A Queueing Network Model for Performance Prediction of Apache Cassandra

Salvatore Dipietro
Imperial College London, UK
s.dipietro@imperial.ac.uk

Giuliano Casale
Imperial College London, UK
g.casale@imperial.ac.uk

Giuseppe Serazzi*
Politecnico di Milano, Italy
giuseppe.serazzi@polimi.it

ABSTRACT

NoSQL databases such as Apache Cassandra have attracted large interest in recent years thanks to their high availability, scalability, flexibility and low latency. Still there is limited research work on performance engineering methods for NoSQL databases, which yet are needed since these systems are highly distributed and thus can incur significant cost/performance trade-offs. To address this need, we propose a novel queueing network model for the Cassandra NoSQL database aimed at supporting resource provisioning. The model defines explicitly key configuration parameters of Cassandra such as consistency levels and replication factor, allowing engineers to compare alternative system setups.

Experimental results based on the YCSB benchmark indicate that, with a small amount of training for the estimation of its input parameters, the proposed model achieves good predictive accuracy across different loads and consistency levels. The average performance errors of the model compared to the real results are between 6% and 10%. We also demonstrate the applicability of our model to other NoSQL databases and other possible utilisation of it.

CCS Concepts

•**Computing methodologies** → **Modeling and simulation;** •**General and reference** → *Performance;* •**Information systems** → Database performance evaluation;

Keywords

NoSQL database, Apache Cassandra, queueing network model, simulation

*This paper has been partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644869. We would also like to thank XLAB for the support for the ScyllaDB installation and the MIKELANGELO project for cooperation. The data generated in this paper are released under CC-BY license at: https://zenodo.org/record/153836

Valuetools 2016 Taormina, Italy IT

DOI: 10.475/123_4

1. INTRODUCTION

The growing importance of Big Data applications has led in recent years to a rapid growth of interest for NoSQL databases [18]. Compared to traditional databases, NoSQL databases offer simple read and write operations, horizontal scalability and low latency. If tuned for the specific workload at hand, NoSQL databases can provide high availability and reliability guaranteeing leverage on distributed data replication mechanisms.

In spite of their popularity, performance engineering for NoSQL databases is still in its early stages. In particular, NoSQL databases are commonly deployed on cloud platforms where hardware resources are rented, calling for dedicated provisioning methods in order to strike the right trade-off between costs and quality-of-service. Unfortunately, database performance can be challenging to manage in cloud environments, where it can be compromised by several factors, such as network contention and CPU multi-tenancy. For this reason, it is desirable to support engineers with analysis tools to assess risks associated to a target deployment.

Today, service providers tend to over-provision NoSQL databases, wasting resources and increasing the costs for their infrastructure. Finding the optimal configuration, in terms of number of nodes, number of replicas (*Replication Factor*) and data consistency (*Consistency Level*), is thus important and desirable, but still considered a research challenge. Performance prediction based on stochastic models can support sizing activities of this kind by allowing engineers to easily compare alternative system configurations and predict performance and costs before deployment in the cloud.

One of the main challenges in modelling NoSQL databases is to properly account for the replication factors and data consistency levels. The goal of this paper is to develop a model that addresses this need for the Apache Cassandra NoSQL database [19]. Data replication is usually executed asynchronously, reducing response time but achieving a weaker data consistency level than traditional databases [6]. In order to ensure consistency, read requests thus need to hit multiple replicas before the database system can respond to the client. This synchronisation can considerably affect system response time and, for this reason, in this work we focus our attention only on the read requests. Finding the optimal trade off between data consistency, costs and offered performance thus requires a quantitative approach that should assess all of these dimensions.

The proposed model is based on extended queueing networks consisting of fork-join elements, finite capacity regions, and class-switching. Since the model includes several features that are not readily amenable to analytical solution, we focus here on simulation-based assessment. We plan to investigate in the future the applicability of mean-value approximations to our models, such as those used in layered queueing network solvers [12]. Our performance model explicit both replication factor and consistency level, allowing

VALUETOOLS 2016, October 25-28, Taormina, Italy

DOI 10.4108/eai.25-10-2016.2266606

engineers to compare alternative system setups.

We validate the proposed model against experiments based on the YCSB database benchmark [9], under varying consistency levels. Our results indicate that average error for the throughput is lower than 7% and lower than 10% for the response time, increasing prediction robustness compared to existing models.

The rest of the paper is organised as follows. Section 2 summarises related work. Section 3 describes the key features of Apache Cassandra involved in the study. Section 4 introduces the proposed queueing network model, which is subsequently validated in Section 5. We also demonstrate the applicability of our model to another NoSQL database, ScyllaDB, and propose a model-based analysis of Cassandra performance under quorum-based synchronization. Section 6 presents conclusions and future work.

2. RELATED WORK

Database performance modelling has been traditionally focused on relational databases [24]. With the increasing growth in popularity of NoSQL databases over the last ten years, research work in performance evaluation has been increasingly conducted also on these emerging database systems. Prior work has focused on comparing NoSQL systems in terms of read/write performance, scalability and flexibility [6, 22, 2, 17]. To the best of our knowledge only a limited number of works have been conducted on performance modelling of NoSQL databases, and available models specific to Cassandra are few [13, 25]. The models in [13] and [25] use two different modelling techniques due to the difficulty to represent, in the same model, the synchronization and the scheduling of the requests.

