Master's Thesis

Reducing Tail Latency In Cassandra Cluster Using Regression Based Replica Selection Algorithm

Chauque Euclides Teles Tomas
Program of Information Science and Engineering
Graduate School of Science and Technology
Nara Institute of Science and Technology

Supervisor: Professor Kazutoshi Fujikawa
Internet Architecture And Systems Laboratory (Division of Information Science)

Submitted on September 14, 2020

A Master's Thesis
submitted to Graduate School of Science and Technology,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
MASTER of ENGINEERING

Chauque Euclides Teles Tomas

Thesis Committee:

Professor Kazutoshi Fujikawa
(Supervisor)
Professor Keiichi Yasumoto
(Co-supervisor)
Associate Professor Ismail Arai
(Co-supervisor)

latency, datastores, databases, regression, cassandra

Reducing Tail Latency In Cassandra Cluster Using Regression Based Replica Selection Algorithm*

Chauque Euclides Teles Tomas

Abstract

Online applications adoption and success is driven by a multitude of factors among them the service response time, this is natural as users tend to prefer a faster service than a slower. However, it is challenging to deliver consistently fast response times due to performance variability inherent to the infrastructure running the application, this performance variability causes a fraction of user requests to experience unusual latency called tail latency. The tail latency assumes a more preponderant effect as the application infrastructure scales out, creating an additional delay point as an additional server is added, hence it is critical to track server's response time in order to prefer the faster server when possible, this is called replica selection. Replica selection algorithms have been proved to help decrease tail latency in key value datastores, however has these datastores evolved to support more sophisticated data models and query languages, previously proposed methods become unusable as they have been designed with certain assumptions about the datastores that no longer hold. In this work, a linear regression based replica selection algorithm is proposed. The regression model helps to estimate the how long a specific query is going to take to be serviced, and based on this information a server with more or less resources is chosen to service the query. The proposed approach is successful in reducing the higher percentiles (p999) latency up to 20% while not impacting negatively the throughput.

*Master's Thesis, Graduate School of Science and Technology, Nara Institute of Science and Technology, September 14, 2020.

Keywords:

latency, datastores, databases, regression, cassandra

Contents

List of Figures

List of Tables

1. Introduction

1.1 Background

For business oriented applications, fast and predictable response times are critical for a good user experience. To examine the impact of delays in application response time, a study was conducted by Amazon and Google [2], where a controlled delay was added on every query before sending back results to the user. Among their findings is the fact that an extra delay of 500ms per query resulted in a 1.2% loss of revenue. Google also found that the bounce probability in a website increases the longer the website takes to load[2].

However, it is challenging to consistently deliver fast response time, since applications are generally structured as multi-tiered, distributed systems, where even serving a single end-user request may involve contacting multiple servers. Significant delays at any of these servers can inflate the latency observed by users. To illustrate this, as described in [15], let us consider a well provisioned server that has an acceptable response time in 99% of the requests made to it, but the last 1% of the requests takes a second or more to serve, this situation would probably be acceptable. However, if we then scale the application to 100 servers with the same performance, and the request to be served is such that it needs a response from all servers, the responsiveness of the application, assuming independence between response times and using the "at least one rule" for probabilities, would change from 1% of the requests being slow to 63% of the requests taking more than a second to serve [27], in the worst case scenario. The problem just described is known as the tail latency problem, and is the reason why tail latency must be taken seriously to provide a good service.

For a given server, the poor tail latency is primarily caused by transient performance variability that cannot be completely removed. [15] describes the source of performance variability in servers as being the attributable, among other factors, to shared resources (when machines are shared by different applications contending for CPU cores, processor caches, memory and network bandwidth), scheduled background daemons, queuing and garbage collection.

In the same way that fault-tolerant computing aims to create a reliable whole out of less reliable parts, systems should be designed to create a responsive whole

out of less-predictable parts. Systems with this property are referred to as tail latency tolerant systems. Some of the techniques used to build such systems, leverage the fact that replication is used in different layers of the application to provide fault tolerance. For the database layer, which is the focus of this work, data is replicated to several servers in a cluster to provide high availability. When data is read, a replica selection algorithm should determine which node in the cluster the request should be sent to. This replica selection algorithm can be designed to meet different objectives, such as evenly distributing the requests among the servers,this is the case for algorithms like round robin 2.1.1, or select the nearest server based on latency or geographic proximity, the way it is done on Mongodb[10]. When the replica selection algorithm is coupled with the ability to predict query execution time, some database management task such as admission control [28], query scheduling [18] progress monitoring[23] and system sizing[30], can be achieved.

In the case of Cassandra, the built in selection algorithm provides good median latency, but the tail latency is often an order of magnitude worse than the median[27], mainly due to the already described server performance variability.

In order to avoid a degraded tail latency, in this work, a new server selection algorithm was experimented. It builds upon previously proposed algorithms, therefore it takes into consideration the server response time as inspired by [25], however [25] does not employ any mechanism to predict the query execution time relying only on the post execution feedback to perform adjustments, the proposed algorithm also use the expected duration of the query, as given by a regression model, as input for the replica selection algorithm. The following are the contributions of this work:

A linear regression based replica selection algorithm is proposed and implemented on a Cassandra cluster with realistic data and queries. The proposed approach is successful in reducing tail latency with an improvement on higher percentile latency of around 99.9 percentile and above, while not decreasing the throughput.

1.2 Terminology and Definitions

1.2.1 Tail latency

In online systems, usually what is more important is the service‘s response time, that is, the time between a client sending a request and receiving a response. In most cases latency and response time are used interchangeably, even though they are not quite the same. The response time is what the client sees, and the latency is the duration that a request is waiting to be handled [21]. In this work, this two terms are used interchangeably since latency is relatively harder to measure, the response time provides a good proxy for it.

For a given application, even if we only make the same request over and over again, a slightly different response time will be observed in every try. We therefore need to think of latency not as a single number, but as a distribution of values that can be measured. Having this distribution, to figure out how bad the outliers are we can look at higher percentiles the 95th, 99th and 99.9th percentiles, commonly abbreviated as p95, p99 and p999. These percentiles represent thresholds at which 95, 99 or 99,9 of requests are faster than a particular value. For example, if the 95th percentile response time is 1.5 seconds, that means 95 out of 100 requests take less that 1.5 seconds, and 5 out of 100 requests take 1.5 seconds or more.

Higher percentiles of response times, are also known as tail latencies. tail latencies become especially important in back-end services that are called multiple times as part of serving a single end-user request such as described on figure 1, since the end users request still needs to wait for the slowest request to complete, it takes just one slow call to make the entire end-user request slow.

1.2.2 Cassandra

Apache Cassandra [13, 1] is a NoSQL, open source, distributed and decentralized database that bases its distribution design on Amazon's Dynamo and its data model on Google's Bigtable. It was created at Facebook in 2007, and is now used at some of the most popular sites on the web.

Cassandra Architecture

Cassandra is frequently used in systems spanning physically separate locations, and provides two levels of grouping that are used to describe the topology

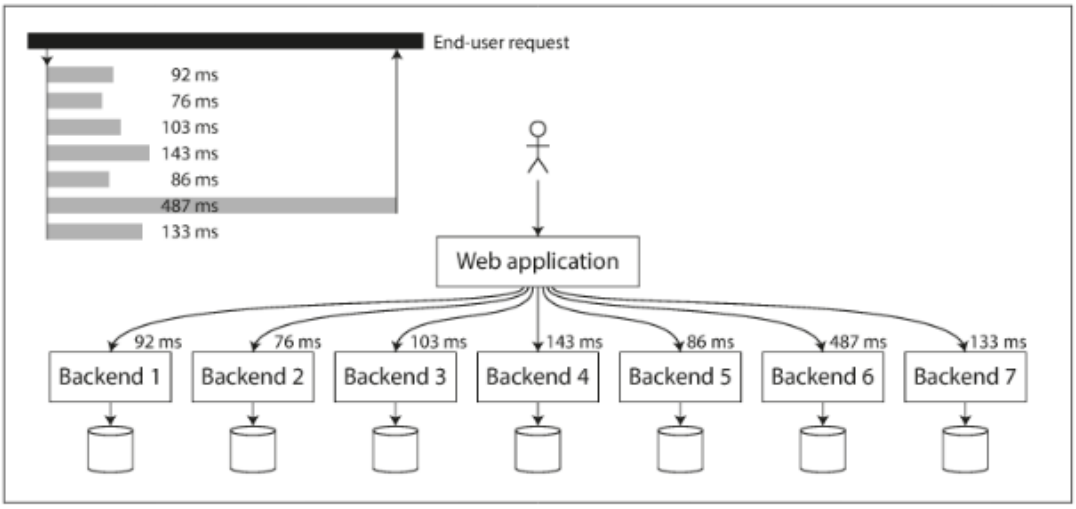


Figure 1: Effect of 1 back-end server poor latency on the entire request response time [21]

of a cluster: data center and rack. A rack is a logical set of nodes in close proximity, perhaps on physical machines in a single rack. A data center is a logical set of racks, perhaps located in the same building and connected by a reliable network.

Rings and Tokens

In Cassandra a cluster of nodes is logically represented as a ring. Each node in the ring is assigned one or more ranges of data described by a token, which determines its position in the ring.

Partitioners

A partitioner determines how data is distributed across nodes in the cluster, and is a hash function for computing the token of a partition key.

Replication

A node serves as a replica for different ranges of data. If one node goes down, other replicas can respond the queries for that range of data. Cassandra replicates data across nodes in a manner transparent to the user, and the replication factor is the number of nodes in the cluster that will receive copies (replicas) of the same data. If the replication factor is 3, then 3 nodes in the ring will have copies of the same data.

