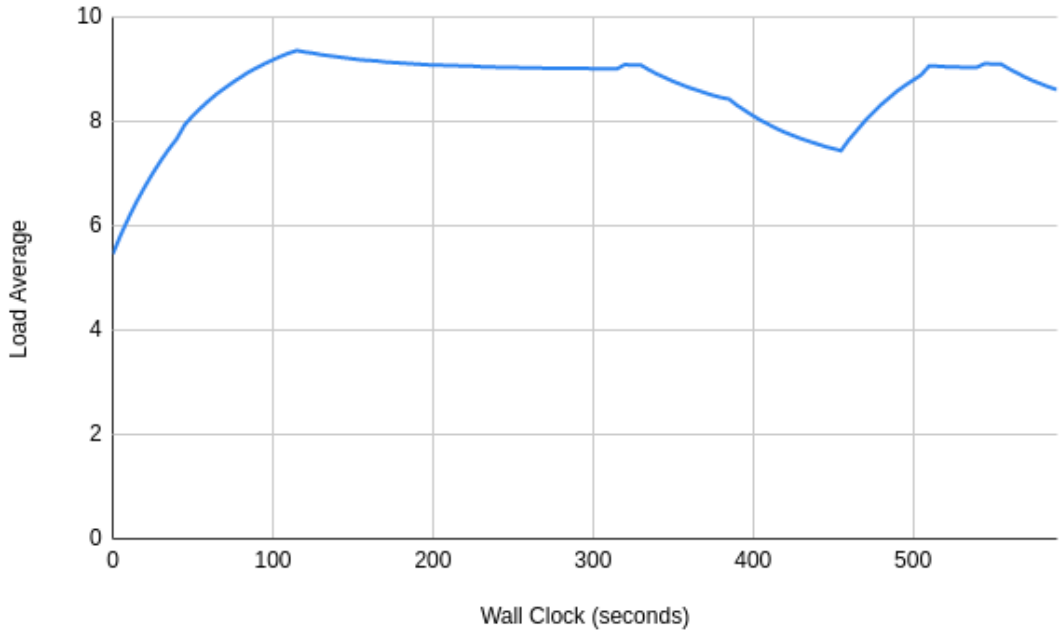


Figure 5.6: Load average curve for Youtube in cluster 1

It is not practical to guarantee the same exact SLA for each job request received. Therefore, a margin value has to be added to the collected SLA values and an upper bound has to be defined as the guaranteed SLA. A margin value of 5% of the collected SLA was added and the guaranteed SLA values were derived. Table 5.5 includes the guaranteed SLA values derived using the previous table data for each cluster.

Table 5.5: SLA upper bounds (Guaranteed SLAs) for graphs

Graph	Cluster 1 Guaranteed SLA (seconds)	Cluster 2 Guaranteed SLA (seconds)	Cluster 3 Guaranteed SLA (seconds)	Cluster 4 Guaranteed SLA (seconds)
Youtube	629	787	484	636
Epinions	719	1191	598	597
Web Google	619	626	361	358
Astro Physics	215	424	113	523

After deriving the guaranteed SLA for each graph then the load tests were started to identify the maximum load average that can be handled by the cluster without exceeding the guaranteed SLA value. This process is a manual process at the moment. The load tests for a given graph will go through multiple iterations with 16,8,4 and 1 user respectively. Each user sends a one triangle count job request at a time. For example, 16 user scenario sends 16 parallel triangle count job requests to the JasmineGraph database.

During each load test iteration, the system derives the expected load average curve for the load it receives based on the load average data recorded during the calibration phase. Using the derived expected load average curve, the system collects the maximum expected load average value. After all the jobs are successfully served the system collects the response time of each job and calculates the average response time for that iteration. Figure 5.7 illustrates the derived expected load average curve for the 4-user load iteration for youtube graph in cluster 1. Green highlighted point is the maximum expected load average for the respective load test iteration.

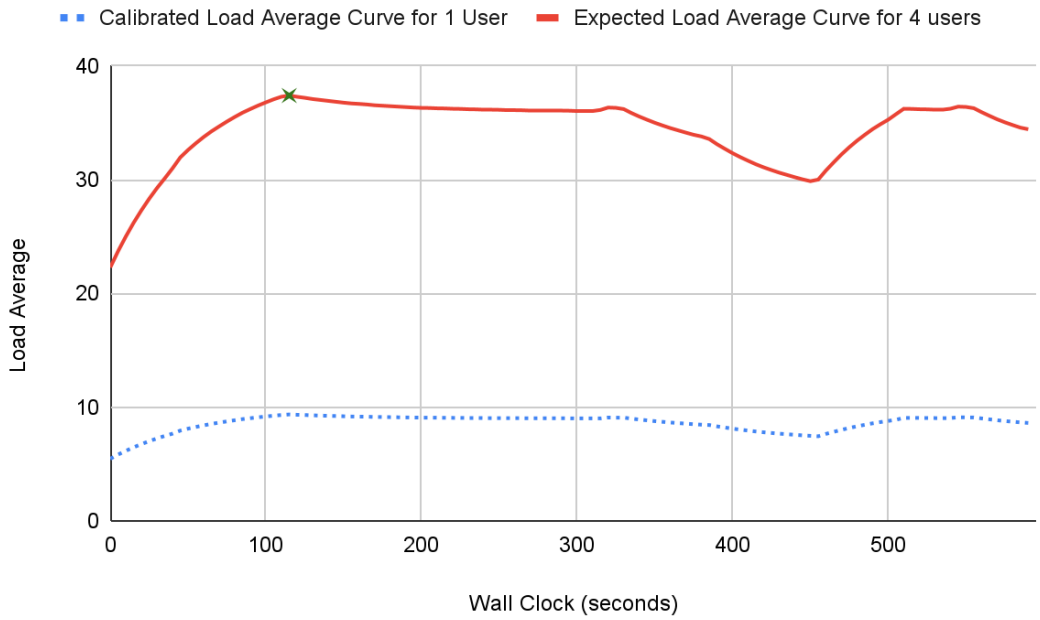


Figure 5.7: Expected load average curve

Table 5.6 includes the maximum load average data and the average response time data collected during each load test iteration for Youtube graph for cluster 1 setup.

Table 5.6: Collected maximum load average, guaranteed SLA and actual response time data

Number of Users	Maximum Load Average	Guaranteed SLA (seconds)	Actual Response Time (seconds)
1	9.36	629	598
4	37.403958	629	614
8	74.759508	629	665
16	149.584218	629	678

Our goal is to maintain the expected guaranteed SLA value at any load average. But the actual response time varies depending on the maximum load average in the system. Figure 5.8 illustrates the graph which was drawn based upon the above table data.

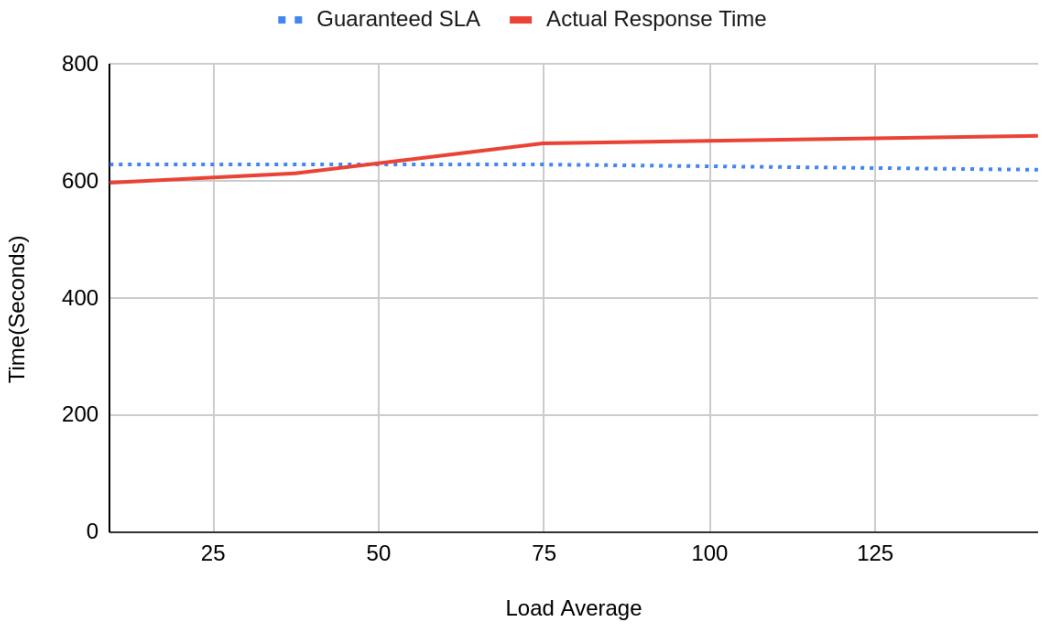


Figure 5.8: Expected SLA vs actual response time

Figure 5.8 illustrates that the actual response time is lower than the expected SLA value of the system when the load average of the system is less than 49. If the load average is higher than 49 then the response time is higher than the expected SLA. This means the system guarantees the SLA for youtube graph when the total expected load average is less than 49. When the load average is more than 49 the system fails to guarantee the SLA. Therefore, when accepting high priority youtube job requests into the system the job scheduler has to guarantee that the expected load average of the system doesn't exceed 49.

Similarly, all the other graph datasets were subjected to load tests and collected the same information. Based on the gathered information Table 5.7 includes the load average threshold the system can allow for each graph for the cluster 1 setup.

Table 5.7: Load average thresholds for cluster1

Graph	Load Average Threshold
Youtube	49
Epinions	82
Web Google	80
Astro Physics	71

For Cluster1, GQSM will use these load average thresholds when scheduling the high priority jobs in JasmineGraph cluster. When there are job requests for multiple graphs

at the same time, then the JasmineGraph server has to maintain the load average threshold to maintain SLA for all the scheduled jobs at that moment. Therefore, using the lowest load average threshold out of the load average threshold values of all the graphs with job requests will maintain SLA for all jobs.

Similarly other cluster thresholds have to be derived and then GQSM will use the respective load average values when processing job requests. The experiment results which were carried out based on these load average thresholds will be analyzed in detail in the next section.

5.2 Model Evaluation

One of the primary objectives of this research is to develop the GQSM which will schedule the job requests with observation-based admission control. There are multiple aspects to be evaluated in this model. Those aspects are discussed in detail below.

First evaluation is to check the parameters which are used in GQSM model are valid to build a concrete GQSM model. Next two sections evaluate the GQSM in a single server cluster and in a distributed cluster. Load average curve prediction will be evaluated later and then the auto calibration process will be analyzed in detail.

5.2.1 GQSM Parameter Evaluation

GQSM model primarily make decisions based on the load average data of the system. Therefore, before evaluating the developed model it is required to evaluate whether the load average information can be used to develop this model. During the evaluation we have to make sure that the similar load average data can be collected when sending the same job request multiple times for a selected graph. To evaluate this, the same request was sent to a selected cluster one after the other and collected load average information from the system. This was carried out for multiple iterations and checked the load average curves for each iteration. The load average curve drawn in each iteration was compared to the load average curve derived in the calibration phase.

Figure 5.9 illustrates the graphs drawn based on the collected load average data for Youtube graph in cluster 1 for 3 iterations.