Gandini et al. present a high level queueing network model for Cassandra [13]. Each node of the distributed setup is described using two queues: one for the CPU and one for the disk. This model is able to capture throughput and latency for read and write requests. However, compared to our model, it does not take in consideration requests other than those executed locally and there is not synchronization of the tasks between the nodes. Moreover, communication latency is not taken in consideration and there is no representation of the CPU tasks limit that Cassandra has.

Subsequently, [25] presents a model for simulating read requests. The authors use Queueing Petri Nets (QPN) [3] in order to describe synchronization. QPNs are an extension of coloured stochastic Petri nets, particularly useful when the model needs to explicit scheduling at queueing resources. Each node is represented by only CPU processes, described as a timed queue. Compared to the model in [13], the authors limit the number of simultaneously running queries for a node and distinguish between local and remote requests. However, no information about the disk and network latency is considered. In case of records of large size, the disk and network response time can considerably affect performance. The model in [25] is validated by predicting response time and throughput of different Cassandra configurations and appears to be the most accurate among existing performance models. However, for some particular configuration the error between real system and simulation can grow over 40%.

Compared to the previous models, our model is able to capture network and disk latencies, distinguishing the different type of requests (represented in the model as classes) and attributing different CPU demands to each class and to each phase of the query execution. These improvements deliver more accuracy on throughput and response time prediction. In addition, our model is able to represent different data replication factor and simulate the major consistency levels implemented in Cassandra.

Queueing Petri Nets have also been used in [23] to predict and study Cassandra energy consumption. A study on how performance changes in relation to different data model structures has been conducted in [7]. Chebotko et al. present a query-driven data modelling for Cassandra whose results are highly different from the one applied for the traditional databases. The paper also argues about the importance of the modelling consistency levels. The relation between performance and consistency level has been studied in several papers [4, 1, 28]. To achieve higher performance, NoSQL databases prefer eventually consistency instead of strong consistency. For this reason, to achieve strong consistency and meet the consistency levels required by the application without sacrificing the performance, variations of Cassandra database have been developed [8, 14]. Chihoub et al. presents a self-tuning mechanism to change at run-time query consistency levels, taking in consideration system state. Differently, in [14] the Cassandra's node architecture is modified to always achieve strong consistency of the data.

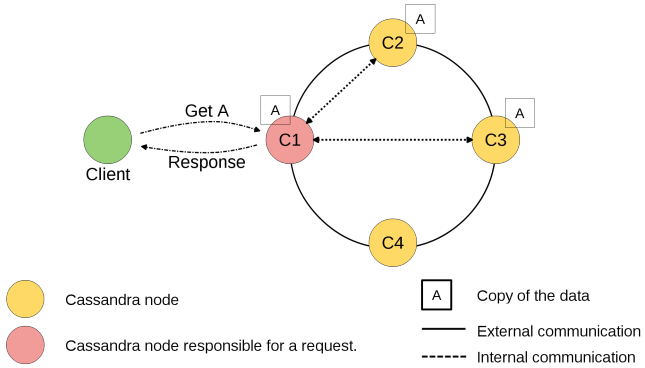

Figure 1: **Cassandra architecture overview.** In this example, Cassandra cluster is composed by 4 nodes with a replication factor of 3. The client is connected to the node c1 and it asks the data 'A' with consistency level ALL.

3. CASSANDRA ARCHITECTURE

Among all NoSQL datastores, Cassandra is one of the most popular and, nowadays, it is used in several Big Data products. Originally developed by Facebook [19], it was released as an open-source Apache project. Strong points are: a completely decentralised architecture without a single point of failure and promising linear scalability, high write performance and tunable consistency levels of the data inside the query.

As shown in Figure 1, a Cassandra cluster is organised as a ring of nodes. Each of these nodes is responsible to store part of the data. To offer an equal data distribution inside the ring, the hash key space is divided by the total number of *tokens* that are set for the database. Based on the number of tokens assigned to a node, a random subset of possible primary hash key values is associated to the node. The node becomes responsible for this subset of data for the entire database. Usually, the number of tokens is the same for each node in the ring. However, if the node characteristics are different inside the ring, the data distribution can take into account these heterogeneities by attributing a different number of tokens to each server. To achieve high availability, Cassandra replicates the data on some other nodes of the ring. The number of replicas is defined by the *Replication Factor* (RF). This implies that, if the cluster is composed by N nodes, each node will store a portion of the key space equal to RF/N.

To avoid network bottlenecks, the architecture allows each node to receive requests directly from the clients. When a client sends

a query to a node, the node serves it as *coordinator* and it is responsible for all the necessary operations to complete the query. Depending on the chosen *Consistency Level* (CL), Cassandra shows different behaviours [10]. The CL sets the number of copies that the system has to retrieve and compares before the response is returned to the client. The main CL options are three: ONE, QUORUM and ALL. For the purpose of this paper, we consider CL only for read requests. CL ONE involves the fewest operations and it is the fastest. To reply to the client, only one copy of the data is gathered. Differently, the QUORUM needs $(RF/2) + 1$ versions of the same data. The last option, ALL, requires that all the replicas are gathered to obtain the data, generating the response.