Cassandra Query Language

The syntax of Cassandra Query Language (CQL) is similar in many ways to SQL, but with some differences derived from the difference of Cassandra data model and those from the relational databases. In newer releases of Cassandra, the CQL has been reworked to support flexible data types, including simple character and numeric types, collections, and user-defined types, and also user defined functions. The later allowed the translation of a subset of TPC-H[8] SQL based queries into Cassandra compatible queries.

Clients

Just like applications connect to a relational databases using drivers like JDBC, there are also Client drivers available for Cassandra in various programming languages, for this work, the python driver was used. These drivers are easily embedded in the application code.

Load Balancing Policies

The client driver provides several functionalities that can be used to tune application behavior. These functionalities include Load Balancing. Because a query can be made to any node in a cluster, if a client were to direct all of its queries at the same node, this would produce and unbalanced load on the cluster. To get around this, the driver provides a pluggable mechanism to balance the query load across multiple nodes. The driver provides two basic load balancing implementations: the RoundRobin policy and the DCAwareRoundRobinPolicy. In this work, the referred pluggable feature was used to provide a custom code to implement the replica selection strategy.

Cassandra Reads and Writes

Writing data is very fast in Cassandra, because its design does not require performing disk reads or seeks on writes, which is what slows down some databases, all writes in Cassandra are append-only. On the other hand, because clients can connect to any node in the cluster to perform reads, without having to know whether a particular node acts as a replica for that data, reads become easy in Cassandra. If a client connects to a node that does not have the data it is trying to read, the connected node will act as a coordinator node, to read the data from a node that does have the data, identified by token ranges. Due to this, reads are generally slower than writes, to fulfill read operations, Cassandra typically has

to perform seeks [13]. With this known behavior, this work focus on improving performance of read operations.

2. Related Work

In this chapter, previous work found in the literature is presented and reviewed and a case for an alternative approach is made. The related work covers two major areas: replica selection algorithms and query duration prediction.

2.1 Replica Selection

As already stated, a recurring pattern to reducing tail latency is to exploit the redundancy built into each tier of the application architecture. [25] shows that the problem of replica selection — wherein a client node has to make a choice about selecting one out of multiple replica servers to serve a request — is a first-order concern in the context of taming tail latency. However, interestingly, the impact of the replica selection algorithm has often been overlooked. For example, layering approaches like request duplication and request reissues (also known as speculative retry) [15] in a poorly performing cluster should be cause for concern as reissuing requests but selecting poorly-performing nodes to process them increases system utilization [29] in exchange for limited benefits. Furthermore [25] shows that the replica selection strategy has a direct effect on the tail of the latency distribution.

Replica selection however, is made challenging by the fact that servers exhibit performance fluctuations over time. Hence, replica selection needs to quickly adapt to changing system dynamics. On the other hand, any reactive scheme in this context must avoid entering unusual behaviors that lead to load imbalance among nodes and oscillating instabilities. In addition, replica selection should not be computationally costly, nor require significant coordination overheads.

2.1.1 Traditional Replica Selection

The interest in replica selection algorithms is not new, it has already existed in different forms, being the most noticeably use of it in well known algorithms such as Round Robin and Least Outstanding Requests, which will be described here.

Round Robin The Round Robin Algorithm [26], is a simple and starvation free algorithm employed by process and network schedulers in computing. As the term is generally used, time slices are assigned to each process in equal portions

and in circular order, handling all processes without priority (also known as cyclic executive). Round-robin scheduling is simple, easy to implement. Round-robin scheduling can be applied to other scheduling problems, such as data packet scheduling in computer networks.

Least Outstanding Requests In the Least Outstanding Requests (LOR), for each request, the client selects the server to which it has the least number of outstanding requests. This technique is simple to implement and does not require global system information, which may not be available or is difficult to obtain in a scalable fashion. In fact, this is commonly used in load-balancing applications such as Nginx[5], recommended as a load-balancer for Riak[7] or Amazon ELB[3].

Static Replica Selection Static scheduling strategies permanently assign one or multiple servers to handle predefined request that might be of primary interest for the application. While this has been shown to be effective in some contexts, scheduling in high load situations where large request overwhelm a server, under certain workloads it may lead to the under utilization of the dedicated servers[16].

2.1.2 Dynamic Replica Selection

Previous algorithms have in common the fact that they do not take into consideration any server side metric, which makes them inefficient in accommodating time-varying performance fluctuations across nodes in the system. Therefore a replica selection strategy that takes into account the load across different servers in the system is needed. In the case of Least-outstanding requests strategy (LOR), [25] shows its flaw in reducing latency under realistic workloads, as shown in figure 2, which shows two replica servers that at a particular point in time have the service times of 4 ms and 10 ms respectively, assuming all three clients receive a burst of 4 requests each, based on purely local information, if every client selects a server using the LOR strategy, it will result in each server receiving an equal share of requests. This leads to a maximum latency of 60 ms, whereas an ideal allocation in this case obtains a maximum latency of 32 ms.

The C3 Algorithm

C3 [25] is a replica selection algorithm that handles service time variations among replicas. It has at its core 2 components: The Replica Ranking (or scor-

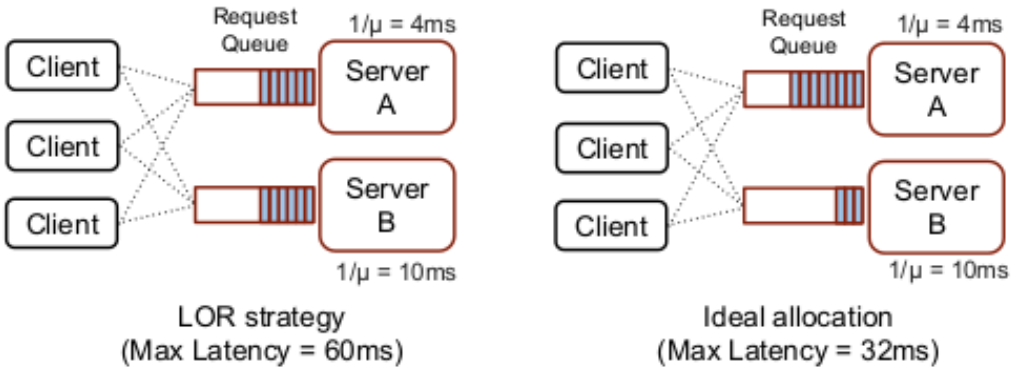


Figure 2: Sub-optimal server selection using Least Outstanding Requests[25]

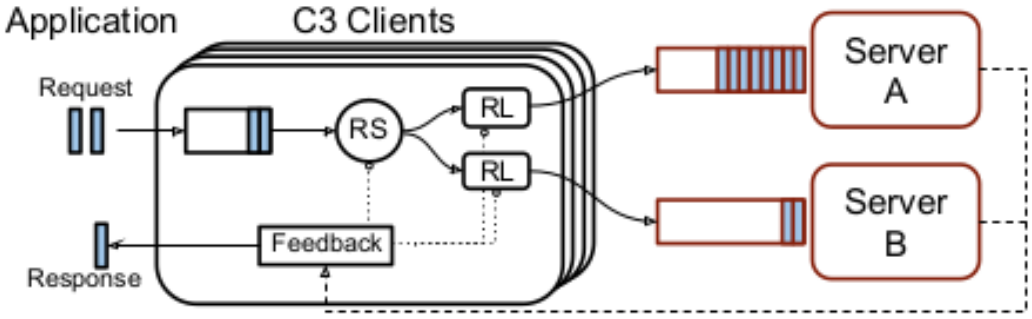


Figure 3: C3 Architecture Overview. RS: Replica Scoring, RL: Rate Limiter [25]

ing), and the Distributed Rate Control and Backpressure (or rate limiter)

The Replica Ranking component computes replica scores based on both service time and queue size, using the feedback from individual servers. The smaller the score is, the better the server is. This score is then used by a request coordinator for choosing the replica that is expected to better help reduce the request waiting time. Replicas are essentially ranked through a cubic function that is driven by an Exponentially Weighed Moving Average of the servers response times (EWMA), and additional terms to compensate for concurrency in the system, and penalizing long queues. The scoring function is given as:

$$\Psi_s = R_s - 1/\mu_s + (\hat{q}_s)^3/\mu_s$$

where

$$\hat{q_s} = 1 + os * n + q_s$$

is the queue-size estimation term, os_s is the number of outstanding requests from client to server s, n is the number of clients in the system, and R_s, $\hat{q_s}$ and μ_s are the exponential moving average of the response time as witnessed by the client, queue-size and service time feedback received from server s, respectively.

Additionally, the Backpressure component is used to avoid overloading a given replica queue. Eventually, because the replica is fast it can be simultaneously selected by several coordinators, C3 uses a rate control mechanism at each replica to limit the arrival of requests.

$$srate \leftarrow \gamma.(\Delta T - \sqrt[3]{\frac{\beta.R_0}{\gamma}})^3 + R_0$$

Where ΔT is the elapsed time since the last rate-decrease event, and R_0 is the saturation rate, the rate at the time of the last rate-decrease event. If the receive-rate is lower than the sending-rate, the client decreases its sending-rate multiplicatively by β. γ represents a scaling factor and is chosen to set the desired duration of the saddle region.