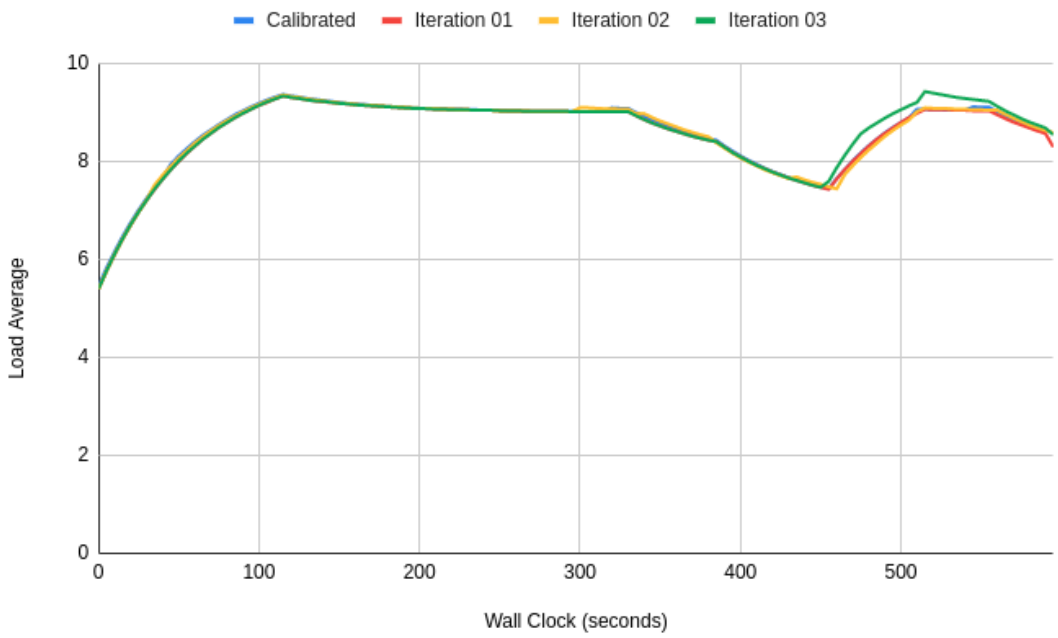


Figure 5.9: Load average curves for multiple iterations-cluster 1

Similarly for cluster 2 also the same experiment was carried out and the graphs were drawn for each host. Figure 5.10 below includes the graphs for that scenario.

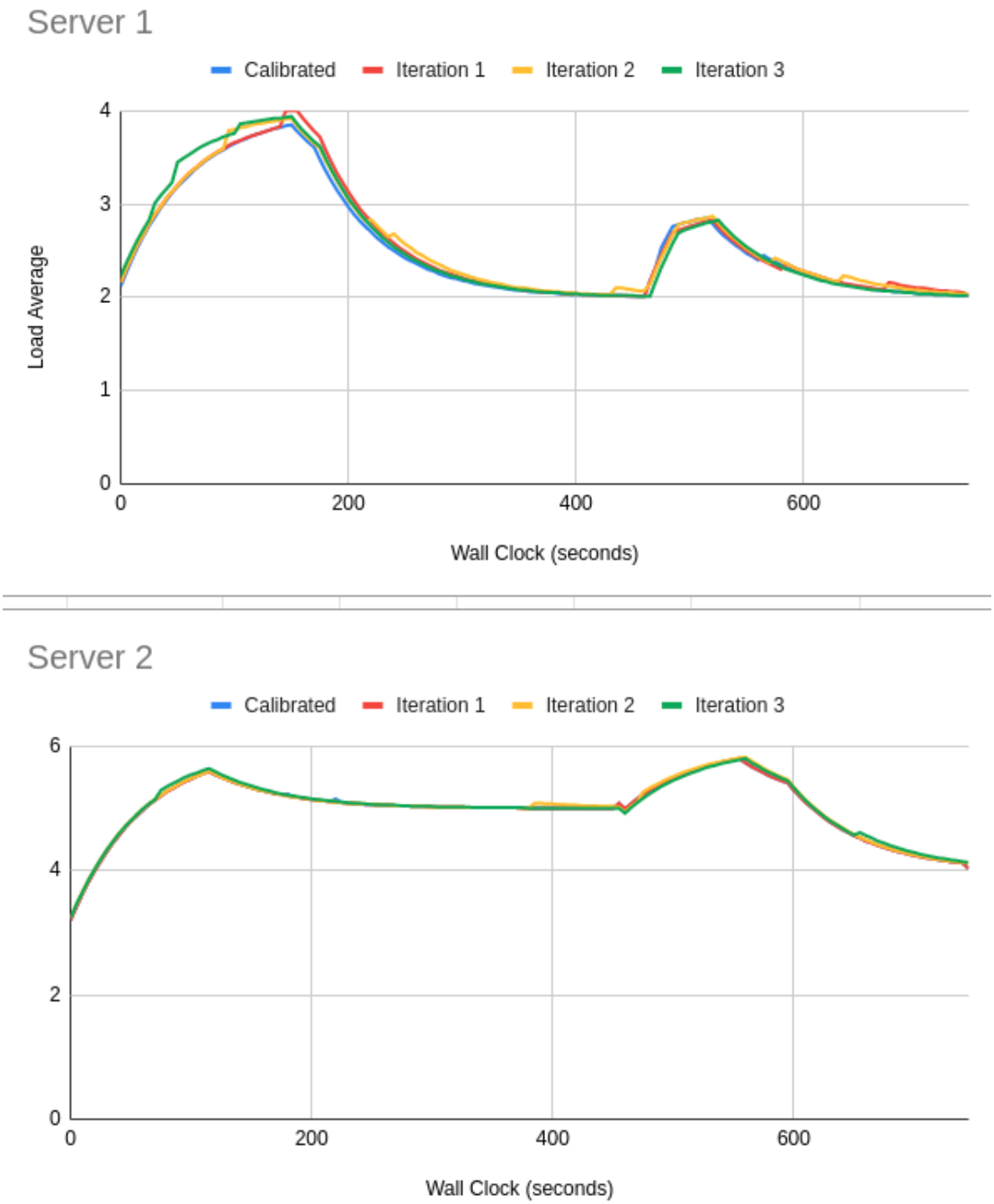


Figure 5.10: Load average curves for multiple iterations-cluster 2

Comparing the load average curve for each iteration in Figure 5.6 provides that they slightly vary from each other. As the variance is low it can be neglected and we can consider that the load average curves coincide with each other in each iteration. Therefore, the load average curve can be considered as a repetitive one for each iteration. If the curves are analyzed for each server in Figure 5.6 also the same behavior can be observed. Therefore, for a distributed environment also the load average curves coincide. This provides a concrete foundation to build the GQSM based on the load average data.

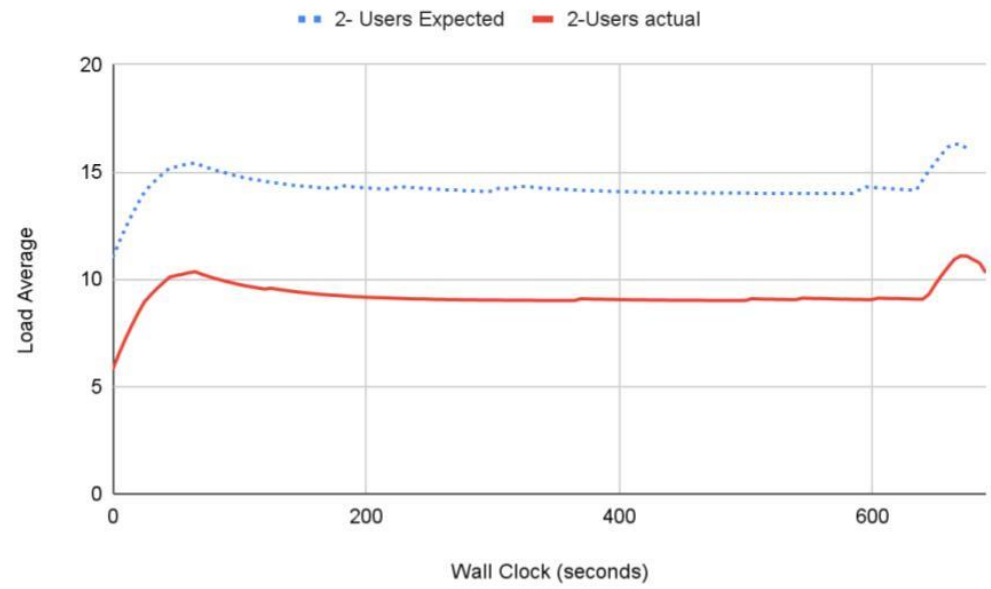
5.2.2 GQSM Evaluation in Single Server Cluster

The job scheduler built with GQSM was deployed in each JasmineGraph cluster. Then the behavior with GQSM can be observed by running below described experiments. The same graph dataset which was used during the calibration phase is used for evaluation as well because the purpose of the proposed model is to maintain the SLA for the calibrated graph datasets. Epinions graph which deployed in cluster 1 was used to analyze the results. During the calibration phase of cluster 1, the load average threshold for Epinions graph was calculated as 82 as per the Table 5.7. Guaranteed SLA for Epinions in cluster 1 is 719 seconds as per the Table 5.5.

Comparison between the predicted load average curve and the actual load average curve can be carried out now by sending different combinations of requests into the system. Figure 5.11 and Figure 5.12 below illustrate predicted and actual load average curves for 2 user,4 user,8 user and 16 user scenarios.

Appendix A includes the load average curve drawn using the load average data collected during calibration phase. When the number of concurrent requests for Epinions is increased the shape of the load average curve does not change. Also, the actual load average curve doesn't exceed the expected load average curve at any point. This preserves the system from not exceeding the predicted model at any given point which can be used as a solid prediction model for GQSM.

If we consider the 16-user case, the curve data is spread across a higher time window than the 1 user case. This is because the 16-user case has higher response time than the 1 user case. Due to that the load average curve data collected time is higher than 1 user case. Therefore, the graph is spread across a higher time window. The same behavior is applicable for any scenario with more than one user. But that is not visible in the graphs clearly.