When a node is acting as the coordinator for a request, it performs two activities. First, in the parsing process, the node accepts the client request, locates one or more nodes holding data (according to the CL) and forwards them the requests. Next, in the end process, the node receives all the data from the other nodes, checks the consistency between all the responses and replies to the client. If one of the nodes has an old version of the data, the coordinator sends a query to update it.

Consider now a read request with CL ONE . The data request is sent to one node of the ring. If the data asked is stored in the node, the node performs a local request, fetching the data from the disk and returning it to the client. However, if the data is not stored in the node, the node acts like a *proxy*, asking the data to another node. Before returning the data to the client, the coordinator node checks if the data obtained corresponds to the latest version. Choosing a higher CL, the node needs to act more often as a proxy to ask the data from before to complete the requests. In case of CL ONE or QUORUM, Cassandra selects the node to retrieve data from based on measured node latencies in serving previous requests.

4. CASSANDRA MODEL

In this section, we present the Cassandra model used to predict database performance. The model is built using queueing network theory [20]. Queuing networks are a classic modelling technique for predicting latencies incurred by jobs that traverse a network of congested resources, each modelled as a queue. To represent and simulate our model, we use the JSIMgraph simulation environment provided with Java Modelling Tool (JMT) [5].

As described in Section 3, each node in Cassandra issues two main types of read requests: read data from the disk (*local request*) or retrieve data from another node (*remote request*). As part of a remote request, if a different node asks the data to the target node, the target node has to provide them (*remote incoming request*). Depending on the type of request that the node needs to process, the request follows different paths through the model. The different workflows are represented in Figure 2.

Figure 3 describes a single Cassandra node (identified as c1). The node is characterized by three queueing stations representing: network (c1_net), CPU (c1_cpu) and disk (c1_disk). For all the queues the scheduling policy used is non-preemptive first-come first-served (FCFS), with the exception of the CPUs for which processor sharing scheduling is used. To model all the functionalities that each node offers, other elements such as class-switches, a fork-join and a finite capacity region have been added to the model. In particular, class-switches are used to change the class of a request inside the model, the fork-join to manage multiple remote requests and finite capacity region to limit the number of simultaneous requests that the system can handle. This limit is static and set by default to 32 requests in the configuration file of each Cassandra node (parameter: concurrent_reads).

In addition, in order to model different types of requests, six

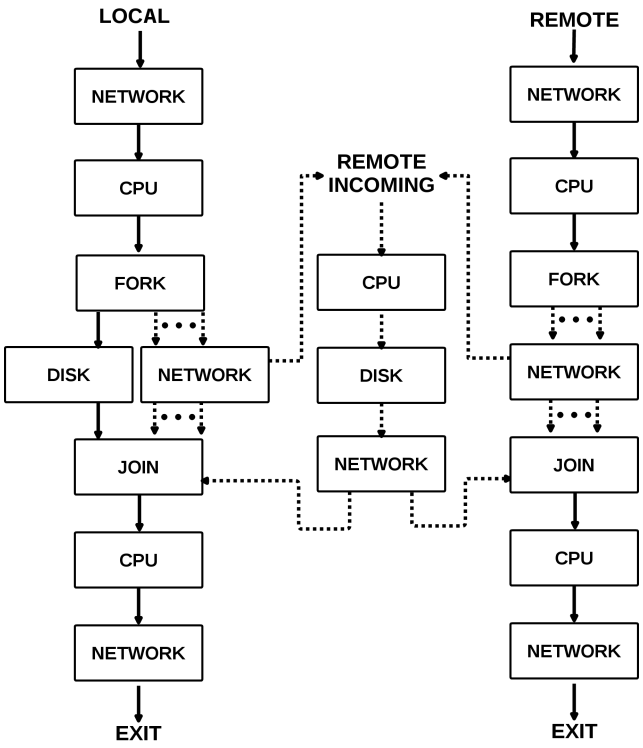

Figure 2: Local, Remote and Remote Incoming requests workflow. The boxes represent the node components used for a single node in our model. The lines represent where the request is forwarded. Continuous lines indicate the connections inside the same node whereas dashed lines represent external node connections.

different classes are defined in the model (described in Table 1). The number of classes is related to the number of nodes present in the system. In fact, to maintain information about the origin of a request for each read-remote request, a read-remote-*ID* class for each node is created, where *ID* identifies a Cassandra node.

In our model, the initial requests received by the node from the client can be only local or remote and enter in the node via the network queue (c1_net element in Figure 3). Based on the type of request, different paths may be taken. Such paths are presented in the following sections. We also discuss model parametrization and workload characteristics.

4.1 Local Request

Local requests are queries for which the node stores data locally on the disk. In this case it does not need to contact any other node if the CL is ONE. The left column of Figure 2 shows the main steps for this type of request. The read-local request from the network queue goes directly to the CPU queue through the *c1_initial* class-switch. After it has been served, the request goes through the fork and router, reaching the disk. In the case of CL QUORUM or ALL, the fork generates as many remote requests as needed in order to satisfy the CL. When the disk finishes the execution of the request, it reaches the join (*c1_join*) where it waits until all the remote requests come back from all the other nodes. In case of CL ONE, the join fires the request immediately. Afterwards, the request class is changed in read-local-end inside the class-switch *c1_end* and it is forwarded to the CPU to perform the ends operations before it leaves the node through the network queue and the class-switch *exit*.