When a request is issued at a client, it is directed to a replica selection scheduler. The scheduler uses the scoring function to order the subset of servers that can handle the request, that is, the replica group (R). It then iterates through the list of replicas and selects the first server S that is within the rate as defined by the local rate limiter for S. If all replicas have exceeded their rate limits, the request is enqueued into a backlog queue. The scheduler then waits until at least one replica is within its rate before repeating the procedure. When a response for a request arrives, the client records the feedback metrics from the server and adjusts its sending rate for that server according to the cubic-rate adaptation mechanism.

Heron Algorithm

Heron [19] improves on C3 by taking into consideration the size of the values associated with the key being requested. This is due to the fact that in realistic workloads, the size in Kb of the values associated to the keys varies, and

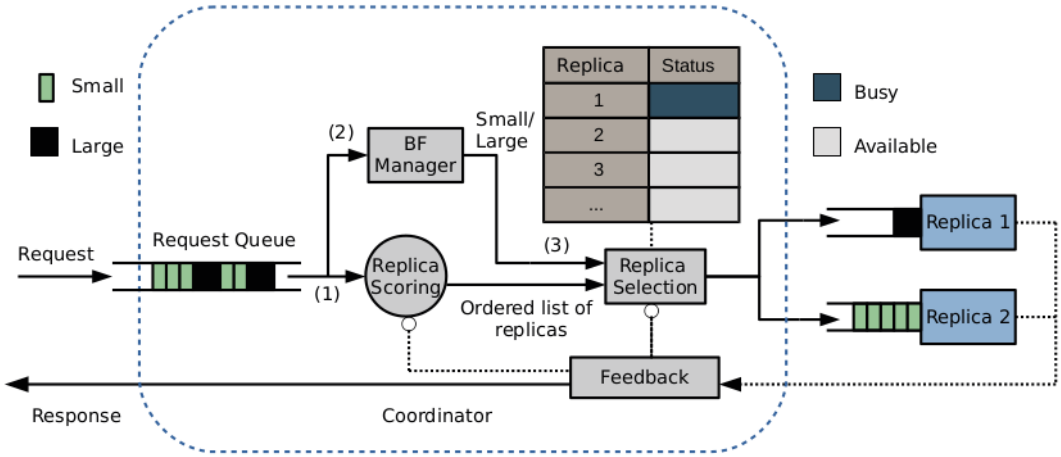


Figure 4: Heron Architecture Overview [19]

that variation in addition to the server performance impacts the service time for requests.

Heron, uses the same replica ranking strategy used by C3, however it does not use the same rate limiter algorithm, it instead uses a Bloom Filter [12] to track the size of the values associated to requested keys. Figure 4 shows the architecture of heron.

The replica selection selects the replica that is expected to serve an incoming request faster than the other replicas. It uses three types of information: (i) whether the request is expected to access a small or a large value, as provided by a bloom filter manager; (ii) the relative score of the servers holding replicas for that key, as provided by the replica scoring module; (iii) whether these replicas are currently handling a request for a large value or not. The last information is maintained over time by the replica selection module. Specifically, the initial status of a replica having no request to process is set to available. As long as this replica processes requests accessing small values, its status remains available. Instead, when a request accessing a large value is scheduled on a given replica, its status becomes busy. The latter comes back to the available status when the processing of the large request is over.

The definition of large or small value, is configurable by the system administrator.

While C3 does not explore any query attribute leading, heron takes into consideration the size of the values associated to keys, as a proxy for the query duration, thus performing better when heterogeneous workloads are considered, however, as it is discussed in the next chapter this approach is no longer trustworthy for modern key/value stores including Cassandra since, more complex queries on the data a possible, thus sum queries, like aggregation queries, may operate in a range of keys rendering the bloom filter used by heron useful. Therefore, alternatives approaches that take into consideration such advanced capabilities of modern key data stores are required.

2.2 Predicting Query Execution Time

The ability to predict query execution time is useful for a number of database management tasks, including admission control , query scheduling, progress monitoring, and system sizing. Different methods are described in the literature, however they can broadly be divided into Analytical-Model based and Machine Learning based approaches.

2.2.1 Analytical-Model Based Approach

Analytical methods are typically used in query optimizer, which have as a primary goal choosing a good query plan. To compare the different plans, the optimizer uses cost models to produce rough costs estimates for each plan. Such models consist of two parts, a logical and a physical model. The former is geared towards the estimation of the data volumes involved, usually statistics about the data stored in the database are used to predict the amount of data that each query operator has to process. The underlying assumption is that a query plan that has to process less data will also consume less resource and/or take less time to be evaluated. The logical cost component depends only on the data stored in the database, the operators in the query, and the order which these operators are to be evaluated. The physical cost model on the other side accounts for the cost of algorithm and implementation of each operator, and it includes metrics like disk I/O required per operation [22]. However, the units used by most optimiser do not map easily onto time units, nor does the cost reflect the use of individual

resources. [17]

2.2.2 Machine Learning Based Methods

The typical workload in a database system consists of a mixture of queries of different types, running concurrently and interacting with each other. The interaction among queries can have a significant effect on performance. Hence, optimizing performance requires reasoning about query mixes and their interactions, rather than considering individual queries. A major hurdle posed by query interactions is in finding effective ways to capture and model them. There is a large spectrum of possible causes for interactions that includes resource-related, data-related, and configuration-related dependencies. Sometimes, interactions are benign. However, depending on the system setting, the effect of interactions can vary all the way from severe performance degradation to huge performance gains. Furthermore, an interaction that occurs when a database system runs on one hardware configuration may not happen when the same system runs on a different hardware configuration. The implication of these challenges is that the analytical cost models query will not work for modeling interactions. [11]

This has inspired alternative approaches to query run-time prediction, fundamentally based on machine learning [28, 11] developed a model that explores the query interaction while [24], take advantage of the query structure, and operators within the query to train a machine learning model on spark jobs.

3. Proposed Approach Using a Linear Regression Model

As the merits and demerits of the algorithms of related work are described in chapter 2, a motivation for an alternative approach that leverages the existing work is presented in this chapter.

3.1 Improvement Opportunity

While C3 and Heron, present very interesting algorithms for the problem of replica selection for reducing tail latency, some improvements are still possible. C3 replica scoring model has shown itself to be very effective so much so that it was not only adopted on the heron paper [19], but also in [20], however its rate control mechanism has been show to be sub-optimal by [20]. In addition, because the experimental work was solely based on synthetic data-set generated by YCSB [14], the heterogeneous nature of real workloads was overlooked.

The Heron [19] algorithm, improves on this aspect by incorporating the requested value size as a proxy for expected duration of the query. The assumption being: larger values correspond to longer service times, and smaller values correspond to shorter service time. This assumption may hold true to key value data-stores using simple queries, however as Cassandra Query Language became richer thus supporting more complex queries, such as aggregation queries and user defined functions, the size of the value requested (query result) can not be know before run-time as it may depend on the calculations the query needs to perform.

Furthermore, for complex queries, as shown by [28] query duration is affected, not only by the retrieved data, but also by concurrent queries, and this can not be modeled by herons algorithm approach.

Last but not least, even for a single query, the response time is not always the same, as seen in figure 5, which shows the response time distribution for 5 different queries.

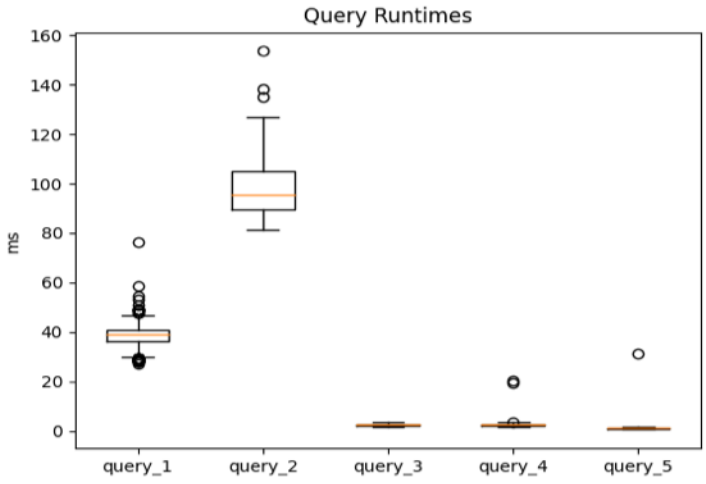


Figure 5: Response Time Variations of Queries

3.2 Regression Based Algorithm

The query behaviors described on section 3.1 motivate the approach presented in this work. For an application with defined number of queries, an experiment driven approach can be adopted in order to understand how those queries interaction affect each others run-time at different levels of concurrency, and a regression model can be used to extrapolate this interaction for other concurrency levels.

The resulting regression model can then be used at run-time to predict the duration of the query, and inform a replica selection decision.

The resultant regression model is used as per defined in the algorithm:

As described on the algorithm, the proposed method keeps a sorted list of servers based on their response time, using an Exponential Moving Average 3.2.1, and for requests expected to have a run-time time greater than a specified threshold, a faster server (the first on the list), is chosen to serve the request. For queries expected to have a run-time shorter than the threshold, the round robin selection is used.

Algorithm 1: Replica Selection Algorithm

```
sort(replicas)
dictRegressors
threshold
makeQueryPlan(query)
if query in dictRegressors.keys then
    execTime = predictExecTime(query)
    if execTime > threshold then
        return replicas[0]
    else
        return roundRobin(replicas)
else
    return roundRobin(replicas)
```

3.2.1 Exponential Moving Average

An exponential moving average (EMA), also known as running average is a technique used to analyze trends in a data set by creating a series of averages of different subsets of the full data set. Given a sequence of numbers and a fixed subset size, the first element of the moving average sequence is obtained by taking the average of the initial fixed subset of the number sequence. Then the subset is modified by excluding the first number of the series and including the next number following the original subset in the series. This creates a new averaged subset of numbers.