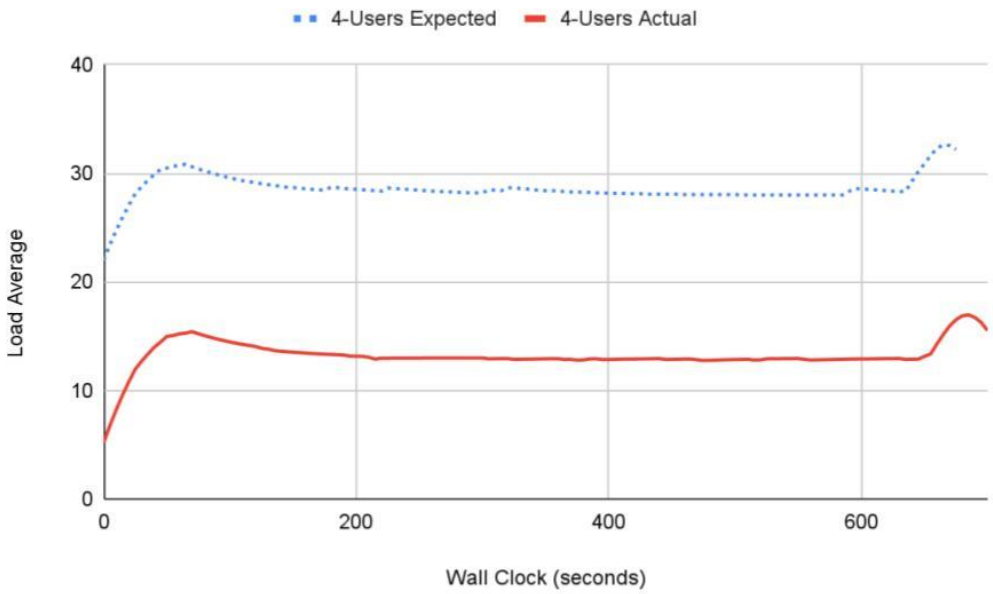


Figure 5.11: Predicted and actual work load curves for Epinions 2 users and 4 users

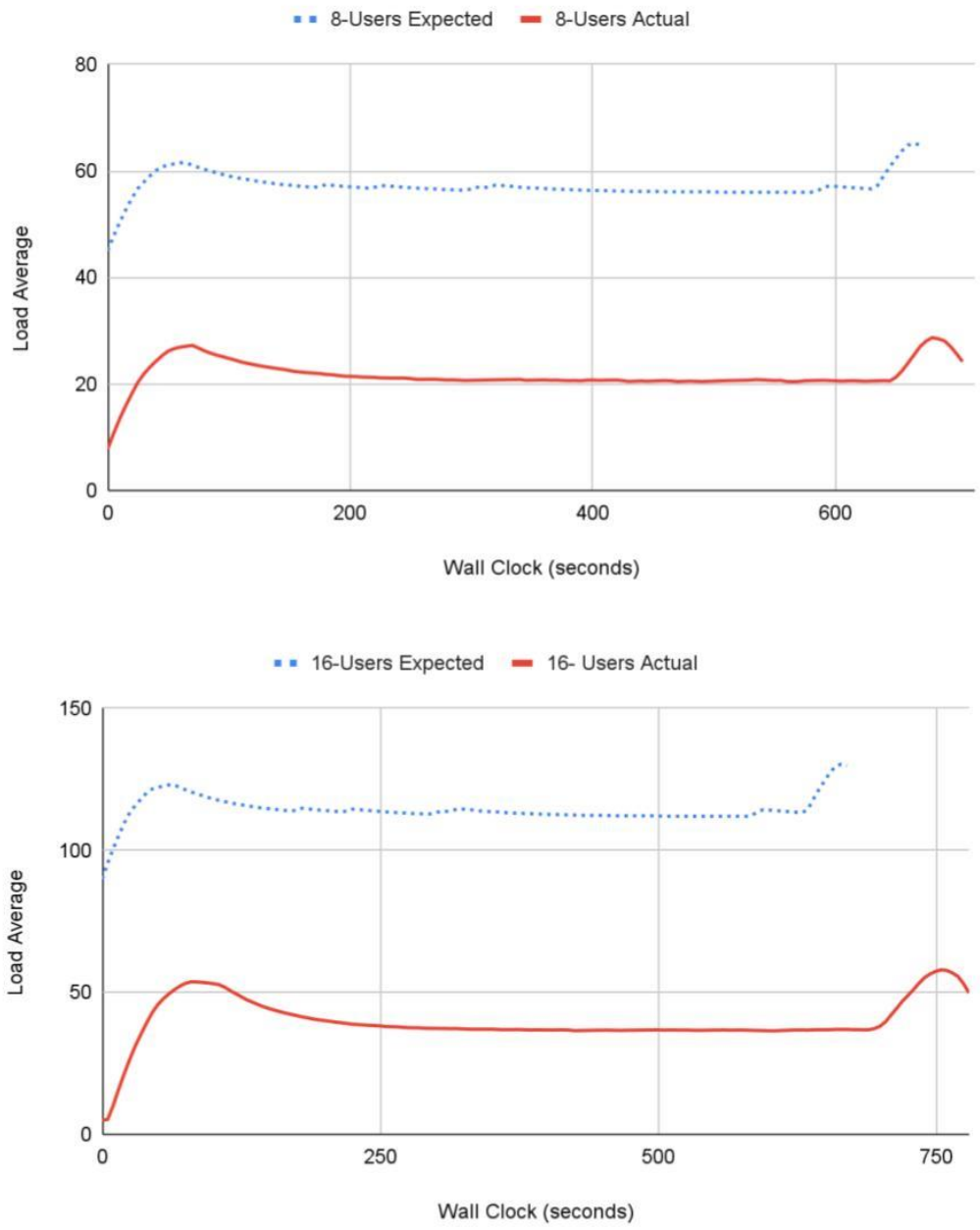


Figure 5.12: Predicted and actual work load curves for Epinions 8 users and 16 users

Table 5.8 below shows the average response times for each user count above and the maximum load average collected from the predicted load average curve for each scenario.

Table 5.8: Response times and maximum load average data for Epinion

Number of Users	Average Response time (seconds)	Maximum Load Average
2	689	16.3
4	698	32.6
8	700	65
16	769	130

Figure 5.13 below depicts the average response time information for each user count and the expected SLA response time.

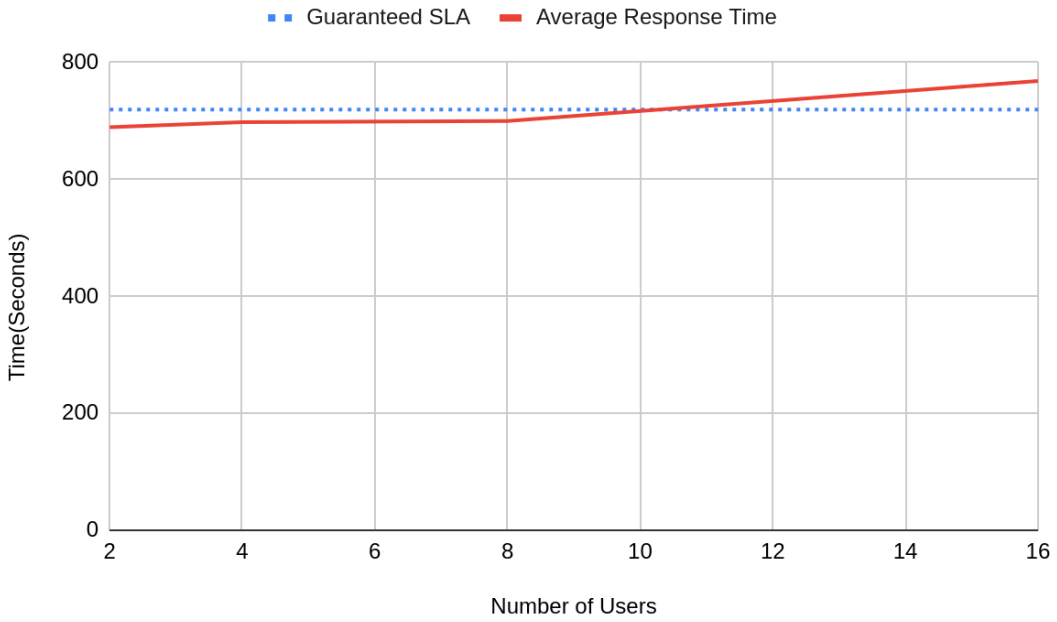


Figure 5.13: Average response time and guaranteed SLA information for Epinion

As Figure 5.13 illustrates for 16 user scenario the average response time exceeds the Guaranteed SLA of the system. Maximum load average for 16 users is 130 which is higher than the load average threshold for Epinions in cluster 1. But for 2 users, 4 users and 8 users the average response time is less than the guaranteed SLA and also the load average for those three user scenarios is less than the load average threshold for Epinions in cluster 1. Above analysis shows that the GQSM model implemented is working as expected and it is also scalable.

5.2.3 GQSM Evaluation in Distributed Cluster

It is interesting to analyze the behavior in a distributed environment. Cluster 2 is a distributed server environment with 2 servers. When multiple servers are participating in the cluster then the evaluation has to be carried out considering all the servers participating in the cluster.

Guaranteed SLA for Web Google graph in cluster 2 is 595847 milliseconds as per Table 5.5. The load average curves for the two servers based on the data collected during calibration phase are illustrated in the Figure 5.14

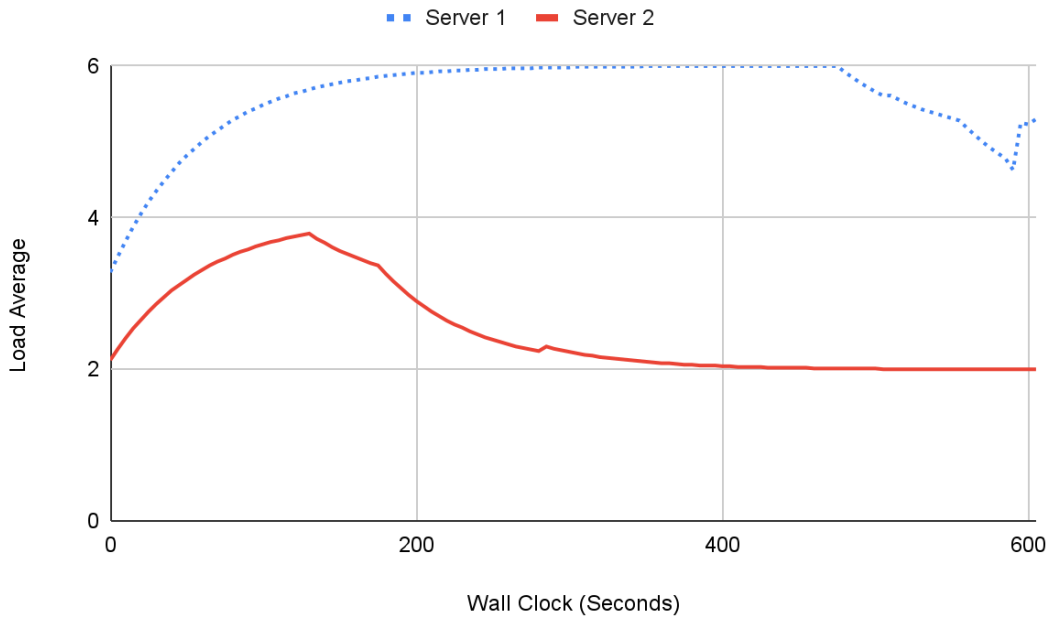


Figure 5.14: Calibrated load average curve for Web Google in cluster 2

Based on the load test carried out Table 5.9 lists the obtained load average thresholds for each of the servers in the cluster.

Table 5.9 Load average thresholds for Web Google in cluster 2

Server	Load Average Threshold
Server 1	78
Server 2	50

Multiple triangle count job requests were sent parallely to the JasmineGraph cluster deployed in cluster 2 to evaluate the load average curve predictability. Figure 5.15 and Figure 5.16 illustrates the predicted and actual load average curves for 4, 8, 12 and 16 users respectively in Server 1.