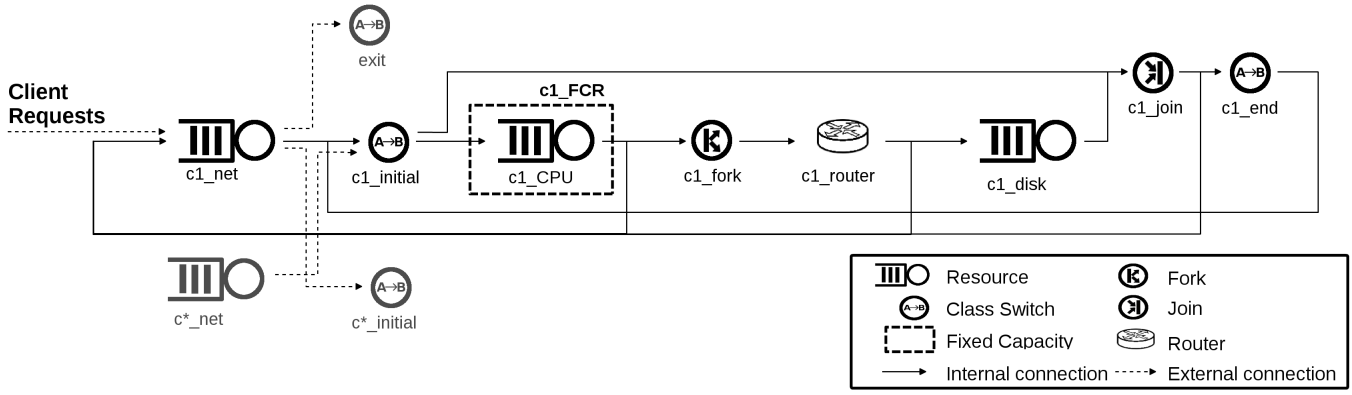

Figure 3: The model representation of a Cassandra node.

#	Class	Description
1	read-local	Local read request
2	read-remote	Remote read request
3	read-remote-*ID*	Remote read request where the *ID* node is acting as proxy
4	read-local-end	Local request that needs to perform the end operations before to return to the client
5	read-remote-end	Remote request that needs to perform the end operations before to return to the client
6	read-remote-return	The returned remote incoming request performed by another node

Table 1: Classes description.

4.2 Remote Incoming Request

Remote Incoming requests (see the central column of Figure 2) are the data that a proxy node retrieves from the target node to complete a query. The data gathering process is defined similarly to local requests. Remote incoming requests enter in the node from the initial class-switch of the node c1. The class request arrived to the node c1 is one of the read-remote-*ID* classes. The request is executed by the CPU. The fork and router forward the request directly to the disk that retrieves the information and sends it to the join. The node network (*c1_net*) takes the request and based on the request class (read-remote-*ID*), forwards it back to the original node (*ID*).

4.3 Remote Request

Remote requests are the requests for which the node does not store the data locally. In this case, the node needs to act as a proxy and issue remote requests towards other nodes to complete it. Requests generated in this scenario are the remote incoming requests received from other nodes. Differently from the local requests, these queries never reach the local disk of the node. After it is served by the network queue, the request reaches the *c1_initial* class-switch in which the request changes class, becoming a read-remote-C1 request, and it continues execution into the CPU. As described in Section 3, this type of requests are characterised by two main CPU operations: parsing and end. When the request hits for the first time the CPU, it executes the parsing phase and after it is sent to the fork. Here, based on the CL applied to the queries, the fork generates as

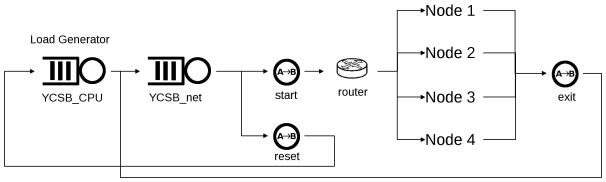

Figure 4: Cassandra model overview at high level of granularity.

many requests as the CL needs. These requests are forwarded to the router and then redirected to the network queue. Here, the requests are randomly sent to some other nodes of the ring and executed as a remote incoming request by another node.

When the read-remote-C1 requests come back from the remote nodes, they change class in read-remote-return (inside *c1_initial*) and they are forwarded to the join. When all the generated requests come back, the join fires generating a single read-remote-end request that is sent to the CPU via the initial class-switch. Inside the CPU a read-remote-end request is processed to perform the end operations. Finally, the request exits from the node through the network queue (*c1_net*) and is sent back to the client through the *exit* class-switch.

4.4 Workload

Another important part of our model is represented by the workload generator. For our experiments, we use the Yahoo! Cloud System Benchmark (YCSB) [9]. The YCSB is a workload generator developed by Yahoo! able to provide different kind of stress-test for various databases. Figure 4 provides an overview of the system, including the YSCB workload generator. In order to approximate better the performance of our model, we have decided to model it with two queues that represents the CPU and the network running the workload generator machine. YCSB CPU queue manages only read-local class requests. The new requests generated by the CPU are sent directly to the *start* class-switch through the network queue. This class-switch changes in read-remote some of the requests, according to the replication factor and CL applied to the system. The proportion of read-remote requests is equal to $(N - RF)/N$. Then, read-local and read-remote requests are forwarded to a router that sends them randomly to the nodes inside the ring.