Mathematically, given a sequence $a_1, ..., a_N$ an n-moving average is a new sequence given by:

$$s_i = \frac{1}{n} \sum_{j=i}^{(i+n-1)} a_j$$

An illustration of a moving average with n equal 2, 3 and 4 is shown on figure 6

An exponentially weighted moving average (EMWA), instead of using the average of a fixed subset of data points, applies weighting factors to the data

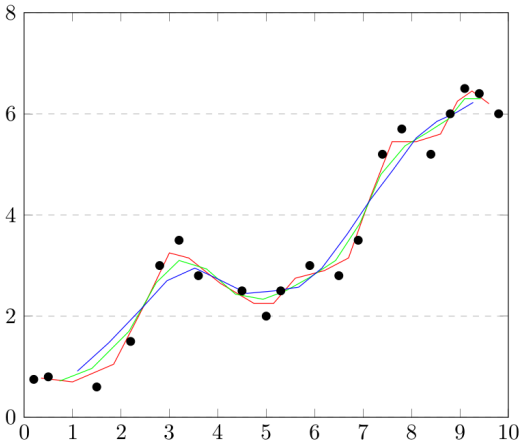


Figure 6: Moving averages for 20 data points with n equal 2(red),3(green) and 4 (blue)

points. The weighting for each older data point decreases exponentially, never reaching zero. The EMWA for a series Y can be calculated as:

$$S_1 = Y_1$$

for t¿ 1

$$S_t = \alpha Y_t + (1 - \alpha) S_{t-1}$$

Where α represents the degree of weighting decrease, a constant smoothing factor between 0 and 1. A higher value of αdiscounts older observations faster. Y_t is the value at a time period t, and S_t is the value of the EMWA at a time period t.

Using an exponential moving average allows for better tracking of servers response time, and provides a good value to infer how fast a server has been performing.

3.2.2 Linear Regression

A Linear regression is a predictive method in which the target value is expected to be a linear combination of the features. In mathematical notation, if $\hat{Y}$ is the predicted value, then:

$$\hat{Y}(w, x) = w_0 + w_1 x_1 + ... + w_p x_p$$

Among the existent linear regression algorithms, for this work the Ordinary Least Squares (OLS) was used, the main goal of this algorithm is to fit a model with coefficients $w = (w_1, ..., w_p)$ to minimize the residual sum of squares between observed targets, Y, in the dataset, and the targets predicted by the linear approximation $\hat{Y}$

$$\hat{\theta} = \arg\min \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

The main motivation for choosing a simple regression is the aim to have a lightweight model in order to avoid delay caused by the overhead of making a prediction, sklearn library documentation shows that linear regression models are much faster on their predictions when compared with other models, this characteristic is desirable to avoid overhead and decreased throughput. Superior models were not considered at this point, as they tend to be realized with additional overhead. Furthermore [28] reported that even though superior models, such as multi-layer perceptron, showed up to 10% improvements in accuracy compared to a simple linear regression, the overall improvement in performance of their system did not justify the added complexity of those models.

R-Squared R-squared evaluates the scatter of the data points around the fitted regression line. It is also called the coefficient of determination, or the coefficient of multiple determination for multiple regression. For the same data set, higher R-squared values represent smaller differences between the observed data and the fitted values. R-squared is the percentage of the dependent variable variation that a linear model explains.

$$R^2 = 1 - \frac{SSE}{SST}$$

where SSE is the Sum of Squared Errors given by:

$$SSE = \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

And SST is the Sum of Squared Total, given by:

$$SSE = \sum_{i=1}^{n} (Y_i - Y_i)^2$$

Y_i is the mean of Y.

R-squared is always between 0 and 100% or 0 and 1. 0 represents a model that does not explain any of the variation in the response variable around its mean. The mean of the dependent variable predicts the dependent variable as well as the regression model. 1 represents a model that explains all of the variation in the response variable around its mean.

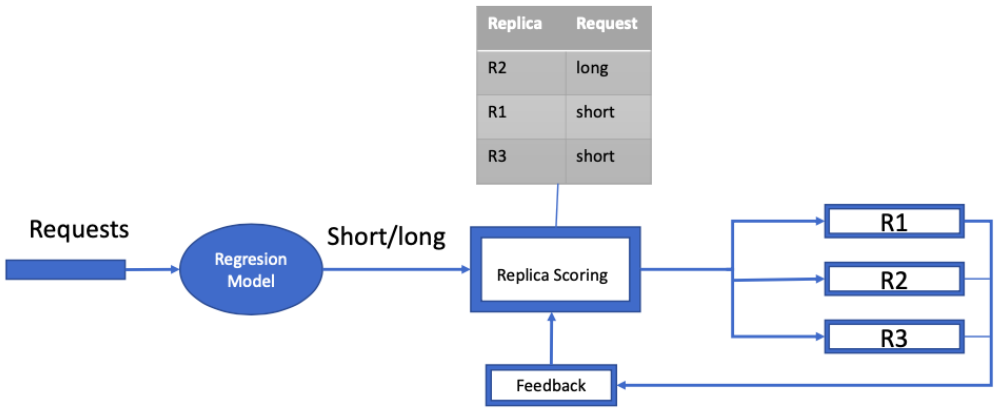


Figure 7: Proposed Method Overview

4. Implementation Of The Regression Based Algorithm In a Cassandra Cluster

This chapter is aimed at describing the details of the implementation of the proposed algorithm. The process of generating the training data, the queries used, etc.

4.1 Regression Based Query Execution Time Prediction

Figure 7 shows the main functional parts of the proposed algorithm. At the client side, a regression model is used to determine whether the incoming request is a long request or not based on a configurable threshold. Based on the result obtained through the regression model, the Replica Scoring component, which keeps a sorted list of servers response time, chooses to forward the incoming request to the fastest server if it is deemed as a long running query, if the query is deemed as a short running, it is sent to the servers following a round robin approach. This strategy allows for the fastest, maybe the more resourceful servers to process high loads than the slower servers. However, whenever long running queries are not present, the fastest servers do not remain idle, as the load is distributed even among the servers, through a round robin approach.

To predict the query run-time before its execution, an experimental approach

is used with the aim of observing each query run-time values distribution.

4.1.1 Training Data Generation

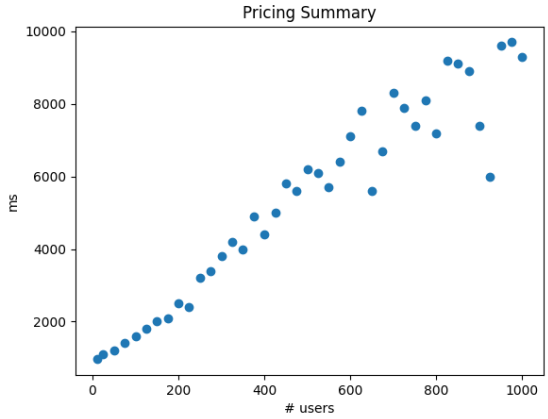
To generate the data to fit the regression model, a key-space with 3 tables was created in the cluster. This tables correspond to TPC-H benchmark tables (order, customer and lineitem tables). Records were inserted in each table and subsequently locust 4.1.1 was used to generate requests using 5 query templates from the TPC-H benchmark. For each run, locust was run for 5 minutes and the output file saved. This process done repeatedly allowed the creation of multiple files containing the run-time information for different percentiles for the queries as per the output of locust 4.1.1.

These files were later parsed and consolidated to create the training and testing data-set. As expected, each query exhibit a different behavior in terms of run-time values, a visualization of these run-times, can be seen in figure 8. And also each query has its own data, thus a regression model for each query is needed.

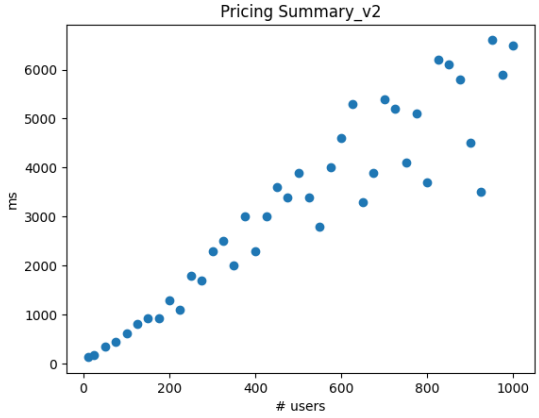
Locust Locust [4] is an easy-to-use, distributed, user load testing tool. It is intended for load-testing web sites (or other systems) and figuring out how many concurrent users a system can handle. It also provides response time percentiles as output of the test, which helps to understand when a system is performing poor than expected.

The idea is that during a test, a swarm of locust users will attack the system. The behavior of each user is defined by using Python code, and the swarming process is monitored from a web UI in real-time, logged to the console or written to a file. In the case of this work, each user would be responsible of running a specific query with a defined frequency.

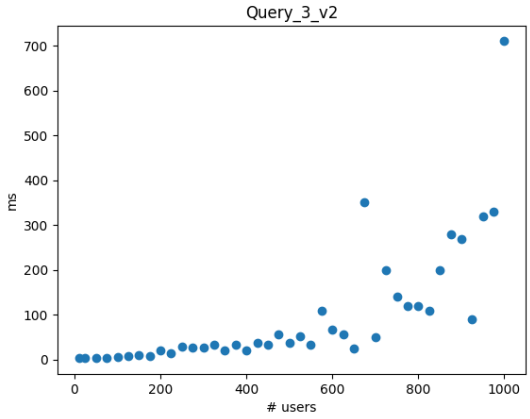
Locust is completely event-based, and therefore it is possible to support thousands of concurrent users on a single machine, and uses light-weight processes, through Gevent[9]. Each locust swarming your site is actually running inside its own process (or greenlet, to be correct). Figure 9 shows a typical output of locust statistics, where the header is mainly composed by specific percentiles, and the columns represent the threshold time value for a specific percentile.



