Performance and Replica Consistency Simulation for Quorum-Based NoSQL System Cassandra

Xiangdong Huang[1,2(✉)], Jianmin Wang[1,2], Jialin Qiao[1,2], Liangfan Zheng[1], Jinrui Zhang[1], and Raymond K. Wong[3]

[1] School of Software, Tsinghua University, Beijing 100084, China
huangxd12@mails.tsinghua.edu.cn, jimwang@tsinghua.edu.cn
[2] National Engineering Laboratory for Big Data System Software, Beijing 100084, China
[3] School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
wong@cse.unsw.edu.au

Abstract. Distributed NoSQL systems such as Cassandra are popular nowadays. However, it is complicated and tedious to configure these systems to achieve their maximum performance for a given environment. This paper focuses on the application of a Coloured Petri Net-based simulation method on a quorum-based system, Cassandra. By analyzing the read and write process of Cassandra, we propose a CPN model, which can be used for performance analysis, optimization, and replica consistency detection. To help users understanding the NoSQL well, a CPN-based simulator called QuoVis is developed. Using QuoVis, users can visualize the read and write process of Cassandra, try different hardware parameters for performance simulation, optimizing system parameters such as timeout and data partitioning strategy, and detecting replica consistency. Experiments show our model fits the real Cassandra cluster well.

1 Introduction

By facilitating high availability for large volumes of data, distributed NoSQL systems have been becoming popular nowadays. For example, Apple company stores over 10 PB of data with over 75,000 Cassandra nodes, and Uber company stores all the GPS data of cars with about 300 Cassandra nodes. Many such systems, e.g., Cassandra [19], Riak [8] and Voldemort [11], are designed as quorum-based systems [28] to support high availability and guarantee replica consistency.

To support different workloads in different environments, NoSQL systems usually have many parameters for users to set. Setting these parameters correctly may have a huge impact on the system performance. For instance, Cassandra (v2.0) has more than 70 parameters, and users may need to tune many of them, e.g., concurrent writes, timeout value and memtable size, to achieve the best performance for their environment; while tuning the I/O parameters in HBase can improve the throughput several times [4].

W. van der Aalst and E. Best (Eds.): PETRI NETS 2017, LNCS 10258, pp. 78–98, 2017.
DOI: 10.1007/978-3-319-57861-3_6

However, tuning the system parameters can be complicated and tedious because it requires users to modify the configuration, then restart and benchmark the system. This process needs to repeat until the targeted performance is achieved. For example, to determine the best table size in memory in Cassandra (i.e., `max_memtable_threshold`), users need to try a particular memory size and apply this new size to all the nodes, then restart the cluster and run some benchmark tests. It needs to repeat until the overall performance is satisfied. The above steps cost a lot of time: just starting up an n-nodes Cassandra cluster, we need at least $2n$ minutes (called as "two minutes rule"[1]). Besides, users have to care about the "cold start" problem in their tests. Worst still, if users are not experts of the target system, they may be confused about what the bottleneck is under the current system configuration or why some exceptions happen (e.g., violating "two minutes rule").

Furthermore, when evaluating or analyzing complex features what NoSQL systems claimed, such as strong consistency guarantee, users or developers need to cover as many as possible situations. However, it is difficult to cover all possible scenarios in a given environment for evaluation purposes. For example, the convergence speed of replicas (from inconsistency to consistency) can be impacted by the network conditions. However, it is almost impossible to adjust the network settings to simulate all possible network performances. Finally, there is no easy way to observe the consistency and monitor the running status of the cluster or specific data values in real-time. Users have to collect distributed logs to track the running trace of the cluster and this method has some fundamental limitations [21].

To address the above issues, we propose building a Coloured Petri Net (i.e., CPN) model for simulating quorum-based NoSQL databases, and then using the model to evaluate and analyze the system. In the paper, we consider Cassandra as an example. Firstly, we analyze the read and write process of Cassandra, and then build a CPN model for the system. The model abstracts the data partitioning strategy, data transitions, network transfer and quorum-based replica control protocol. Secondly, we propose how to use the model for performance evaluation, consistency analysis and evaluation.