When the requests have been executed by the node, the network queue of the node sends the response to the *exit* class-switch which

Class	ONE	QUORUM	ALL
read-local	0.8028	0.4078	0.5
read-remote	0.374	0.5712	0.6
read-remote-*ID*	0.6309	0.6423	0.6
read-local-end	0	0.4968	0.5
read-remote-end	0.4994	0.8917	1
read-remote-return	n/a	n/a	n/a

Table 2: CPU demands (in milliseconds) for different classes

Component	Demand value
Network Request	0.01333
Network Data	0.1333
Disk	0.0679
YCSB CPU	0.428

Table 3: Different component demands (in milliseconds).

changes class to read-remote-end. In the end, before being received by the YCSB CPU, the request is sent to the YCSB network and to a class-switch to reset the class in read-local. When the query comes back to YCSB CPU queue, the request is considered completed and a new query is generated by the YCSB workload generator.

4.5 Model Parametrization

To parameterize the model, we need to estimate disk and network capacity and the CPU demands for YCSB and nodes. To gather these information, we set up a Cassandra ring with four nodes and an additional machine for YCSB workload generator. Regarding network latency, two different requests are taken in consideration: query and response. They differ in size of data that they are carrying. We have observed that a normal query size is around 128 bytes. Differently, the response packet is characterized by the size of the data asked. To calculate the network demand, we multiply the packet size with the bandwidth information gathered from the Linux tool iperf [11]. Similarly, we have estimated disk demand using the data size and the Linux tool hdparm [21].
A different approach has been used for the estimation of the time that each class spends inside the CPU (demand). The demand has been calculated using the Complete Information algorithm [26]. To calculate demands, the algorithm requires request timestamps and response times. To gather these information we observed the network traffic exchanged between the nodes and the workload generator (YCSB). Analysing the IP address of different packets, we categorised three types of read requests. For the demand estimation of the different classes we took in consideration the timestamps packets of:

- Local request: the packets arrived from and departed to YCSB IP address.
- Remote incoming request: the packets arrived from and departed to each other node IP address.
- Remote request (parsing phase): from when the packet arrives from YCSB to the last packet departure directed to another node.
- Remote request (end phase): from when the first packet comes back from a node to YCSB response.

In case of local requests with CL QUORUM or ALL, the demand is calculated as a remote request. In the same way, we also estimate CPU demand of YCSB workload generator. The timestamps taken

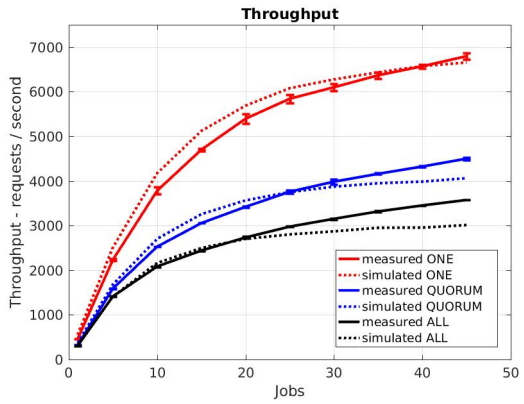

Figure 5: Mean throughput for the Cassandra cluster and our model.

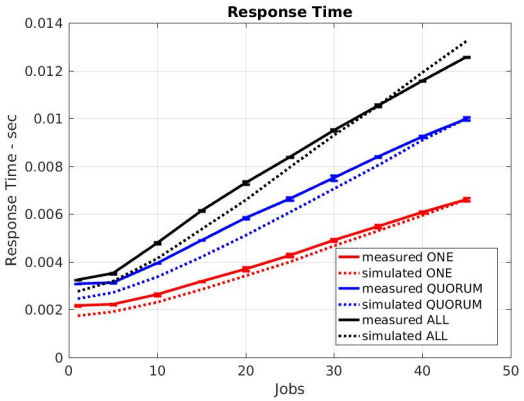

Figure 6: Mean response time for the Cassandra cluster and our model.

in consideration in this case are, simply, the packets that the server sent to and received from a node. To avoid the simultaneity of multiple requests, all the demand estimation tasks have been taken with a single job in the system. All the demand values estimated are reported in tables 2 and 3.

5. VALIDATION AND WHAT-IF ANALYSIS

5.1 Experimental Evaluation

To validate our queueing model we set up a Cassandra cluster composed of four nodes with a replication factor of 3. The nodes are installed on a private cloud managed with OpenNebula. The machines used as nodes have 2 CPUs and 4 GBs of memory. Each machine has 2 disks attached: a 20 GBs, where Cassandra commits logs and saved cache files are stored, the second of 100 GBs to store only database data. The operating system is Debian 8 with Sun Java 1.8 and Cassandra version 2.2.4. To avoid interfering with the performance of Cassandra nodes, the workload generator YCSB is installed on a separate virtual machine with 4 CPUs and 4 GBs of memory.