(a)

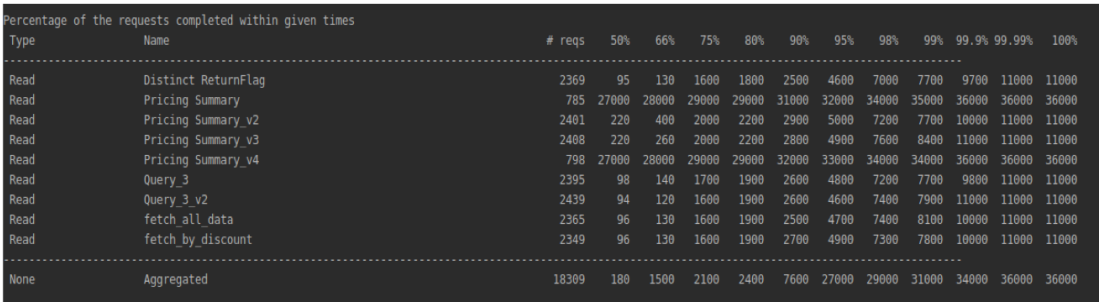


(b)



(c)

Figure 8: Run-time for Different Queries Under Different Concurrency Levels



Percentage of the requests completed within given times

Type	Name	# reqs	50%	66%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%
Read	Distinct ReturnFlag	2369	95	130	1600	1800	2500	4600	7000	7700	9700	11000	11000
Read	Pricing Summary	785	27000	28000	29000	29000	31000	32000	34000	35000	36000	36000	36000
Read	Pricing Summary_v2	2401	220	400	2000	2200	2900	5000	7200	7700	10000	11000	11000
Read	Pricing Summary_v3	2408	220	260	2000	2200	2800	4900	7600	8400	11000	11000	11000
Read	Pricing Summary_v4	798	27000	28000	29000	29000	32000	33000	34000	34000	36000	36000	36000
Read	Query_3	2395	98	140	1700	1900	2600	4800	7200	7700	9800	11000	11000
Read	Query_3_v2	2439	94	120	1600	1900	2600	4600	7400	7900	11000	11000	11000
Read	fetch_all_data	2365	96	130	1600	1900	2500	4700	7400	8100	10000	11000	11000
Read	fetch_by_discount	2349	96	130	1600	1900	2700	4900	7300	7800	10000	11000	11000
None	Aggregated	18309	180	1500	2100	2400	7600	27000	29000	31000	34000	36000	36000

Figure 9: Locust Statistics Output in the Console

Table 1: R Scores Values Per Query.

Query	R Squared(OLS)	R Squared(SGD)
Aggregated	0.89	0.89
Distinct ReturnFlag	0.82	0.82
Pricing Summary	0.92	0.92
Pricing Summary v2	0.88	0.87
Pricing Summary v3	0.88	0.87
Pricing Summary v4	0.91	0.91
Query 3	0.82	0.82
Query 3 v2	0.51	0.51
fetch all data	0.83	0.83
fetch by discount	0.83	0.83

4.1.2 Input and Output Data, and Fitting

As for the machine learning model, a linear regression algorithm implemented by sklearn[6] was used.

As mentioned before, each model query has its own regression, and each of these models takes as input the concurrency level value, and the query identifier. Thus a dictionary is used to hold all fitted regression at run time. As for the output the model provides an estimate of the time the query will take to be processed.

For each query regression the R squared value was computed as seen on the table 1, and the fitted line for one of the queries is shown in figure 10.

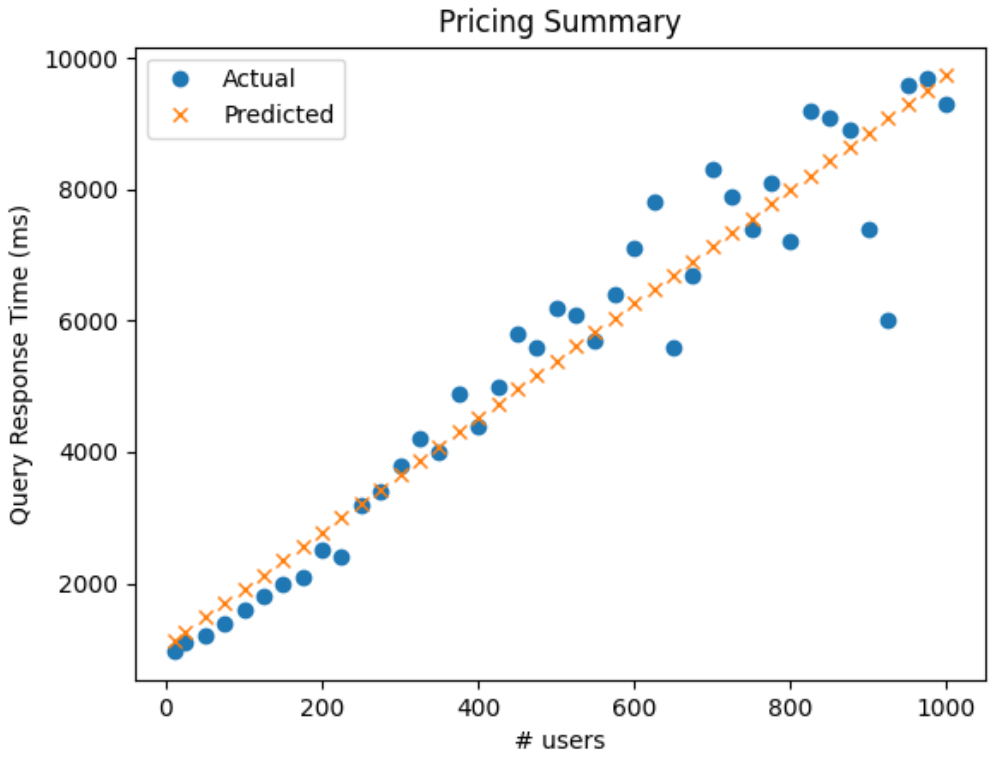


Figure 10: Fitted line for one query

4.2 Replica Selection Implementation

Once the regression models for each query were defined, they had to be used on run-time to actually make the prediction for every incoming query. To accomplish these the python driver was extended to support an additional loading balancing policy. The main difference of the policy implemented with those that come with Cassandra out of the box is the fact that the proposed policy takes into consideration a threshold value, and also the concurrency level as provided by locust, and use the model to predict the run-time of the query at hand. If the predicted run-time is greater than the threshold a faster server is selected, as per the exponential moving average calculation. Because there are multiple regression, 1 for each query type, every time a new query comes it needs to be identified so that the correct model is used. To accomplish this a dictionary, which is an implementation of a hash function is used, the query string constitutes the key of the dictionary and the trained regression model the corresponding value.

5. Experimental Results and Discussion

This chapter is divided into 3 major section, on the first the experimental environment is described, as well as the data used. Next the performed experiments are described and then on the final section the results of the experiments is presented and discussed.

5.1 Experimental Environment

As a testing bed for the experiments, a cluster comprised of 8 servers Sun Fire X2270 2.26GHz Intel Xeon L5520 4 Cores, 12GB RAM and 500GB 7200rpm SATA disk, was used. All 8 servers were connected through a Gigabit speed network. As for the load generating machine, which was used to instrument locust and generate the load, it was a Intel Core i7-4790 3.60GHz 8 cores with 16GB RAM, also connected through a Gigabit network.

The cluster was setup with a replication factor of 3, which means all data was distributed among the 8 servers, and each chunk of data was replicated among 3 servers. The read consistency level was set to 1, which means that the client driver will only wait for the response of 1 server.

To allow all the queries to finish the defaults timeouts were increased on Cassandra server side and client side (driver), to values big enough to prevent timeouts.

5.2 Experiments

The experiments executed were mainly aimed at comparing the higher latency percentiles using the proposed algorithm and other replication selection methods that is officially supported by all Cassandra drivers, which is at this point the round robin. Additionally, the throughput values are compared.

5.2.1 Environment With Homogeneous Servers

This was the simplest experiment, a workload was run using each replica selection algorithm for equal number of times, and the results were compared. This experiment aimed at observing how the proposed algorithm would behaved in

an environment where the server performance differences were inherently derived from transient processes on the server itself. Since the cluster used is composed by servers with the same specifications, it was expected a less accentuated improvement than it would be expected for a cluster with servers with different specifications.

5.2.2 Environment With Heterogeneous Servers

For this experiment the tc Linux tool was used to introduce some controlled delay, 0.5 seconds to 2 seconds in 4 servers. This was mainly to study how the replica selection algorithms would behave in an environment comprised of servers with different resources, which is not an unusual situations for systems that can scale horizontally, such as Cassandra. The servers on the original environment are from the same series thus have the same specifications, therefore under same conditions their performance is expected to be the same, and any difference can only be attributed to the (transient) behaviour of the processes running, and not on the underlying hardware. It was expected that in a heterogeneous environment, the proposed algorithm would perform significantly better than in homogeneous environments, since the difference in server performance would be much accentuated.