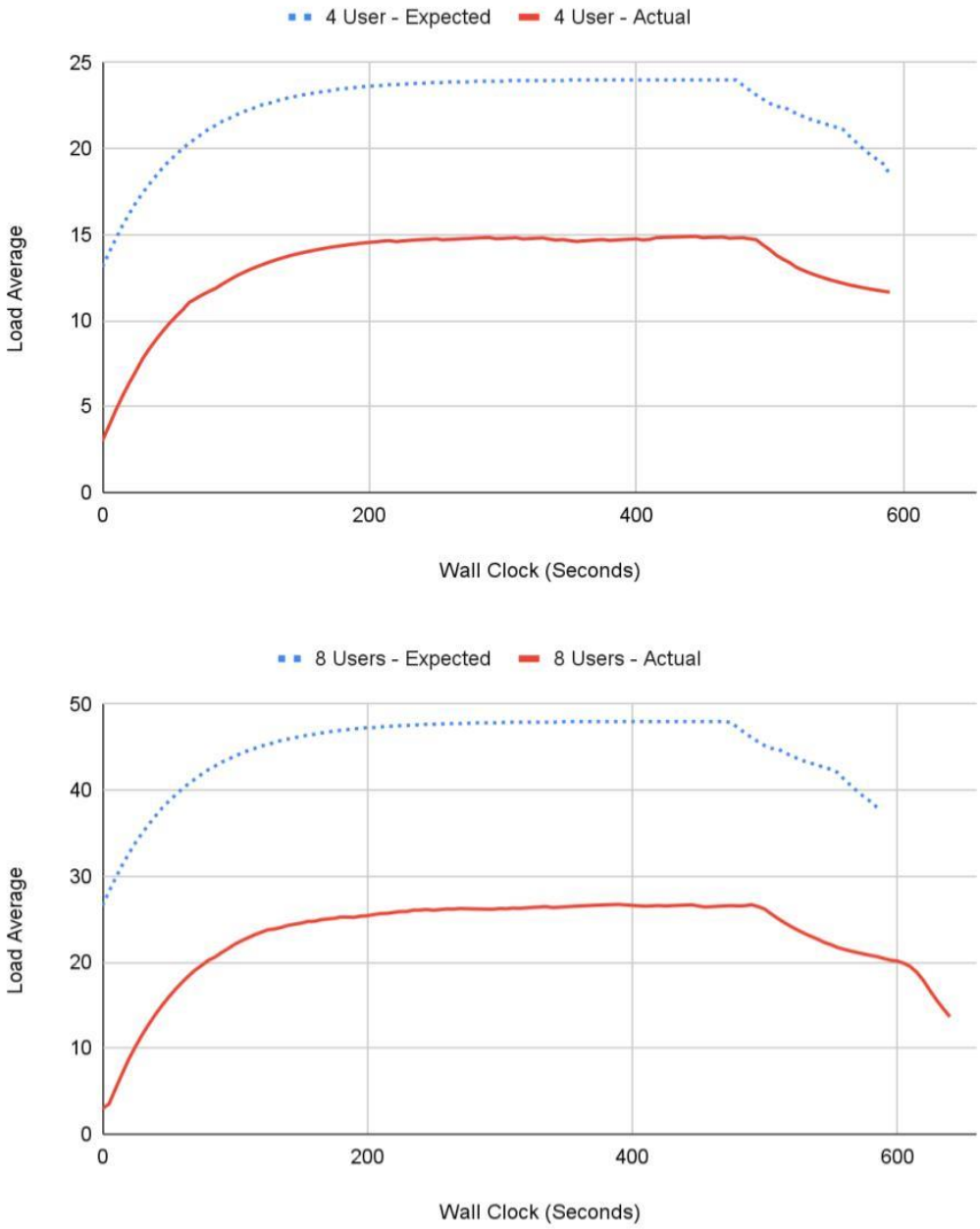


Figure 5.15: Predicted and actual workload curves for Web Google in cluster 2 server 1 for 4 users and 8 users

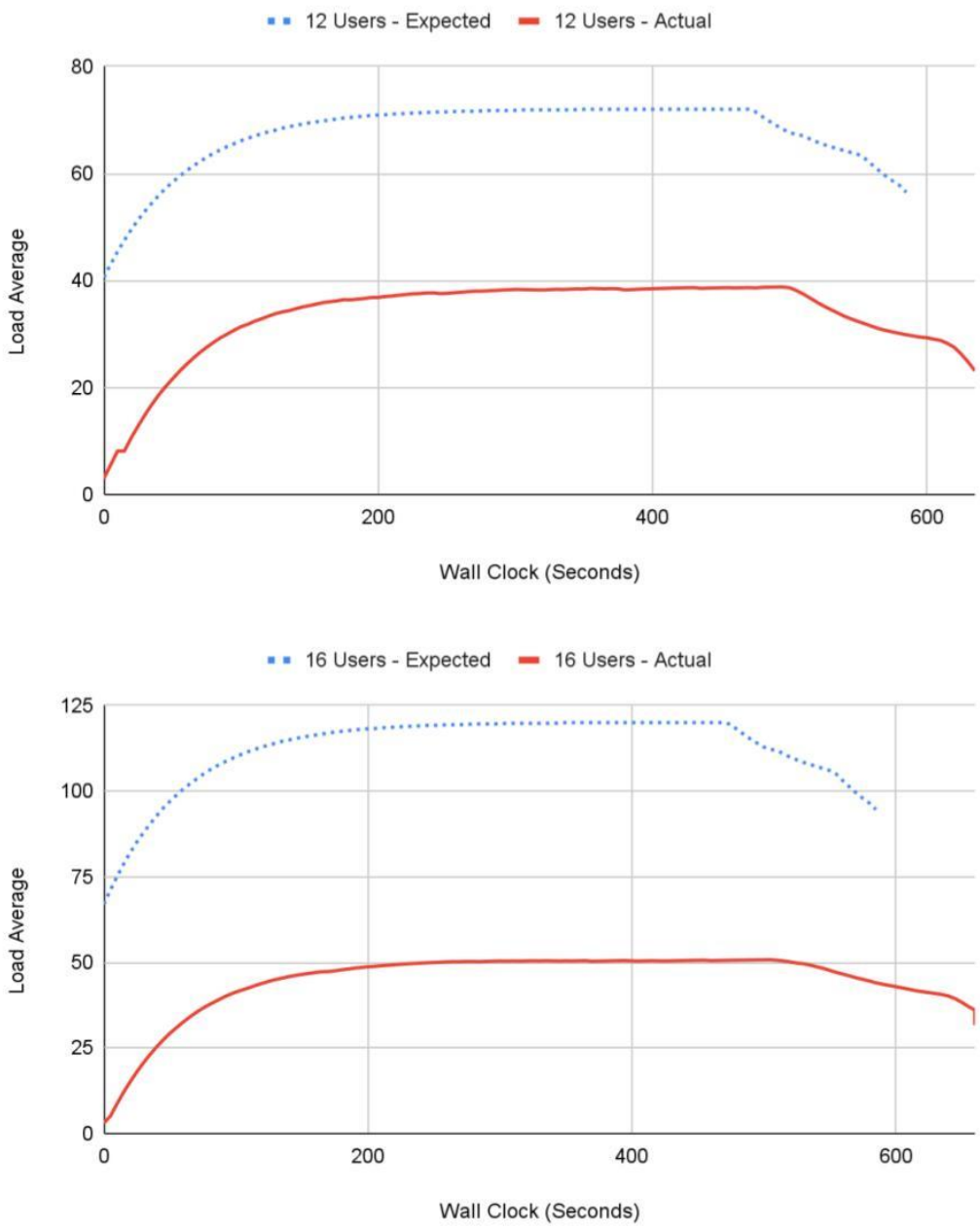


Figure 5.16: Predicted and actual workload curves for Web Google in cluster 2 server 1 for 12 users and 16 users

Similarly Figure 5.17 and Figure 5.18 illustrates the predicted and actual load average curves for 4, 8, 12 and 16 users respectively in Server 2.

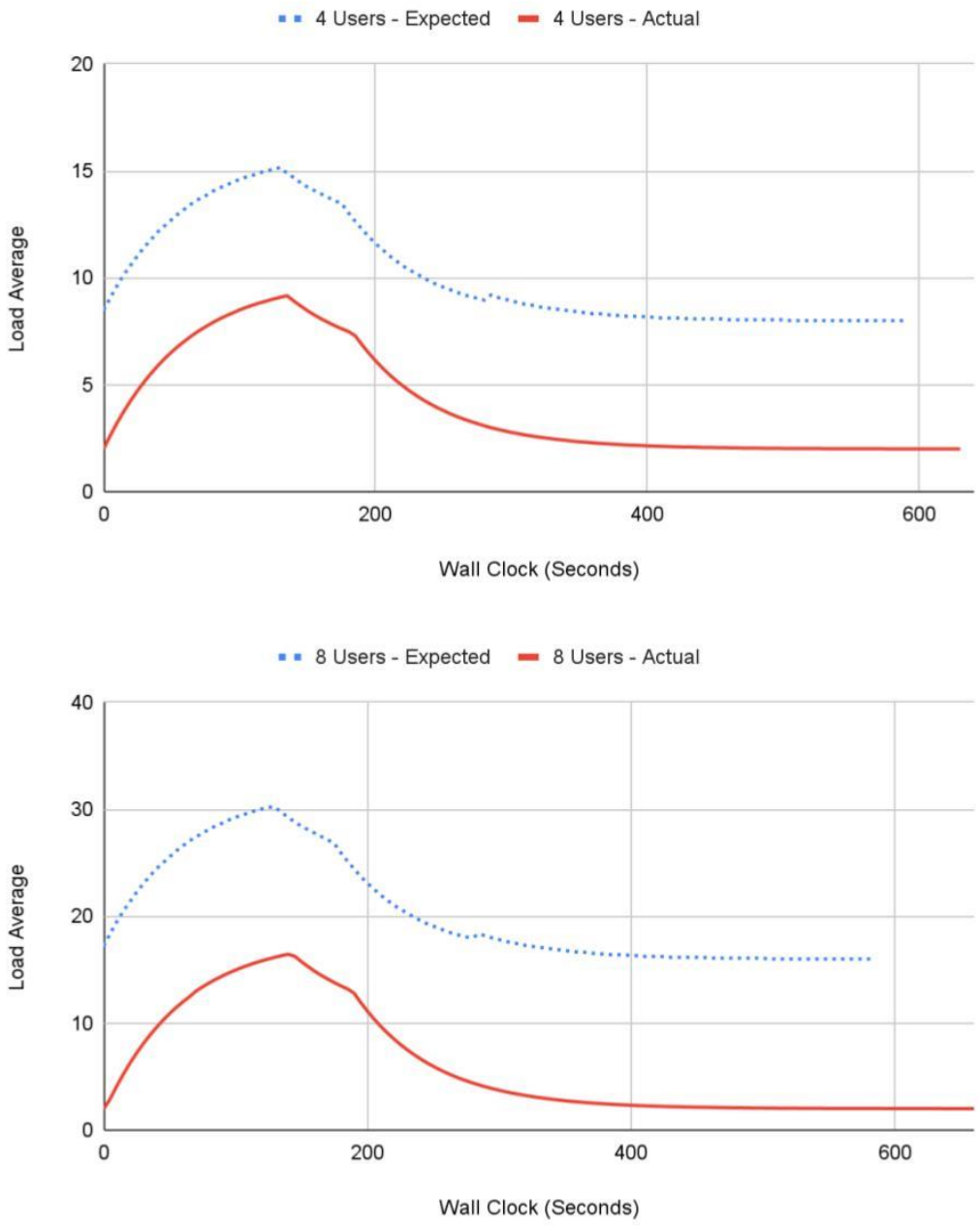


Figure 5.17: Predicted and actual workload curves for Web Google in cluster 2 server 2 for 4 users and 8 users