The database is set to equally distribute the key range between all the nodes in the ring. For the workload generator we used the default parameters that come with YCSB. Each row of the database is composed of 10 fields, each filled with 100B of data. The total data size for each record is 1KB, excluding the identification key. The keyspace is filled with 15 million of records. To emulate the

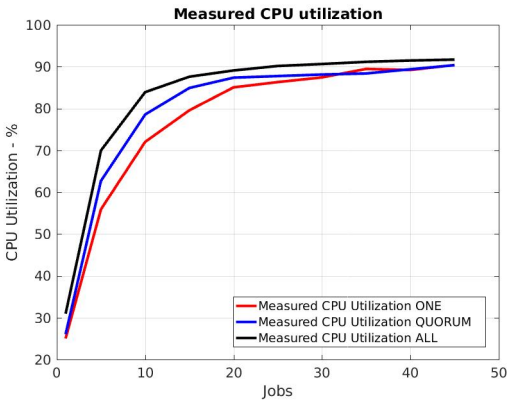

Figure 7: CPU utilisation for a Cassandra node with different Consistency level and clients in the system.

Consistency Level	Average error
ONE	6.79 %
QUORUM	6.41 %
ALL	6.98 %

Table 4: Average throughput relative error with the different consistency levels.

different number of simultaneous clients in the system (also called jobs), the same number of threads are set in YCSB. Each experiment on the real system is repeated three times and it has the duration of 30 minutes, plus 5 minutes of warm-up.

The evaluation compares the throughput and response time of the real system with the results obtained through the simulation of the JMT model. Figure 5 shows the average throughput values recorded by YCSB at the end of the execution of each experiment. The figure also shows the standard deviation values recorded for the same experiment for the data collected from the real system.

Before comparing the result of the two approaches, we can analyse the effects that different CLs have on the real system. Using ONE as CL with 45 jobs in the system, or 45 simultaneous client requests, the system is able to serve around 6800 requests per second. Compared to ONE, the throughput of the system is reduced more than 40% and up to 60% when QUORUM and ALL CL are used. In fact, increasing the CL, the CPU utilisation of each node grows faster like the graph reports in Figure 7. In addition, the response time also changes considerably, increasing linearly the average time that the clients waits for query completion.

Comparing the results obtained from the real system and from our queueing model, it is possible to notice that our model is sensitive to all major changes in the database. As shown in Figure 5, the model is able to approximate with a very small error the throughput of the real system, as the number of jobs and CLs vary. As presented in Table 4, the average throughput relative error is below 7% for all analysed CLs. Using our model, we also estimate the response time of the queries for the same experiments. The results are reported in Figure 6. Even in this case, the model is able to catch with good accuracy the real system performance. The average response time error, as reported in Table 5, is below the 10%.

5.2 Case Study: applicability of our model to other NoSQL databases

As part of our evaluation, we consider the possibility to apply

Consistency Level	Average error
ONE	8.17 %
QUORUM	9.59 %
ALL	7.64 %

Table 5: Average response time relative error with the different consistency levels.

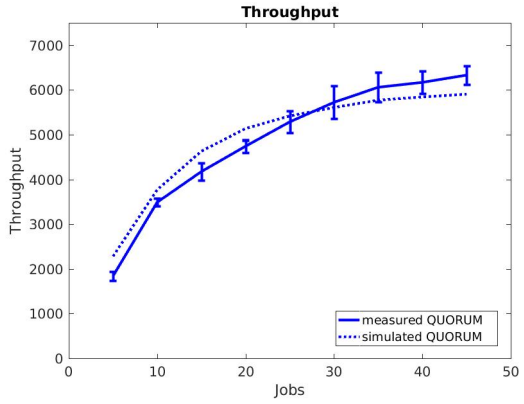

Figure 8: Mean throughput for ScyllaDB cluster and our model.

our out-of-the-box model to other NoSQL databases. Due to the similar system architecture and behaviour of the queries execution, we decided to apply the model to ScyllaDB [29].

ScyllaDB is an Apache Cassandra compatible column store database that aims to optimise the performance of each node participating to the database. This increment of performance is obtained using a different programming language (C++) and a complete new software architecture optimised for modern hardware resources. Typically, NoSQL datastores run on a single Java Virtual Machine. To reduce the idle periods generated by the usage of the Garbage Collector, which tend to increase the latency of the system, these applications usually use page cache and complex memory allocation strategies. To avoid these expensive locking mechanisms, ScyllaDB uses an engine for each CPU core of the system, which operates with a low degree of synchronisation. This reflects on the database with a lower usage of the hardware resources and an increment of database performance.

To gather performance data for ScyllaDB, we deploy the same Cassandra configuration used in Section 5.1. The ScyllaDB deployment is composed by four machines, with a replication factor of three and 1KB of data for each record stored. Due to the optimised usage of the recent hardware features, ScyllaDB requires the presence of specific features of the CPU on the VM. For this reason, ScyllaDB VMs are deployed using a different CPU architecture that supports the SSE4.2 CPU flag instructions [16]. To accomplish ScyllaDB requirements, we force the hypervisor to create virtual machines with the Intel Nehalem CPU microarchitecture [15]. The usage of this microarchitecture also brings more benefits to the hardware resources allocated to the VM thanks to a larger bandwidth for the network and disk.