5.3 Results Discussion

5.3.1 Environment With Homogeneous Servers

As for the experiment with Homogeneous servers, following is the summary of the results found. As per figure 11, it is possible to observe an improvement for the long running queries, generally the data-set for this queries presented a more clear relation between the response time and the concurrency level thus resulting in a model with better predicting capacity, this resulted in a better segregation of these queries at runtime, conversely the short running queries had their performance as given by the response time affected negatively and the short running queries had their response time not affected by the concurrency level. This might suggest that a better performing model mighty help reduce the tail latency, however the prediction overhead mighty be increased, thus a compromise

will need to be made. Still in figure 11, the last two bars represent the aggregated response time i.e. the p999 percentile when all requests are considered together. In this case when can clearly see that the regression based model fares better than the round robin model. This might be attributed to the reduction in the response time of the longer running queries, because these longer running queries were directed to the fastest running server at runtime.

For a more granular comparison between the different percentiles, the p50, p90 and p999 are compared in figure 12, it is possible to observe that while higher percentiles are improved, the lower percentiles are negatively affected, this is also related with the poor performance of short running query, since the number of delayed queries among this type of queries increases.

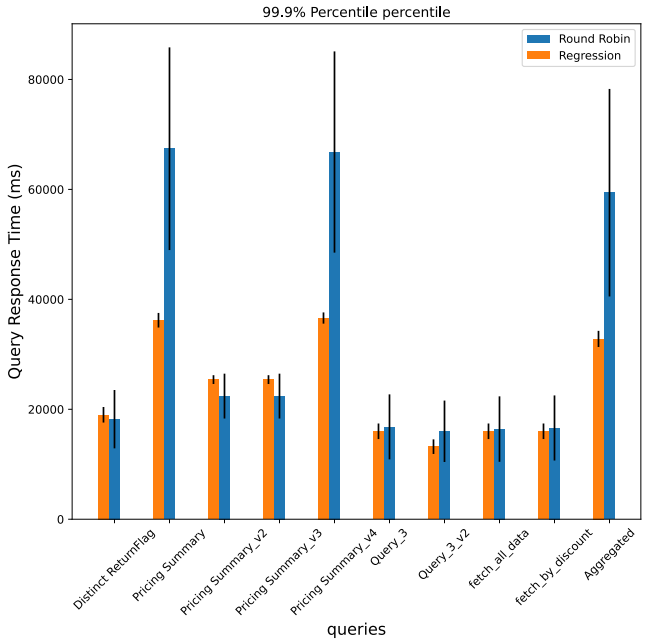
Finally, the effect of the proposed approach was evaluated in terms of its effect on the throughput, i.e the number of request per second. The result can be seen on figure 13. It can be seen that the result presents no major difference between the round robin and the proposed approach, if the standard deviation is taken into account.

5.3.2 Environment With Heterogeneous Servers

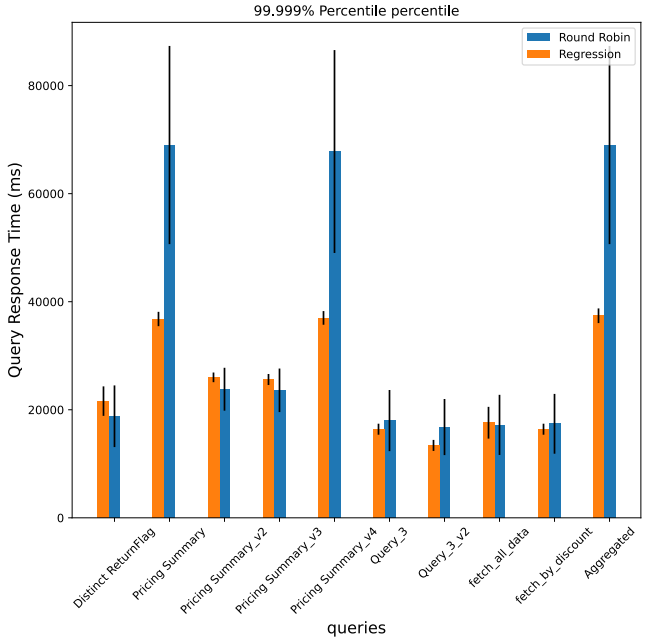
Following are the results for the experiment with added delay to simulate an environment with servers with different resources. From the figure 14 it can be seen that the response time for all queries increased, as expected, however it can also be seen that the linear regression based algorithm yields better response time for all queries. This suggests that the algorithm has potential for delivering consistently better results in environments with servers with tangible resource disparity, a situation that is common in horizontally scalable system that use commodity servers. The difference between servers response time, helps to absorb the overhead caused by the prediction process, thus the time lost in doing the prediction is compensated by the fact that a significantly faster server processes the query more rapidly.

Furthermore, as seen in figure 15 the lower percentiles p50 and p90 suffer less degradation when the policy is used. And last but not least, the throughput is substantially improved as seen in figure 16

Last, Fig 17show the comparison between the state of the art algorithm



(a)



(b)

Figure 11: Comparison of the a) p999 and b) p99999 for round robin and regression based replica selection

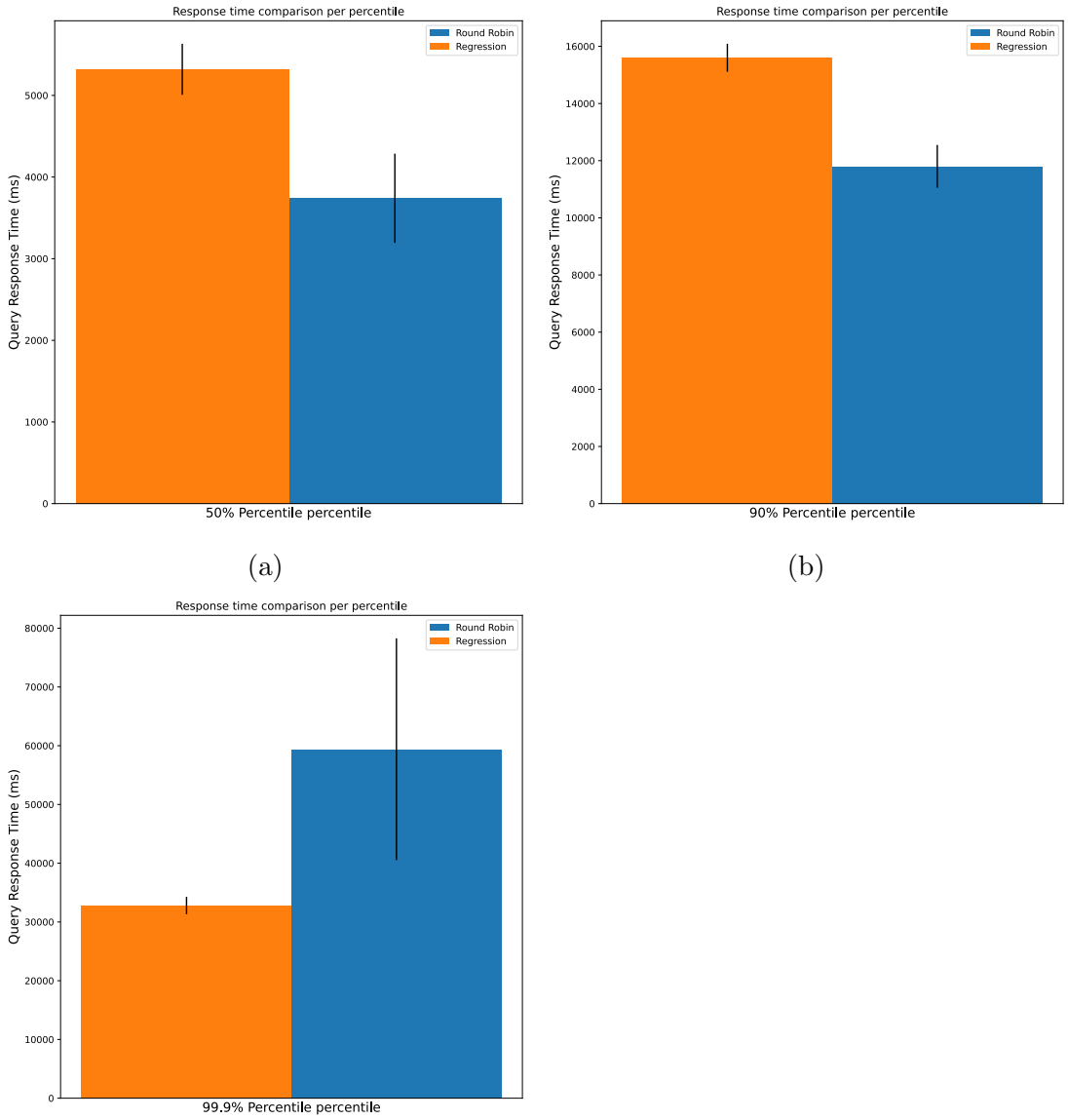


Figure 12: Comparison Between a)p50, b)p90 and c) p999 for homogeneous server environment