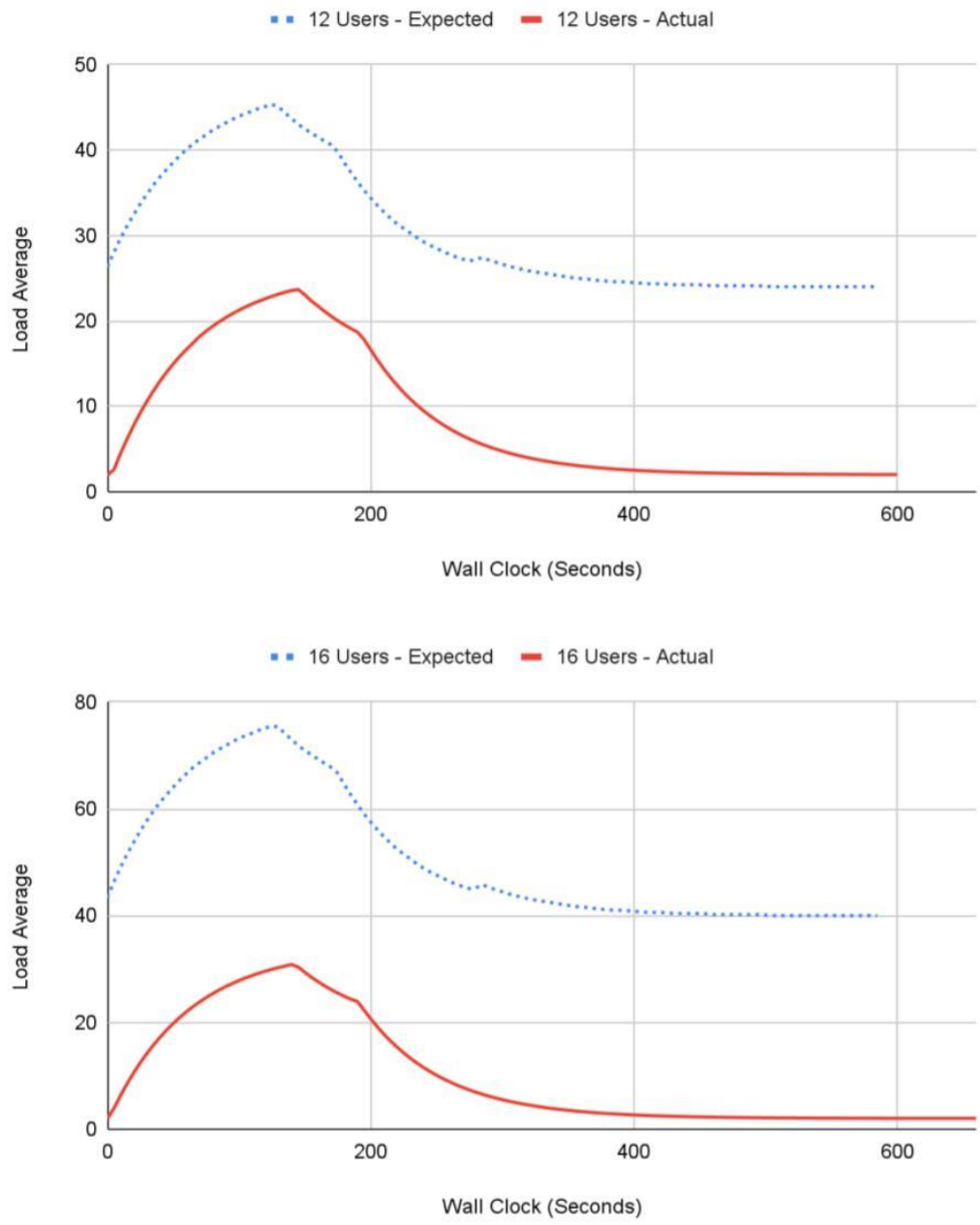


Figure 5.18: Predicted and actual workload curves for Web Google in cluster 2 server 2 for 8 users and 16 users

For a given server for different user loads the system preserves the same load average curve shape. In both the servers actual curve doesn't exceed the predicted curve for any user combination. Therefore, the load average data can be used for building GQSM in a distributed environment as well.

For each server Table 5.10 below shows the average response times for each user count above and the maximum load average collected from the predicted load average curve for each scenario.

Table 5.10: Average response times and maximum load averages for Web Google in cluster 2

Number of Users	Server 1		Server 2	
	Average Response time (seconds)	Maximum Load Average	Average Response time (seconds)	Maximum Load Average
4	603	24	603	15
8	609	48	609	30
12	616	72	616	45
16	678	120	678	76

Figure 5.19 below depicts the average response time information for each user count and the expected SLA response time for server 1.

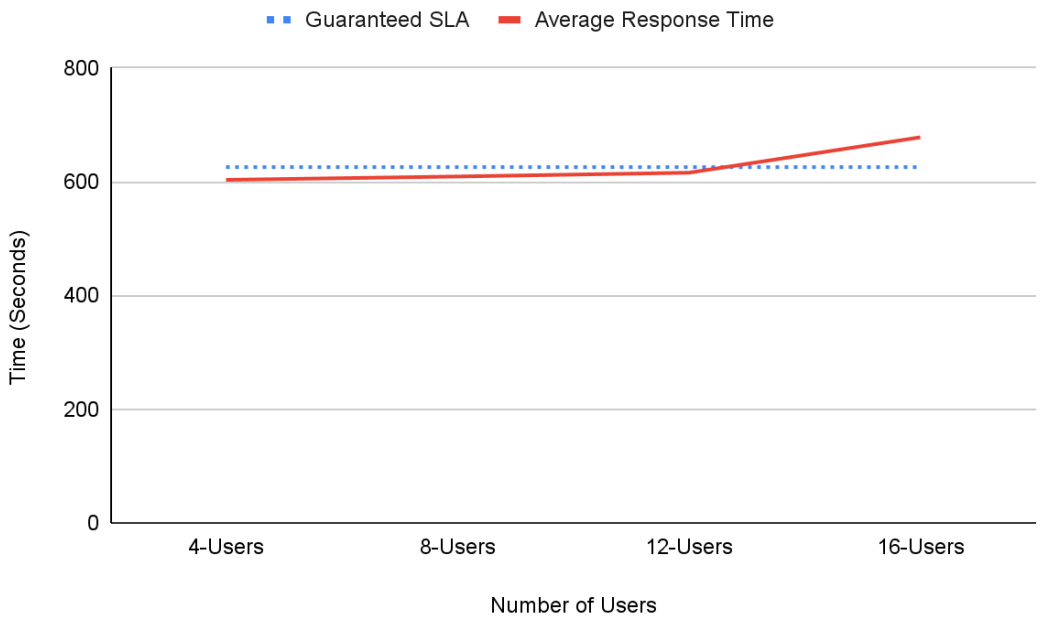


Figure 5.19: Average response time and guaranteed SLA information for Web Google in cluster 2 server 1

Above graph shows that the average response time doesn't exceed the guaranteed SLA value for 4 users, 8 users and 12 users scenarios in server 1. But for 16 users, the average response time exceeds the guaranteed SLA value. The maximum load average for 16 user scenario is 120 which exceeds the load average threshold for server 1 which is 78.

Figure 5.20 below depicts the average response time information for each user count and the expected SLA response time for server 2.

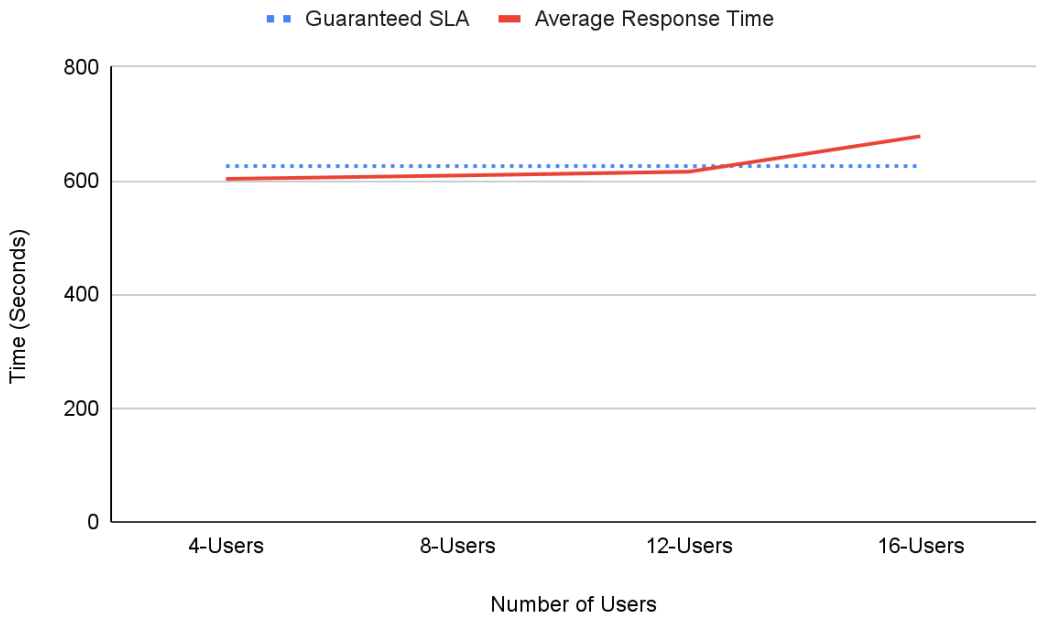


Figure 5.20: Average response time and guaranteed SLA information for Web Google in cluster 2 server 2

Above graph shows that the average response time doesn't exceed the guaranteed SLA value for 4 users, 8 users and 12 users scenarios in server 2. But for 16 users, the average response time exceeds the guaranteed SLA value. If the load average threshold is checked there is a violation in the load average threshold for server 1 for 16 user scenarios. The maximum load average for 16 users is 120 which exceeds the load average threshold for server 1 which is 50. This depicts that the GQSM model preserves the SLA if the total maximum load average for the job requests is less than the load average threshold for the respective server.

Both Server 1 and Server 2 in cluster 2 preserve the expected behavior of the GQSM model. Therefore, the GQSM is scalable to distributed remote clusters as well.

5.2.4 Load Average Curve Data Prediction

Comparing the derived curve for multiple users against the actual curve for the same number of users shows that the actual curve can be predicted using the derived curve. As discussed previously the shape of the predicted curve and the actual curve is similar. But there is an amplitude difference between those curves. A base amplitude adjustment curve can be used to adjust the amplitude difference between the predicted and actual curves. Base amplitude adjustment curve is unique for a selected graph for a selected cluster. Points in the base amplitude adjustment curve can be derived using the equation 2 below.

Base Amplitude adjustment curve load average at time T	=	2 Users derived curve load average at time T	_	2 Users actual curve load average at time T

…..……...(2)

Figure 5.15 below illustrates the base amplitude adjustment curve for epinion graph in cluster 1.