To help users (who have little knowledge about CPN) understanding the work process of quorum-based NoSQL systems and tuning them, we developed a visual simulator for Cassandra, called QuoVis. QuoVis can simulate the local data transition in a node, data propagation between nodes. It provides a set of plugins specifically for tuning and analyzing Cassandra. For example, some default plugins allow users to simulate the hardware performance (such as the time cost of each step in a read or write process), and modify the system parameters (such as data partitioning strategy, timeout, etc.). These plugins help us to simulate cluster behavior, especially during message communications, in different scenarios. Based on the plugins, users can simulate and determine appropriate system parameters for a particular production environment. There are also other more specific plugins for more advanced usages. For example, one of them is to detect whether the replicas are consistent at a particular time, or whether a

[1] https://issues.apache.org/jira/browse/CASSANDRA-2434.

protocol can make the replica consistent eventually in a given system configuration. Additional plugins can be built by users and added to the simulator to support more analysis ability.

To the best of our knowledge, the model for Cassandra is the first CPN model for NoSQL systems which provides many analysis applications. QuoVis is the first simulator tool for NoSQL clusters. Though the CPN model is built based on Cassandra, we believe that it is also useful for guiding readers to build models for other quorum-based systems, e.g., Riak and Voldemort, and other NoSQL systems.

2 Related Works

Petri Net has been used in many applications. For example, Quentin Gaudel et al. proposed using Hybrid Particle Petri Nets to diagnose the health status of a hybrid system [12]. Petri Net has also been used in the molecular programming area, such as verifying DNA walker circuits using Stochastic Petri Nets [5]. These applications show the power of Petri Net.

As for big data system performance optimization and correction verification areas, there are two kinds of methods: modifying parameters and workloads on real clusters for performance tuning, or using simulation results to analyze the real system. For example, to optimize the performance of jobs on MapReduce, MRTuner [26] was proposed. MRTuner gave a new cost model which considering the holistic optimization, and designed a fast search method to find the optimal execution plan. Similarly, Apache Spark was also optimized by similar methods [25]. Other works, such as Starfish [15] was also proposed. Some simulation works using PeerSim [23,30], Gloudsim [10] and others. However, all of these works do not consider recent popular NoSQL systems as targets.

Though many works were done using (Coloured) Petri Net for system evaluation [20,22,29], using Petri Net to model recent big data systems is still emerging [1,24]. HDFS was modeled in [1] while Cassandra was targeted in [24]. [1] proposed using a CP-net to build the behaviour of HDFS. Using this model, a special scenario, which data nodes leave from the cluster and join into the cluster back frequently, was discussed. In [24], Osman et al. used a QP-net to model the replica of Cassandra. They used the model to evaluate the quorum mechanism. However, the model lacks of ability for analyzing some different consistency metrics of NoSQL, such as data-centric and client-centric consistency metrics [6]. Comparing with QP-net, CPN is more suitable for analyzing the replica consistency. It is because that in the CPN model, users can observe the values of tokens, so that it is easy to determine whether the replicas of a data item are consistent or not. Besides, the model does not fit Cassandra well in some processes. For example, Cassandra splits its reading process as "read digests" and "read original values", while the model does not consider that.

The Quorum-based system has a long history. Quorum can be traced back to 1979 [13], and now many NoSQL systems use it. In brief, suppose one data item has k replicas. Quorum-based systems allow only w replicas to update successfully, and r replicas return responses for a reading request. If $w + r > k$, then clients can read the latest version of the data item.

3 The CPN Model

In this section, we first investigate the write process of Cassandra, and then build a CPN model according to that.

3.1 Writing Operation

We use Cassandra as an example to describe the writing operation. The reason that we choose Cassandra is that it combines the design of Dynamo-like and BigTable-like NoSQL systems [19] and it is popular in the world. We have used Cassandra solving the data management problems for meteorological data and time series data.