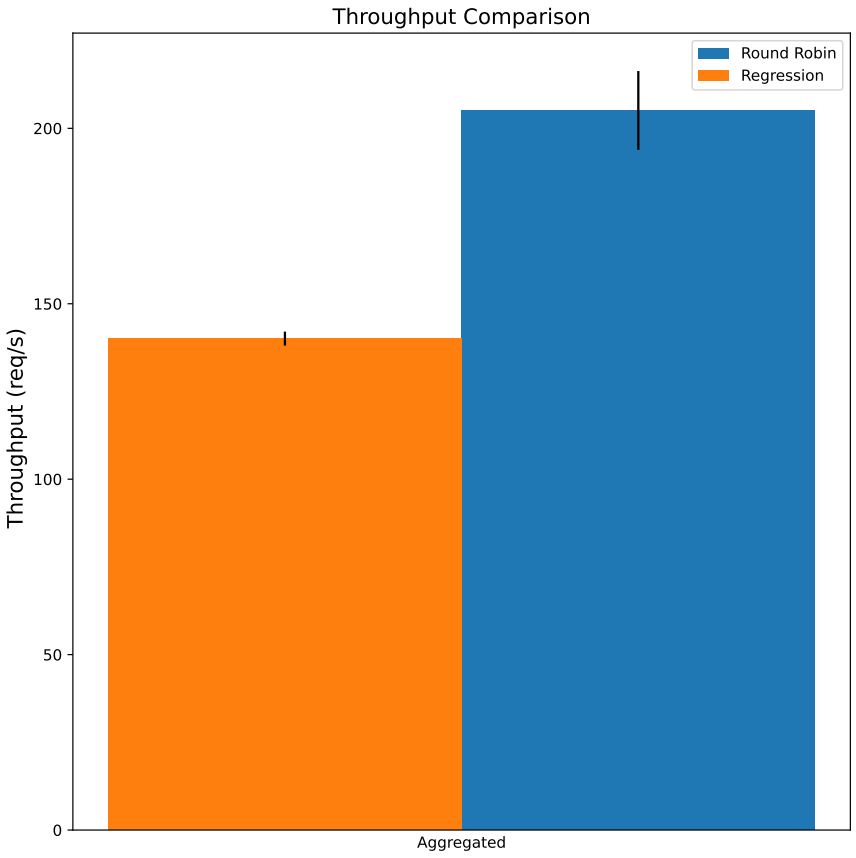
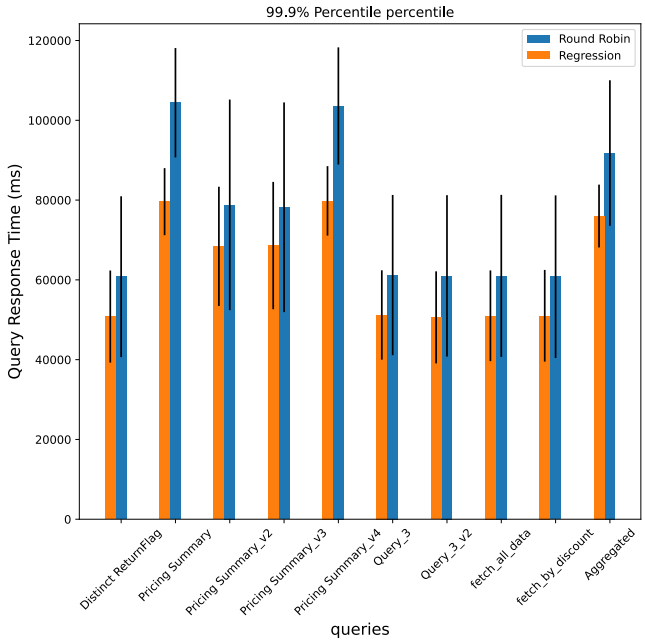
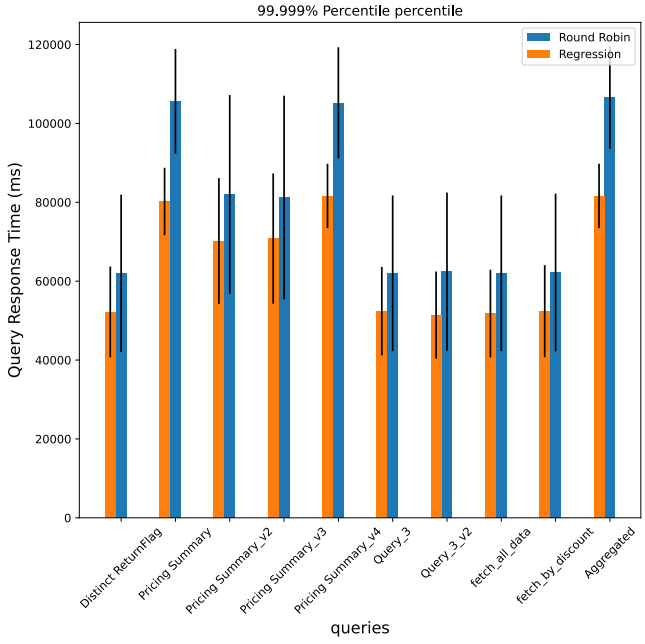


Figure 13: Throughput Comparison For Homogeneous Environment



(a)



(b)

Figure 14: Comparison of the a) p999 and b) p99999 for round robin and regression based replica selection when delay is introduced

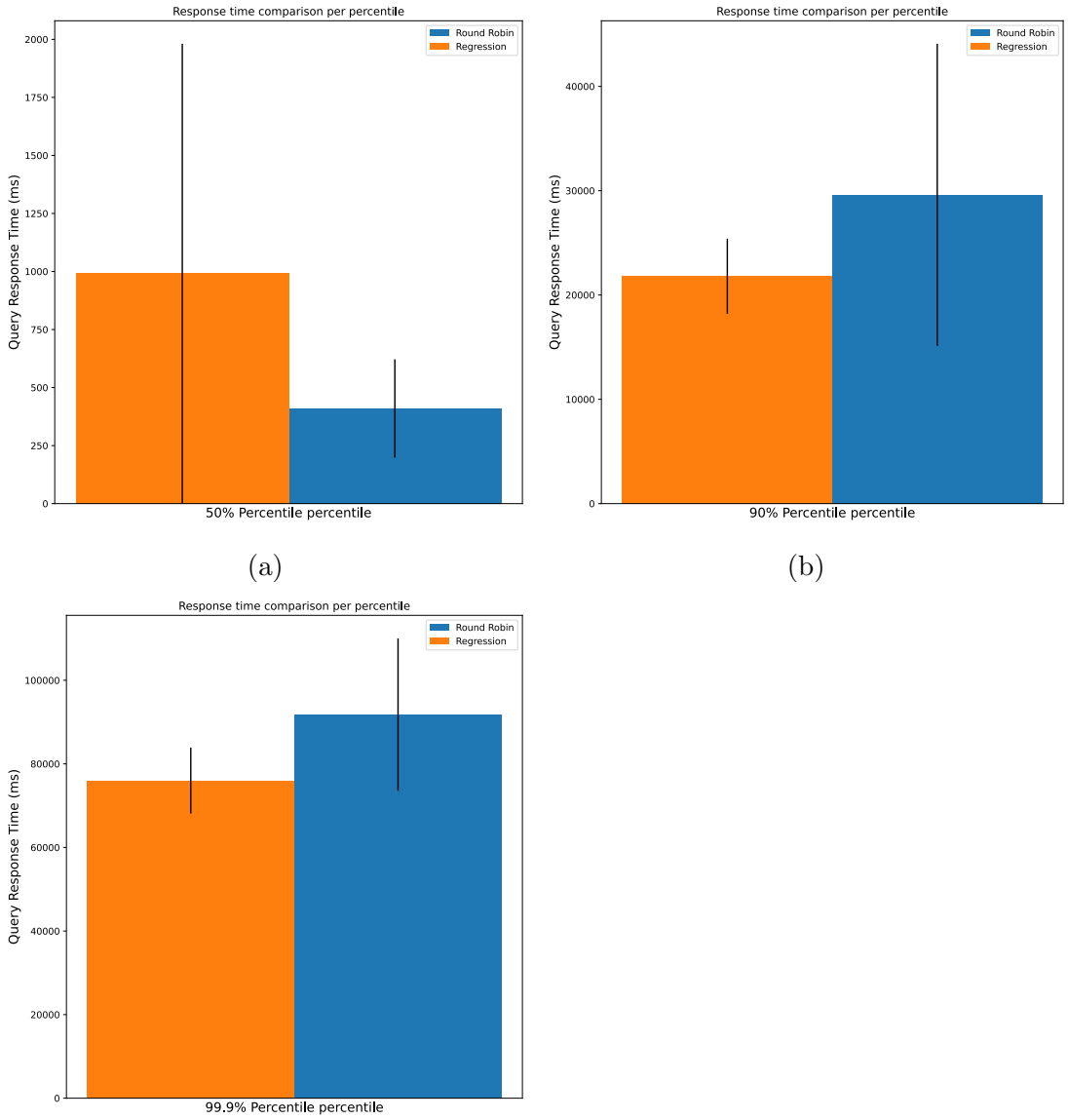


Figure 15: Comparison Between a) p50, b) p90 and c) p999 when delay is introduced