To adapt the model for ScyllaDB, we also need to remove the fixed capacity region set on the CPU because ScyllaDB does not limit the number of simultaneous queries that each node can handle. To parametrize our model, we:

- Estimate the network and disk bandwidth using the Linux tools hdparm and iperf. The two bandwidths have been multi-

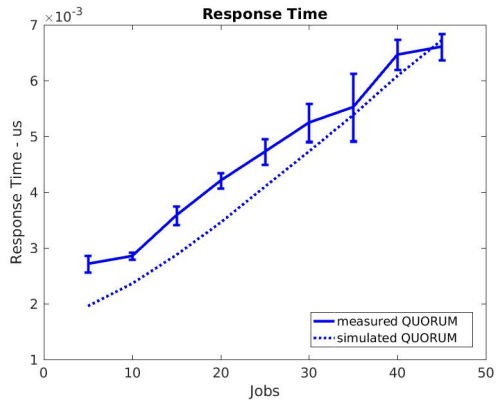

Figure 9: Mean response time for ScyllaDB cluster and our model.

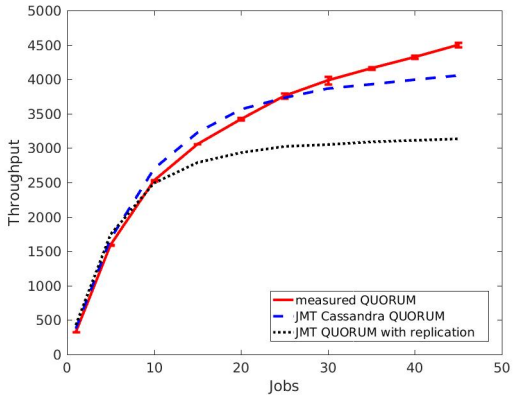

Figure 10: Mean throughput of a Cassandra cluster and the queueing model with Cassandra QUORUM and the real QUORUM.

plied with the average size of the data stored in the database or query network packet size.

- Estimate the YCSB CPU demand analysing the network packet exchanged between the client and a node. We measure the latencies that the client takes to generate a new request when a complete one is received.
- Estimate the node CPU demand for local parsing operations sniffing the network communication of a node. The packets are then analysed to measure the latencies from the instant which a client query arrives to the node to the one in which the node sends the remote incoming request to another machine. This estimation process is the same also for the remote parsing class. These two classes are distinguished by the number of requests that the node generates. The local requests generate one request less than the remote since a copy of the asking data is stored on the local disk of the node.
- Estimate the node CPU demand for local end operations sniffing the network communication of a node. The packets are then analysed to measure the time that a node takes from the instant which the asking data arrives back to the node until when the response to the client is sent. In case of CL ONE the demand for this class is set to zero. The same process is used for the CPU estimation demand of the remote end class.
- Estimate the node CPU demand for remote incoming requests sniffing the network communication of a node. The packets are then analysed to measure the time that a node takes from when a data request comes to the server to when the request is completed and it is sent back to the asking node.

All the CPU demands are calculated with a single job in the system and the collected latencies are processed with the Complete Information algorithm [26] to obtain the final demand for each class. The model is parameterised with the calculated demands for all the classes and, according to the real implementation, the same number of CPU cores are set for each node.

ScyllaDB results, measured and simulated, are shown in the Figures 8 and 9. They represent, respectively, the throughput and the response time of the system. For this case study, only the QUORUM case is taken in consideration. Comparing ScyllaDB with Cassandra QUORUM results, it is possible to observe that ScyllaDB is able to achieve better performance. We observed that, differently from the Cassandra scenario, at high load the workload generator network represents the bottleneck of this set up. The results provided by the simulation tool are close to the measured one and the average relative error is 8% for the throughput and 12.9% for the response time. With this case study, we demonstrate the versatility of our model and the possibility to apply this model to other NoSQL database with minimal adaptations.

5.3 A What-If Scenario: the Impact of Query Replication in Cassandra

We conclude by illustrating an application of the proposed model. Recent research have highlighted the potential advantages of replicating requests in systems in order to consistently achieve low latencies [27]. We here consider a variant of the proposed model to assess the potential impact of query replication on Cassandra.

In the default configuration, when the CL is set to QUORUM, Cassandra generates a fixed number of remote requests. The number of remote requests depends on the RF applied to the database and is equal to $(RF/2)+1$. We now consider a variant of this approach that generates RF requests, but where only the first $(RF/2)+1$ responses needs to be synchronised before completing the query. In this way, the system neither needs to apply any decision on which nodes to send the requests to nor needs to keep track of node performance. Clearly, we expect a trade-off between the enhancement of performance obtained from the higher parallelism and the heightened contention placed on physical resources.

In order to assess this scenario, we have devised an extension of the JMT queueing simulator which allows to synchronise a subset of requests at join nodes. Requests that reach a join after this has completed the synchronization are ignored in the computation of performance metrics. This extension is now integrated in the stable release of JMT version 0.9.3 under the name of *Quorum Join Strategy*. Figure 10 illustrates simulation results. We here compare real Cassandra measurements, with the proposed Cassandra QUORUM mechanism proposed in the previous sections, and the real QUORUM that refers to the replication mechanism. We notice that with a very light load, the throughput under replication is nearly identical to the one without replication. However, at higher loads Cassandra is expected to suffer on average a performance loss compared to the system without replication. This simulation results seem to indicate that increasing replication levels might not be beneficial in a system like Cassandra. While experimental evidence would be needed to corroborate this prediction, this example provides intuition on the ease of performance analysis that comes with the proposed simulation models.