In a Cassandra cluster, each data item has k replicas. k is configurable. Therefore, given a writing request, there are at least k servers will be involved. We define 3 kinds of nodes (i.e., servers) for each request: *coordinator*, *participator* and *other*s. The coordinator is the node which the client sends the request to. It is also the node which gives the response to the client. The participator is the node which receives a forwarded request. Except for the coordinator, all the nodes which have replicas of the requested data item are participators. Other nodes in the cluster which do not handle the query or writing operation are others. For a request, we only care about the coordinator and participators.

Figure 1 shows a Cassandra cluster with 5 nodes ($s_1 \sim s_5$) and each data item has two replicas: s_1 and s_2 store item A. A client sends a writing request for A to s_1. In this case, s_1 is the coordinator and s_2 is a participator. Both $s_3 \sim s_5$ are other nodes.

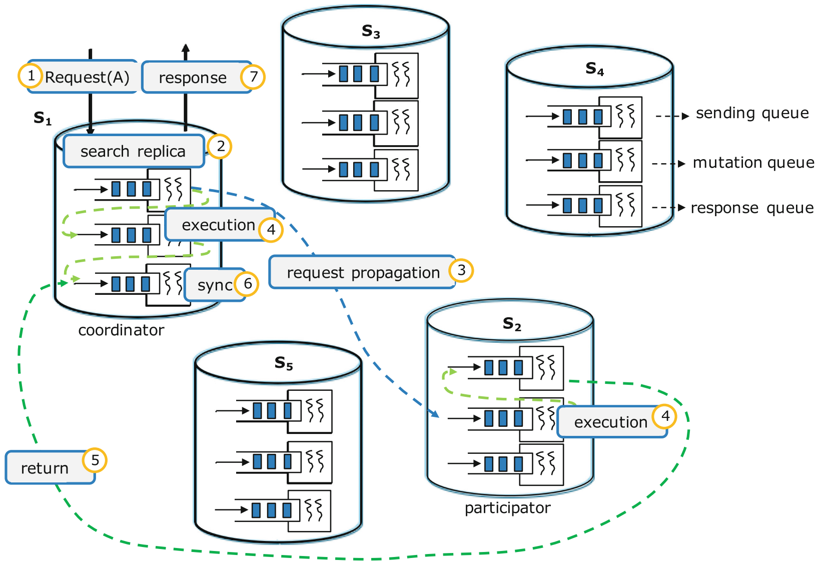


Fig. 1. Writing operation between two nodes in a 5-nodes Cassandra cluster

Figure 1 splits a writing operation of Cassandra into 7 steps (① ∼ ⑦). There are two branches: a *remote branch*, which is ① → ② → ③ → ④ → ⑤ → ⑥ → ⑦; and a *local branch*, which is ① → ② → ④ → ⑥ → ⑦. If the replica number $k > 1$, then at least one remote branch will be fired. If the coordinator has one replica of the requested data item, the local branch will be fired. Otherwise the local branch will be not fired. A detailed description about these steps can be found in our previous work [16] and Osman et al.'s work [24] has a similar discussion.

⑥ is for synchronizing writing acknowledgements. In Cassandra, for each writing request, it will be propagated to k servers (which the coordinator may be one of), but the coordinator only accepts w ($w \leq k$ and w is also user-defined) writing acknowledgements. That is why Cassandra claims it is a consistency-tunable system.

Because requests may come in faster than they can be digested, a temporary space to cache the requests is necessary. For each node-to-node connection, the coordinator has a *propagation queue* to cache the requests for sending them out. Except for the propagation queue, each node has a *mutation queue* to cache the requests from clients or coordinators for executing local writing operations. Similarly, each node has a *reading queue* for reading operations. After finishing the reading or writing operation locally, the node sends a response message to the coordinator. The step is also handled by the propagation queue. To handle received response messages, a *response queue* is needed.