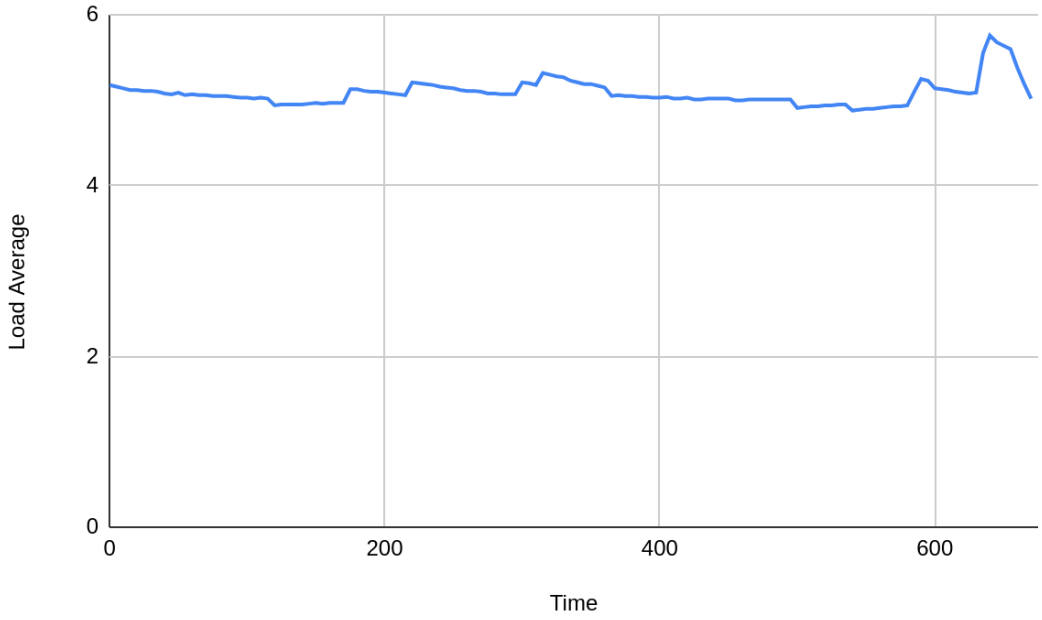


Figure 5.21: Base amplitude adjustment curve for epinion graph in cluster 1

Equation 3 below can be used to adjust the amplitude of the derived load average curve for n users.

Adjusted load average at time T	=	Load average of derived curve at time T	-	(n-1) * load average of base amplitude adjustment curve at time T	…(3)

Figure 5.22 below compares the derived curve based on the calibrated data, derived amplitude adjusted curve and the actual curve for epinions graph for 6 users in cluster 1 by applying the equation 3 above. The graph illustrates that the expected curve for 6 users coincides with the amplitude adjusted curve which is derived based on the calibrated load average data. Therefore, equation 3 can be used to predict the actual load average curve based on the calibrated data.

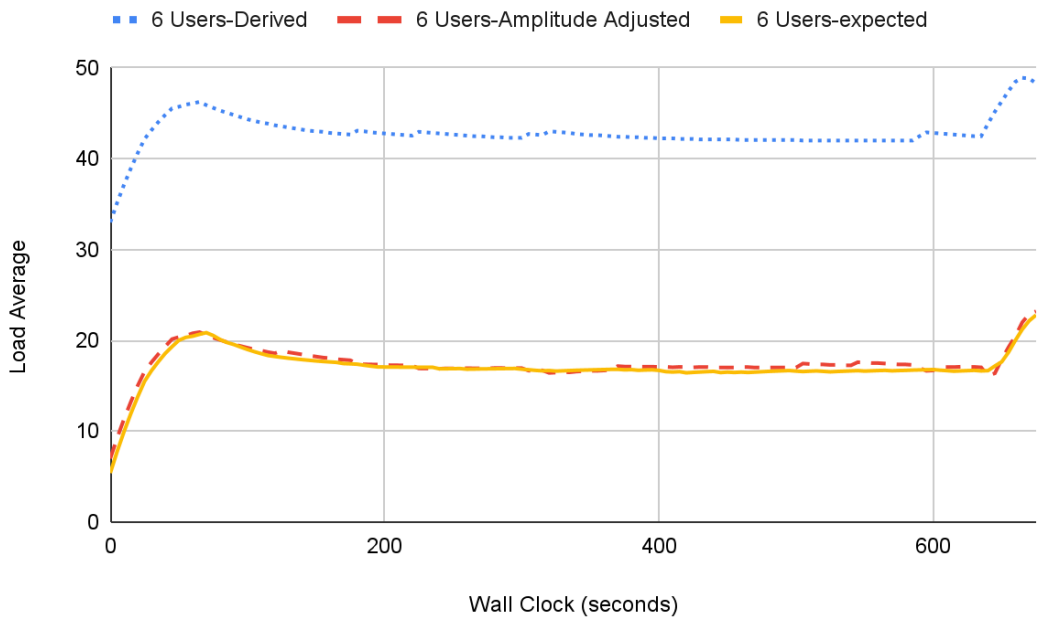


Figure 5.22: Comparison of derived curve, derived amplitude adjusted curve and expected curve for epinions graph for 6 users in cluster 1

5.2.5 GQSM Auto Calibration

Amplitude adjusted curve can be used to derive the load average curve for an uncalibrated graph. Figure 5.23 below illustrates the graphs obtained during the auto calibration process of the epinion graph in cluster 1. The Youtube graph is taken as the calibrated graph in this example and the epinions graph is the uncalibrated graph.

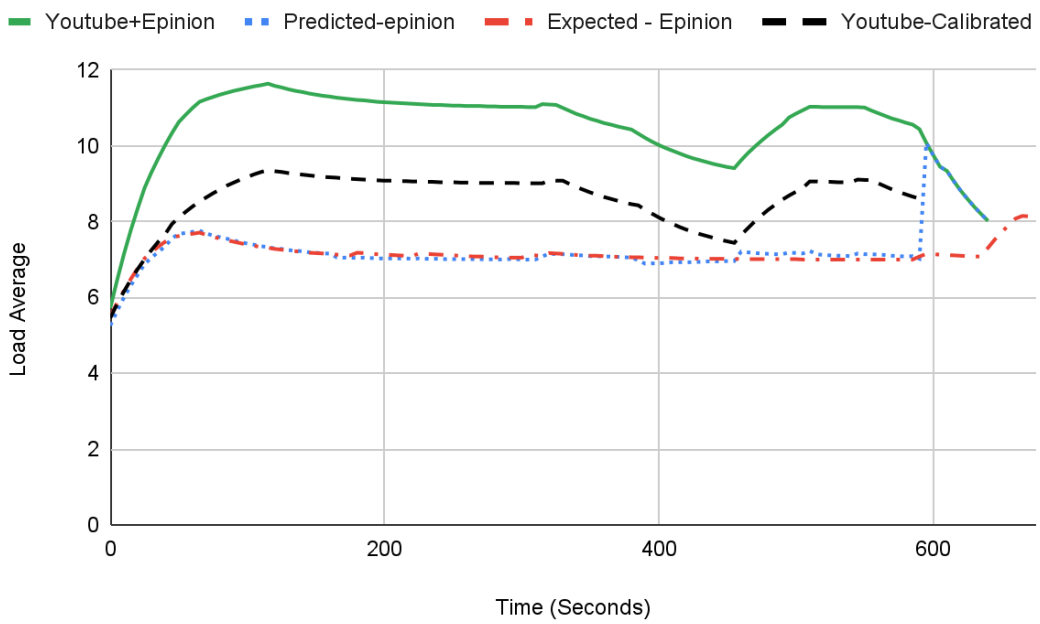


Figure 5.23: Obtained graphs during auto calibration process of Epinion in cluster 1

In the above example one youtube job request is executed along with one epinion job request. Two jobs were run in parallel and the load average data for both of the graphs were collected. . Load average curve data was available for the youtube graph and then the difference between the collected load average data for both graphs and load average data for youtube graph returns the load average curve data for epinion graphs. Expected graph for epinion is displayed in Expected-Epinion graph and as observed the derived epinion graph has slight variations compared to the expected-epinion graph and the variation can be neglected.

After around 590 seconds there is a sudden spike in the predicted epinion. The reason is by this time the execution of the youtube graph is completed and there is no data available for calculating the difference. Therefore, the predicted-epinion graph converges with the load average curve for both graphs. After this convergence this predicted-epinion graph starts converging with the expected-epinion graph because the only graph running in the system at that moment is the epinion graph.

When an uncalibrated graph runs with multiple calibrated graph job requests, a similar behavior can be observed. Figure 5.24 illustrates the behavior when 1 epinion uncalibrated graph runs with two youtube calibrated graphs.

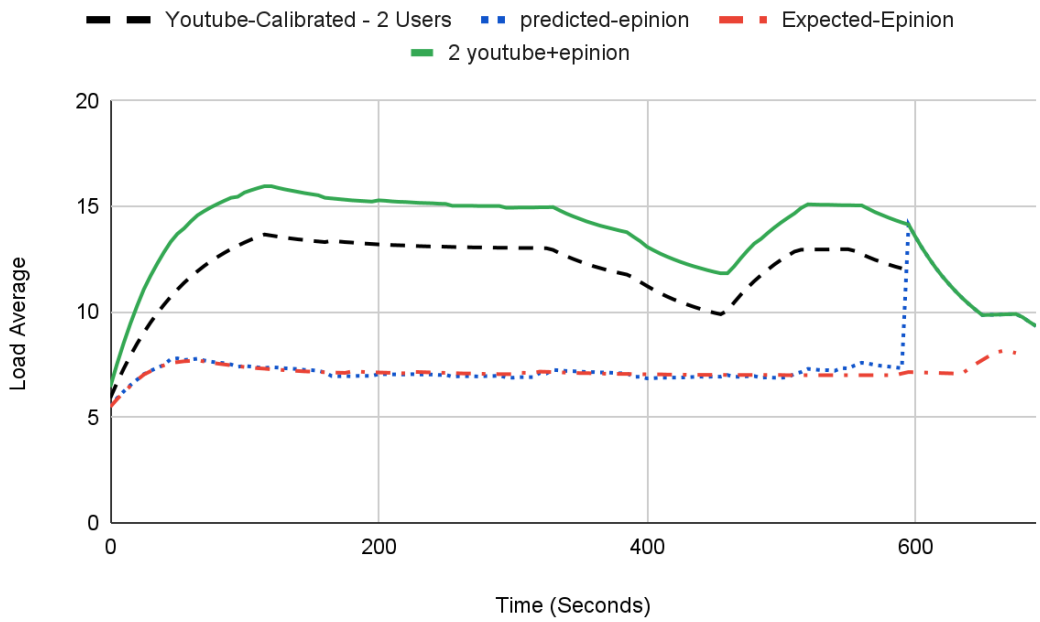


Figure 5.24: Obtained graphs during auto calibration process of Epinion in cluster 1 with two youtube calibrated graphs

Load average curve derivation for an uncalibrated graph provides a graph which coincides with the expected graph for the execution time period of the calibrated graph. Once the calibrated graph execution completes then the derived graph deviates from the expected one. After that sudden spike eventually, the derived graph converges with the expected graph.

Above analysis shows that load average curve derivation for an uncalibrated graph can be carried out while running the usual triangle count jobs in the system. Therefore, the system doesn't need to go through a calibration phase for each and every graph and the system is scalable to uncalibrated graphs as well.

5.3 Results Discussion

After uploading the graph datasets into the JasmineGraph cluster, the system underwent a calibration phase. During this calibration phase the triangle count jobs for the uploaded graphs were scheduled one by one. The job execution time was captured and it was used to derive the SLA for the graph. During the calibration the load average data was also captured periodically. Then the system was subjected to a load test to identify the maximum load average each host can allow without violating the SLA for a given graph.