6. CONCLUSION AND FUTURE WORK

In this paper we have proposed a novel queueing network model for the Apache Cassandra database system. Simulation-based analysis shows that our model is able to predict with an error below the 10% the throughput and response time of the system under different consistency levels and number of concurrent requests. The proposed model appears generally more accurate than existing models in the literature that also consider the YCSB benchmark for validation purposes. As part of this paper, we also demonstrate the applicability of our model to another NoSQL database.

As future work, we are planning to evaluate our model with other Cassandra configurations involving a different number of nodes, different data sizes stored in the database and different replication factors. In addition, we are considering to analyse the extended queueing network, possibly LQNs, in order to progress from a simulation model to an analytic evaluation based on mean value analysis algorithms. This would allow computationally-efficient analysis, compatible with usage in nonlinear optimisation programs, which are commonly used for capacity planning under performance and reliability guarantees.

7. REFERENCES

[1] E. Anderson, X. Li, and M. Shah. What consistency does your key-value store actually provide. *Proceedings of the Sixth international conference on Hot topics in system dependability. USENIX Association*, pages 1–16, 2010.

[2] M. Bar-Sinai. Big Data Technology Literature Review. *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing - STOC '97*, 2013(April 2013):654–663, jun 2015.

[3] F. Bause. Queueing Petri Nets-A formalism for the combined qualitative and quantitative analysis of systems. In *Proceedings of 5th International Workshop on Petri Nets and Performance Models*, number class 1, pages 14–23. IEEE Comput. Soc. Press, 1993.

[4] D. Bermbach and S. Tai. Eventual consistency. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing - MW4SOC '11*, number May, pages 1–6, New York, New York, USA, 2011. ACM Press.

[5] M. Bertoli, G. Casale, and G. Serazzi. Jmt: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009.

[6] R. Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12, 2011.

[7] A. Chebotko, A. Kashlev, and S. Lu. A Big Data Modeling Methodology for Apache Cassandra. *2015 IEEE International Congress on Big Data*, pages 238–245, 2015.

[8] H.-e. Chihoub, S. Ibrahim, G. Antoniu, and M. S. Perez. Harmony: Towards Automated Self-Adaptive Consistency in Cloud Storage. In *2012 IEEE International Conference on Cluster Computing*, volume 2012, pages 293–301. IEEE, sep 2012.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, page 143, New York, New York, USA, 2010. ACM Press.

[10] Datastax. Cassandra documentation, jun 2016.

[11] ESnet. Iperf, jun 2016.

[12] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced Modeling and Solution of Layered Queueing Networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, mar 2009.

[13] A. Gandini, M. Gribaudo, W. J. Knottenbelt, R. Osman, and P. Piazzolla. Performance Evaluation of NoSQL Databases. In *Computer Performance Engineering*, pages 16–29. 2014.

[14] P. Garefalakis, P. Papadopoulos, and K. Magoutis. ACaZoo: A Distributed Key-Value Store Based on Replicated LSM-Trees. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 211–220. IEEE, oct 2014.

[15] Intel. Intel next generation microarchitecture (nehalem), jun 2016.

[16] Intel. Intel sse4 programming reference, jun 2016.

[17] S. D. Kuznetsov and A. V. Poskonin. NoSQL data management systems. *Programming and Computer Software*, 40(6):323–332, nov 2014.

[18] A. Labrinidis and H. V. Jagadish. Challenges and opportunities with big data. *Proceedings of the VLDB Endowment*, 5(12):2032–2033, 2012.

[19] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35, apr 2010.

[20] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[21] M. Lord. hdparm, jun 2016.

[22] J. R. Lourenço, V. Abramova, M. Vieira, B. Cabral, and J. Bernardino. NoSQL Databases: A Software Engineering Perspective. In A. Rocha, A. M. Correia, S. Costanzo, and L. P. Reis, editors, *Advances in Intelligent Systems and Computing*, volume 353 of *Advances in Intelligent Systems and Computing*, pages 741–750. Springer International Publishing, Cham, 2015.

[23] R. Niemann. Towards the Prediction of the Performance and Energy Efficiency of Distributed Data Management Systems. In *Companion Publication for ACM/SPEC on International Conference on Performance Engineering - ICPE '16 Companion*, pages 23–28, New York, New York, USA, 2016. ACM Press.

[24] R. Osman and W. J. Knottenbelt. Database system performance evaluation models: A survey. *Performance Evaluation*, 69(10):471–493, 2012.

[25] R. Osman and P. Piazzolla. Modelling Replication in NoSQL Datastores. In *Quantitative Evaluation of Systems*, pages 194–209. Springer International Publishing, 2014.

[26] J. F. Perez, S. Pacheco-Sanchez, and G. Casale. An Offline Demand Estimation Method for Multi-threaded Applications. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 21–30. IEEE, aug 2013.

[27] Z. Qiu and J. F. Pérez. Evaluating the effectiveness of replication for tail-tolerance. In *CCGrid*, pages 443–452. IEEE Computer Society, 2015.

[28] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. *Cidr*, pages 134–143, 2011.

[29] XLAB. Scylladb, jul 2016.