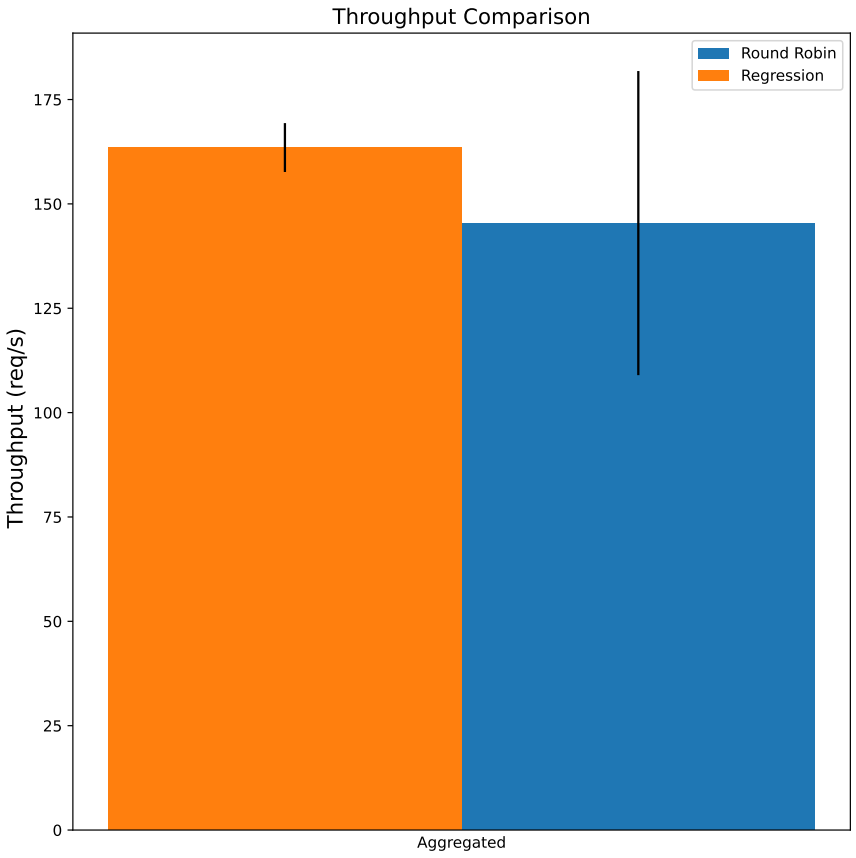
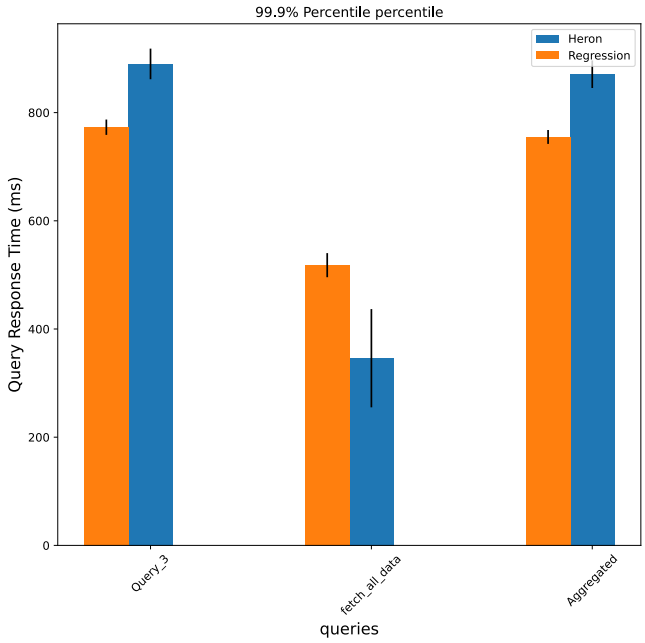
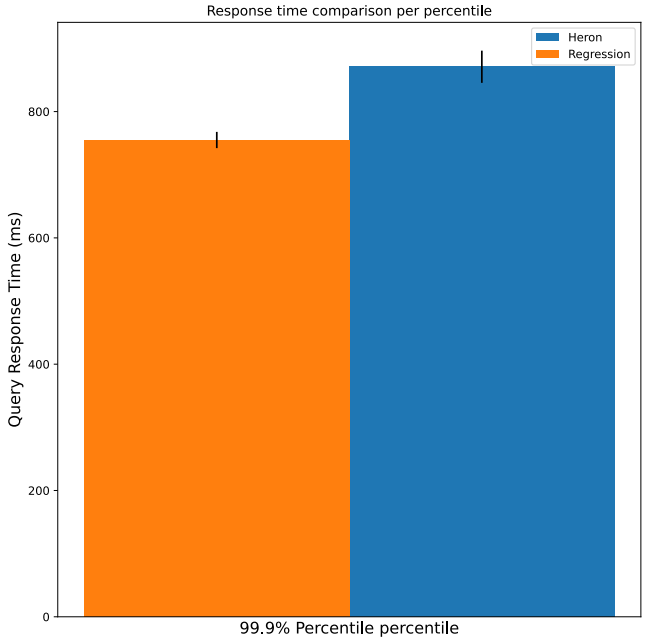


Figure 16: Throughput Comparison when delay is introduced

(Heron) and the proposed algorithm for a homogeneous servers scenario. The comparison was done for just a subset of queries since the Heron algorithm was implemented in a Cassandra code base that did not support aggregation queries, hence Heron algorithm itself was not designed to support them, i.e., the Bloom track how big individual values are, and not how big the result of a computation over many values is. For this reason, an in-depth comparison is not done. The results are consistent with what was observed when comparing the proposal with round robin.



(a)



(b)

Figure 17: Comparison of the a) p999 for all queries, and b) p999 aggregated for Heron and regression based replica selection in homogeneous environment

6. Conclusion

In the present work the tail latency problem was reviewed, and the problem of optimal server selection was considered as a method to reduce tail latency. Previous work had been based in a simpler query pattern, thus is no longer suitable for the complex queries that came to be supported in Cassandra, this served as motivation for exploring a new approach for server selection using a regression model to model the interaction between the queries. This new approach proved to be successful in reducing tail latency, while preserving the throughput, however it affected negatively the lower percentiles. While this could be less favorable for short running queries, it is a trade-off that can be beneficial for large scale application where the tail latency problem is more noticeable. A still remaining issue, is to evaluate how the proposed method will behave when heterogeneous data-set such as those Heron was designed to handle are considered.

Acknowledgements

I would like to thank the Japanese Ministry of Education, Culture, Sports, Science and technology for the opportunity given to me to pursue higher education in Japan and engage in research.

I would also want to thank NAIST for the support provided, specially to Professor Fujikawa and all faculty members of the Internet Architecture and Sytems Laboratory, for the guidance provided throughout my studies.

References

[1] Apache Cassandra. https://cassandra.apache.org/, accessed: 2020-07-10.

[2] Amazon found every 100ms of latency cost them 1% in sales. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/, accessed: 2020-07-14.

[3] Elastic Load Balancing - Amazon Web Services. https://aws.amazon.com/elasticloadbalancing/, accessed: 2020-07-14.

[4] Locust - A modern load testing framework. https://locust.io/, accessed: 2020-07-14.

[5] NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com/, accessed: 2020-07-14.

[6] Prediction Latency — scikit-learn 0.23.1 documentation. https://scikit-learn.org/stable/auto$_e$*xamples/applications/plot$_p$rediction$_l$atency.htmlsphx*$-$*glr* $-$ *auto* $-$ *examples* $-$ *applications* $-$ *plot* $-$ *prediction* $-$ *latency* $-$ *py*, *accessed* : $2020-07-14$.

[7] Riak. https://riak.com/, accessed: 2020-07-14.

[8] TPC. http://www.tpc.org/tpch/, accessed: 2020-07-14.

[9] What is gevent? — gevent 20.6.3.dev0 documentation. http://www.gevent.org/, accessed: 2020-07-14.

[10] Server Selection Algorithm — MongoDB Manual. https://docs.mongodb.com/manual/core/read-preference-mechanics, accessed: 2020-07-29.

[11] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. Modeling and exploiting query interactions in database systems. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM '08, page 183–192, New York, NY, USA, 2008. Association for Computing MBibTeX Cannot find 'references.bib'!achinery.

[12] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[13] Jeffrey. Carpenter and Eben Hewitt. *Cassandra: The Definitive Guide: Distributed Data at Web Scale*. O'Reilly Media, 2016.

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[15] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[16] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 497–509, 2016.

[17] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *2009 IEEE 25th International Conference on Data Engineering*, pages 592–603, 2009.

[18] Shenoda Guirguis, Mohamed A Sharaf, Panos K Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. Adaptive scheduling of web transactions. In *2009 IEEE 25th International Conference on Data Engineering*, pages 357–368. IEEE, 2009.

[19] Vikas Jaiman, Sonia Ben Mokhtar, Vivien Quéma, Lydia Y Chen, and Etienne Rivière. Héron: Taming tail latencies in key-value stores under heterogeneous workloads. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 191–200. IEEE, 2018.

[20] Wanchun Jiang, Haiming Xie, Xiangqian Zhou, Liyuan Fang, and Jianxin Wang. Haste makes waste: The on–off algorithm for replica selection in key–value stores. *Journal of Parallel and Distributed Computing*, 130:80 – 90, 2019.

[21] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.

[22] Stefan Manegold, Peter Boncz, and Martin L Kersten. Generic database cost models for hierarchical memory systems. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 191–202. Elsevier, 2002.

[23] Chaitanya Mishra and Nick Koudas. The design of a query monitoring system. *ACM Transactions on Database Systems (TODS)*, 34(1):1–51, 2009.

[24] Sara Mustafa, Iman Elghandour, and Mohamed A Ismail. A machine learning approach for predicting execution time of spark jobs. *Alexandria engineering journal*, 57(4):3767–3778, 2018.

[25] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 513–527, Oakland, CA, May 2015. USENIX Association.

[26] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[27] Sofie Thorsen. Replica selection in apache cassandra: Reducing the tail latency for reads using the c3 algorithm, 2015.

[28] Sean Tozer, Tim Brecht, and Ashraf Aboulnaga. Q-cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 397–408. IEEE, 2010.

[29] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, page 283–294, New York, NY, USA, 2013. Association for Computing Machinery.

[30] Ted J Wasserman, Patrick Martin, David B Skillicorn, and Haider Rizvi. Developing a characterization of business intelligence workloads for sizing new database systems. In *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP*, pages 7–13, 2004.