3.2 Query Operation

The query operation is similar to the writing operation. To describe a query operation step by step, we just need to rename the "execution" in Fig. 1 as "query", and modify the time costs of transitions. In detail, a query process has two stages: read data digest, and read data values. Some works such as [24] do not consider that. We have implemented the two stages with CPN language[2].

The outputs of "search replica" and "sync" also need to be modified. "search replica" in the writing operation generates k tasks, while only r tasks are generated in the query operation ($r \leq k$ and r is a user-defined value). Similarly, "sync" in the writing operation only accepts w response messages while it needs r response messages for query operations.

3.3 CPN Model

Model with 2 Nodes. Firstly, we use a CP-net to describe the two branches in Sect. 3.1. When converting the two branches to a CP-net model, we can use a multi-choice pattern [9] to represent ②. Similarly, ⑥ is a synchronizing merge pattern [9] in CP-net form. Other steps can be modeled by a sequence pattern [9].

[2] How to model the two stage is not shown in this paper, but the source code of the model and all the source codes of CPN models in this paper can be found from the Github website: https://github.com/jixuan1989/color-petri-net-cassandra.

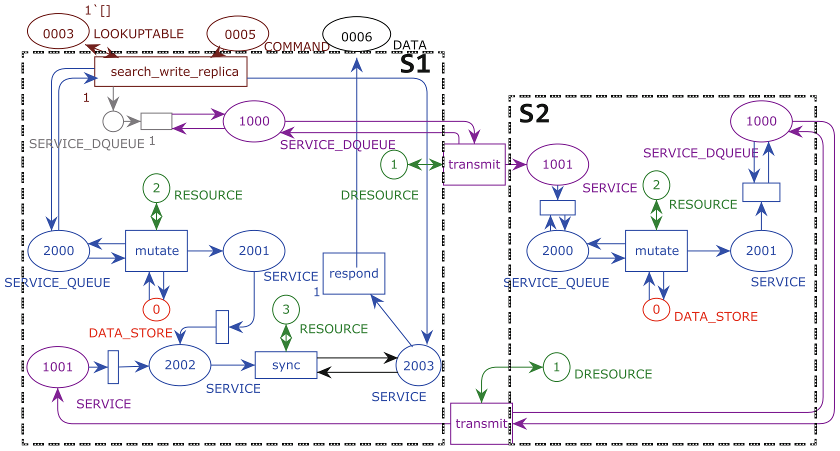


Fig. 2. Using CP-net to describe a writing operation between two nodes

Figure 2 shows a CP-net model for a writing operation instance between two nodes. In the figure, "search write replica" transition represents step ②, "transmit" transition represents step ③ or ⑤, "mutate" is step ④, "sync" is step ⑥, and "respond" is step ⑦. ① is a token in the "0005" place. The definitions of col_set, expressions and functions are omitted here, and users can get them in a ".cpn" file from Github.

Cassandra uses consistent hashing ring [18] as its data partitioning strategy [19]. The tokens in the "0003" place represent the information of the consistent hashing ring. In our model, the ring is modeled using a numeric space K, e.g., [0,100]. Each node is responsible for some ranges of the space, e.g., node s_1 may be responsible for ranges [40,50) and [50,60).

The data item is modeled as a triple $item = (key, value, version)$, where $hash(key) \in K$. A request is a quintuple $req = (type,\ key,\ value,\ version,\ clevel)$ where $type$ is `READ` or `WRITE`, and $clevel$ is the required consistency level (i.e., the value of the quorum parameter w or r). So, if a client sends a request $(\texttt{WRITE}, 45, 3, 2, 1)$, i.e., the key is 45, value is 3, version is 2 and the required consistency level is $w = 1$, then the request will be propagated to node s_1 and other replicas.

Tokens in the "2003" are callbacks which count how many acknowledgements that have been received. Tokens in the "0006" place represent the results that clients receive. Tokens in the "0" place represent the persistent data items on disk in Cassandra nodes.