GQSM model derives the aggregate load average curve for all the scheduled graphs. At a given time this aggregate load average curve data is always higher than the actual load average data for that environment. If the actual values exceed the predicted values, then the GQSM model could violate the SLA values because it cannot be properly predicted. But actual values always lower than the predicted values means the GQSM model has sufficient margin to make decisions. This shows that the evaluations carried out by the GQSM model are always on the safer side.

GQSM model uses calibrated data in the JasmineGraph server to analyze the state and to make decisions. GQSM model derives the aggregate workload curve for all the scheduled high priority jobs as part of its decision-making process. It was observed that the shape of the aggregate workload curve is similar to the shape of the actual curve. But there is an amplitude difference between the derived aggregate curve and the actual curve. It was released that the actual curve can be derived based on the derived aggregate curve and the base amplitude adjustment curve.

It was realized that the GQSM model preserves the SLA values when the system doesn't exceed the load average values suggested by the GQSM model. If the system exceeds the load average values suggested by GQSM, then the system fails to maintain the SLA values. The same behavior is applicable for distributed clusters as well. But in a distributed cluster the GQSM model does the evaluation for all the hosts participating in the cluster. If at least one host violates the SLA threshold limits, the whole job gets affected.

Auto calibration process is used to calibrate graphs which are uncalibrated. The auto calibration begins with the job execution of the respective graph. It was observed that the auto calibration process which is used in JasmineGraph server is capable of providing a more realistic load average curve as well as the SLA values for the respective graph.

In a production environment the JasmineGraph server always maintains the SLA for scheduled high priority jobs. The server accepts all high priority jobs until it exceeds the load average threshold for that server. Then the server keeps rejecting the new high priority jobs until the load average of the server is higher than the respective threshold value. But the server accepts all the low priority jobs. All the low priority jobs get onhold till the execution of all high priority jobs complete. This is because the server doesn't guarantee any SLA for low priority jobs. Upon completion of high priority job execution, the server starts scheduling low priority jobs.

CHAPTER 6: CONCLUSION

This chapter first focuses on the summary of the proposed GQSM model and the experiment results. Then it describes the limitations of this research and finally suggests the future work to extend this work.

6.1 Summary

Current graph database servers do not consider the maintenance of SLA when executing graph queries. Having SLA enforcement for graph database servers is important for them to be used as online transaction processing systems (OLTP). This research proposed a model for maintaining service level agreements for triangle counting via observation-based admission control. It focuses on modeling a job scheduler which uses Graph Query Scheduling Mechanism (GQSM) for scheduling the jobs.

The system first goes through a calibration phase where the graphs which need to maintain the SLA levels are being calibrated in a selected cluster environment. During the calibration phase the triangle count algorithm is executed against the graph. While executing the triangle count algorithm the load average information of the system is collected and stored as the base load average curve for the respective graph for that cluster. With the completion of the triangle counting algorithm the system collects the response time and derives an SLA value based on the response time for the respective graph. After this initial calibration system is subjected to a load test with uniform job requests. The objective of this load test is to identify the load average threshold value the system can handle such that it doesn't exceed the derived SLA value. All the graphs which need to maintain the SLA have to undergo this calibration phase.

GQSM is modeled using the queueing theory where all the job requests are first received to the job queue in the system. Job scheduler analyzes the job request and categorizes them into high priority and low priority jobs. SLA is maintained only for high priority jobs. All the low priority jobs are scheduled for execution if there are no high priority jobs scheduled in the system. If there are high priority jobs scheduled then the scheduler will hold those job requests till execution of all high priority jobs completes.

As SLA has to be maintained for high priority job requests, the job scheduler derives the aggregated load average curve imagining the job is scheduled and checks whether the maximum load average of the aggregated curve exceeds the load average threshold of that graph. If the maximum load average of the aggregated load average curve doesn't exceed the load average threshold of the graph, then the job is accepted for execution. If it exceeds then the scheduler checks an appropriate time to accept the job

without SLA violation. If the scheduler finds a valid time, then the job is accepted for future execution. If it cannot find a proper time the job gets rejected from the system.

The results indicate that the load average of the system can be used to make decisions related to SLA violations. For example, if the load average values exceed 82 in cluster 1, then the SLA gets violated for Epinions graph job requests. Similarly, if the aggregated load average value exceeds 78 in server 01 of cluster 2, then the SLA gets violated for Web google graph. Therefore, the proposed model is scalable to multiple users. The model is applicable for distributed environments as well.

6.2 Limitations

The main objective of this research is to develop a model which guarantees SLA for the job request. GQSM model which was introduced in the previous chapters has the capability to guarantee the job requests it receives. Triangle count algorithm was used to evaluate the GQSM model and based on the experiment results we think that GQSM can work as a generic model for other algorithms as well.

First few graph datasets have to go through a calibration phase. This takes a considerable amount of time. Therefore, the initial setup of the cluster takes a considerable time before the system is capable of guaranteeing SLA for job requests. Taking a longer time for calibration can be a concern from users' perspective. Hence it is a constraint for the users.

According to current implementation auto calibration will be conducted when there are scheduled jobs for one uncalibrated graph with one or more calibrated graphs for the same graph. The server doesn't initiate calibration when the list of calibrated job requests is for various graph datasets. In production grade systems there are job requests for various graphs.

6.3 Future Work

The main objective of this project is to build the GQSM model to guarantee SLA for triangle counting jobs via observation-based admission control.

It is interesting to extend the proposed model such that the system can be set up with minimal effort. At the moment it consumes a considerable time to do initial setup due to the calibration phase. Graph dataset properties-based SLA prediction is one approach which is interesting for trying out.

Production grade applications handle various job requests at once. At the moment the proposed auto calibration model works only for multiple job requests for the same graph. Therefore, it is interesting to extend the model such that the server is capable of calibrating a graph along with job requests for various calibrated graphs.

Proposed model uses the load average of the hosts as the main factor to make decisions on the SLA management. There are multiple hosts participating in a graph database cluster. Workload of each host is not uniformly distributed. At a given time there can be hosts which have less workload average than the others. The model can be extended to intelligent workload distribution. This will lead to having more evenly distributed workloads among the hosts participating in the cluster. Eventually the underlying compute resources will be efficiently managed and the SLA management can also be improved along with this extension.

GQSM model proposed in this research project executes jobs only using the CPU. Modern GPUs are highly resourceful and they can be used efficiently in highly parallel tasks. In the proposed GQSM model there are multiple parallely executing tasks such as collecting load average information periodically, logging load average information periodically and scheduling job requests in the server. It is interesting to enhance the parallel tasks of the proposed GQSM model with GPU and analyze the results.

Proposed GQSM model doesn't consider the network factor for evaluation. It is interesting to analyze the behavior when the distributed servers are located in different geo locations. The network factor is the key factor to be considered here. Enhancing the GQSM model to work in a slow network is another interesting area to study.

Proposed GQSM model accepts jobs based on the order the jobs are queued in the job request queue. Therefore, in some instances the jobs which take less time to execute can get rejected because the long running jobs are already scheduled. The system can be improved for smart job scheduling where the GQSM model make decisions based on the calibrated SLA values for the graphs as well.

References

[1] C. Mayer, M. A. Tariq, R. Mayer, and K. Rothermel, “GrapH : Traffic-Aware Graph Processing,” vol. 29, no. 6, pp. 1289–1302, 2018.

[2] M. Dayarathna *et al*., "Acacia-RDF: An X10-Based Scalable Distributed RDF Graph Database Engine," *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, San Francisco, CA, 2016, pp. 521-528. doi: 10.1109/CLOUD.2016.0075

[3] M. Dayarathna et al., "An X10-Based Distributed Streaming Graph Database Engine," 2017 IEEE 24th International Conference on High Performance Computing (HiPC), Jaipur, 2017, pp. 243-252. doi: 10.1109/HiPC.2017.00036

[4] M. Dayarathna and T. Suzumura, "Towards Scalable Distributed Graph Database Engine for Hybrid Clouds," 2014 5th International Workshop on Data-Intensive Computing in the Clouds, New Orleans, LA, 2014, pp. 1-8. doi: 10.1109/DataCloud.2014.9

[5] Aurelius (2018), Titan: Distributed Graph Database, http://titan.thinkaurelius.com/

[6] "Oracle Labs PGX: Parallel Graph AnalytiX Overview", *Oracle.com*, 2018. [Online]. Available: https://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytix/overview/index.html. [Accessed: 24- Oct- 2018].

[7] "The Neo4j Graph Platform", *Neo4j Graph Database Platform*, 2018. [Online]. Available: https://neo4j.com/. [Accessed: 24- Oct- 2018].

[8] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: a distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (SIGMOD '13). ACM, New York, NY, USA, 505-516. DOI: http://dx.doi.org/10.1145/2463676.2467799

[9] "GraphChi/graphchiDB-scala", *GitHub*, 2018. [Online]. Available: https://github.com/GraphChi/graphchiDB-scala. [Accessed: 24- Oct- 2018].

[10] "AllegroGraph", *Allegrograph.com*, 2018. [Online]. Available: https://allegrograph.com/. [Accessed: 24- Oct- 2018].

[11] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput., 20(1):359–392, Dec. 1998.

[12] S. Ravindra, M. Dayarathna, and S. Jayasena, “Latency Aware Elastic Switching-based Stream Processing Over Compressed Data Streams,” Proc. 8th ACM/SPEC Int. Conf. Perform. Eng. - ICPE ’17, pp. 91–102, 2017.

[13] WSO2. Wso2 Stream Processor. URL:https://docs.wso2.com/display/SP400/Stream+Processor+Documentation, 2018.

[14] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch. Balancing load in stream processing with the cloud. In Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on, pages 16–21, April 2011.

[15] W. Hummer, B. Satzger, and S. Dustdar. Elastic stream processing in the cloud. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 3(5):333–345, 2013.

[16] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann. Stormy: An elastic and highly available streaming service in the cloud. In Proceedings of the 2012 Joint EDBT/ICDT Workshops, EDBT-ICDT ’12, pages 55–60, New York, NY, USA, 2012. ACM.

[17] R. Li, Q. Zheng, X. Li and J. Wu, "A Novel Multi-objective Optimization Scheme for Rebalancing Virtual Machine Placement," 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), San Francisco, CA, 2016, pp. 710-717. doi: 10.1109/CLOUD.2016.0099

[18] Rishan Chen, Mao Yang, Xuetian Weng, Byron Choi, Bingsheng He, and Xiaoming Li. 2012. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing* (SoCC '12). ACM, New York, NY, USA, Article 3, 13 pages. DOI=http://dx.doi.org/10.1145/2391229.2391232

[19] M. K, "Know more about the system load and its impact on the server performance", Site24x7.com, 2022. [Online]. Available: https://www.site24x7.com/blog/load-average-what-is-it-and-whats-the-best-load-average-for-your-linux-servers. [Accessed: 18- Feb- 2022].

[20] A. Karunarathna et al., "Scalable Graph Convolutional Network based Link Prediction on a Distributed Graph Database Server," 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), 2020, pp. 107-115, doi: 10.1109/CLOUD49709.2020.00028.

[21] "nmon for Linux | Main / HomePage", *Nmon.sourceforge.net*, 2022. [Online]. Available: http://nmon.sourceforge.net/pmwiki.php. [Accessed: 26- Feb- 2022].

[22] "man page getloadavg section 3", *Manpagez.com*, 2022. [Online]. Available: http://www.manpagez.com/man/3/getloadavg/. [Accessed: 27- Feb- 2022].

[23] "Apache JMeter - Apache JMeter™", *Jmeter.apache.org*, 2022. [Online]. Available: https://jmeter.apache.org/. [Accessed: 01- Mar- 2022].

[24] "SNAP: Network datasets: Youtube social network", *Snap.stanford.edu*, 2022. [Online]. Available: https://snap.stanford.edu/data/com-Youtube.html. [Accessed: 02- Mar- 2022].

[25] "SNAP: Network datasets: Epinions social network", *Snap.stanford.edu*, 2022. [Online]. Available: https://snap.stanford.edu/data/soc-Epinions1.html. [Accessed: 02- Mar- 2022].

[26] "SNAP: Network datasets: Google web graph", *Snap.stanford.edu*, 2022. [Online]. Available: https://snap.stanford.edu/data/web-Google.html. [Accessed: 02- Mar- 2022].

[27] "SNAP: Network datasets: Astro Physics collaboration network", *Snap.stanford.edu*, 2022. [Online]. Available: https://snap.stanford.edu/data/ca-AstroPh.html. [Accessed: 02- Mar- 2022].

[28] S. Noel, D. Bodeau, and R. McQuaid, “Big-Data Graph Knowledge Bases for Cyber Resilience,” CEUR Workshop Proc., vol. 2040, no. 17, pp. 6–21, 2017.

[29] S. Noel, E. Harley, K. H. Tam, and G. Gyor, “Big-Data Architecture for Cyber Attack Graphs,” no.14–3549, pp. 1–6, 2016.

[30] Andreas Wagner and David A. Fell, *The small world inside large metabolic networks*, Proceedings of the Royal Society of London. Series B: Biological Sciences, http://doi.org/10.1098/rspb.2001.1711

[31] Jeong, Hawoong, Bálint Tombor, Réka Albert, Zoltan N. Oltvai, and A-L. Barabási. "The large-scale organization of metabolic networks." Nature 407, no. 6804 (2000): 651.

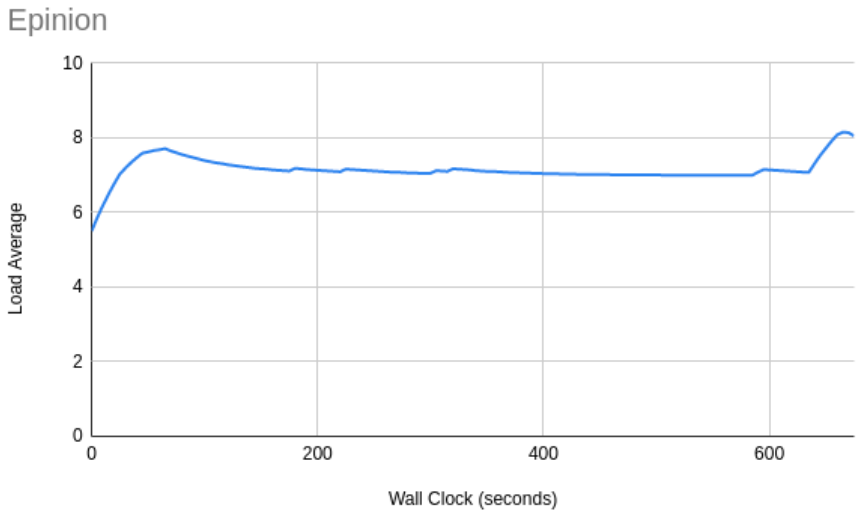
[32] Moore, Cristopher, and Mark EJ Newman. "Epidemics and percolation in small-world networks." *Physical Review E* 61, no. 5 (2000): 5678.

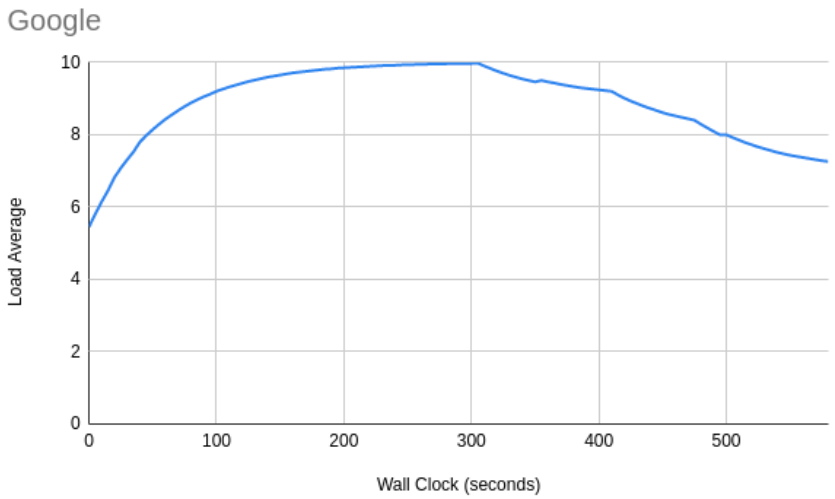
[33] McGovern, Amy, Lisa Friedland, Michael Hay, Brian Gallagher, Andrew Fast, Jennifer Neville, and David Jensen. "Exploiting relational structure to understand

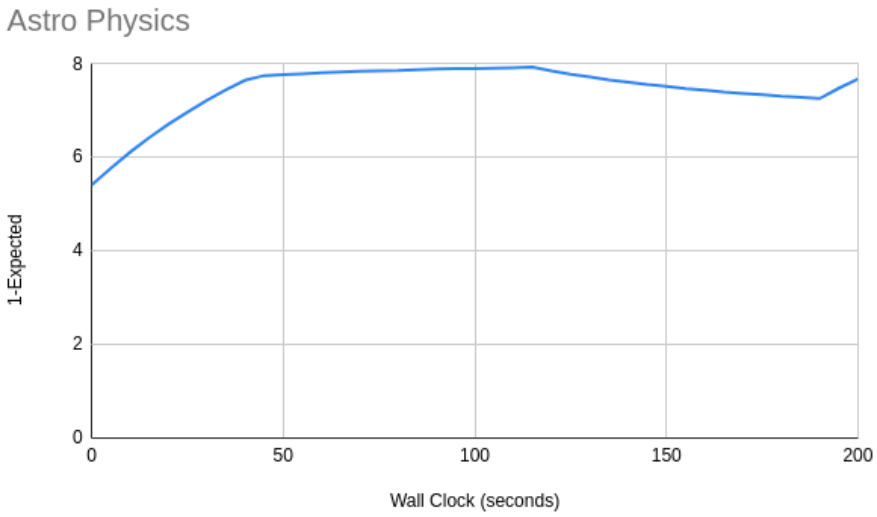
publication patterns in high-energy physics." ACM SIGKDD Explorations Newsletter 5, no. 2 (2003): 165-172.

[34] "Home - Docker", *Docker*, 2022. [Online]. Available: https://www.docker.com/. [Accessed: 21- Mar- 2022].

Appendix - A: Load Average Curves for Epinion, Google and Astro-Physics Graphs for Cluster 1







Appendix - B: Job Scheduler Implementation

```
while (true) {
    if (jobQueue.size() > 0) {
        jobScheduler_Logger.log("##JOB SCHEDULER## Jobs Available for "
                                "Scheduling", "info");

        std::vector<JobRequest> pendingHPJobList;
        std::vector<std::string> highPriorityGraphList;

        while (!jobQueue.empty()) {
            JobRequest request = jobQueue.top();

            if (request.getPriority() ==
            Conts::HIGH_PRIORITY_DEFAULT_VALUE &&
                    request.getJobType() == TRIANGLES) {
                pendingHPJobList.push_back(request);
                highPriorityGraphList.push_back(
                        request.getParameter(Conts::PARAM_KEYS::GRAPH_ID));
                jobQueue.pop();
            } else {
                jobQueue.pop();
                JobScheduler::processJob(request,
                        refToScheduler->sqlite, refToScheduler->perfSqlite);
            }
        }


        if (pendingHPJobList.size() > 0) {
            std::string masterIP = pendingHPJobList[0].getMasterIP();
            std::string jobType = pendingHPJobList[0].getJobType();
            std::string category = pendingHPJobList[0].getParameter(
                    Conts::PARAM_KEYS::CATEGORY);

            std::vector<long> scheduleTimeVector =
                    performanceUtil.getResourceAvailableTime
                    (highPriorityGraphList,jobType, category,
                            masterIP, pendingHPJobList);

            for (int index = 0; index != pendingHPJobList.size();
                                    ++index) {
                JobRequest hpRequest = pendingHPJobList[index];
                long queueTime = scheduleTimeVector[index];

                if (queueTime  < 0) {
                    JobResponse failedJobResponse;
                    failedJobResponse.setJobId(hpRequest.getJobId());
                    failedJobResponse.addParameter(
                            Conts::PARAM_KEYS::ERROR_MESSAGE,
                            "Rejecting the job request because "
                            "SLA cannot be maintained");
                    responseVectorMutex.lock();
                    responseMap[hpRequest.getJobId()] = failedJobResponse;
                    responseVectorMutex.unlock();
                    continue;
                }

                hpRequest.addParameter(Conts::PARAM_KEYS::QUEUE_TIME,
                        std::to_string(queueTime));
                JobScheduler::processJob(hpRequest, refToScheduler->sqlite,
                        refToScheduler->perfSqlite);
            }
        }
    } else {
        std::this_thread::sleep_for(std::chrono::seconds(2));
    }
}
```

Appendix - C: Performance Statistics Collection

```
int elapsedTime = 0;
time_t start;
time_t end;
PerformanceUtil performanceUtil;
performanceUtil.init();
Utils utils;

std::vector<Place> placeList = performanceUtil.getHostReporterList();
std::string slaCategoryId = performanceUtil.getSLACategoryId(command,category);

std::vector<Place>::iterator placeListIterator;

for (placeListIterator = placeList.begin();
        placeListIterator != placeList.end(); ++placeListIterator) {
    Place place = *placeListIterator;
    std::string host;

    std::string ip = place.ip;
    std::string user = place.user;
    std::string serverPort = place.serverPort;
    std::string isMaster = place.isMaster;
    std::string isHostReporter = place.isHostReporter;
    std::string hostId = place.hostId;
    std::string placeId = place.placeId;

    if (ip.find("localhost") != std::string::npos ||
                ip.compare(masterIP) == 0) {
        host = ip;
    } else {
        host = user + "@" + ip;
    }


    if (!(isMaster.find("true") != std::string::npos ||
        host == "localhost" || host.compare(masterIP) == 0)) {
        performanceUtil.initiateCollectingRemoteSLAResourceUtilization(host,
                atoi(serverPort.c_str()),isHostReporter,"false",
                placeId,elapsedTime, masterIP);
    }
}

start = time(0);

while(!workerResponded)
{

    if(time(0)-start== Conts::LOAD_AVG_COLLECTING_GAP)
    {
        elapsedTime += Conts::LOAD_AVG_COLLECTING_GAP*1000;
        performanceUtil.collectSLAResourceConsumption(placeList,
                graphId,masterIP,elapsedTime);
        start = start + Conts::LOAD_AVG_COLLECTING_GAP;
    }
}

performanceUtil.updateRemoteResourceConsumption(perDB,graphId,
        partitionCount,placeList,slaCategoryId,masterIP);
performanceUtil.updateResourceConsumption(perDB, graphId,
        partitionCount,placeList, slaCategoryId);

return 0;
```