As described in Sect. 3.1, there are 3 queues in our system. In the figure, place "1000", "2000" and "2003" are the *sending queue*, the *mutation queue* and the *response queue*. We implement the queues by the `List` in the CPN Tools. By using `hd list`, `tl list` and ^^ operations, we can implement the FIFO strategy of a queue[3]. That is why the arcs between the place "2000" (or "1000", "2003") and

[3] http://cpntools.org/documentation/tasks/editing/constructs/queuesstacks.

its adjacent transitions are round arrows. Tokens in the place “2” are concurrent write threads. If there are t tokens totally in the place “2”, then the place “2”, “2000” and the “mutate” transition form a queueing system which has t service counters. Place “1” and “3” are similar with “2”. In this model, all the anonymous transitions can be simplified by merging its input place and output place. We draw them for generating the folded model in the rest of this paper.

Modeling a Cluster. Next, we extend Fig. 2 to describe a cluster. We use a 5-nodes cluster as an example. By copying the model structure of s_1 (s_2 is a sub-model of s_1’s structure), we get CP-net structures for $s_2 \sim s_5$. As shown in Figure 3, we organize the structures of $s_1 \sim s_5$ together by connecting the “input” (i.e., “1001”) and “output” (i.e., “1000”) places with anonymous transitions.

Figure 3 shows the architecture of a cluster clearly. A client can choose any node as the coordinator. Data will be stored in k “0” places eventually (k is the replica number).

It is hard to use the model in Fig. 3 for modeling arbitrary sized clusters. For example, to simulate a cluster with n nodes, we need to construct a model which is similar with Fig. 3 by copying, pasting and some minor modifications. To solve the problem, a folded Petri Net model is built based on Fig. 3. Figure 4 shows

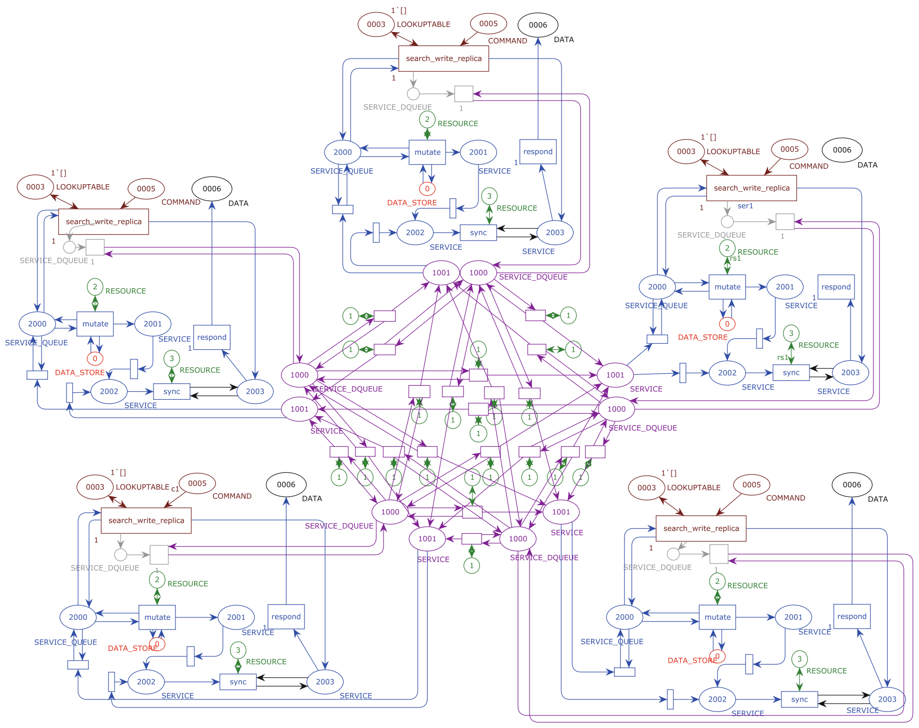


Fig. 3. Replicating two nodes to a cluster with 5 nodes

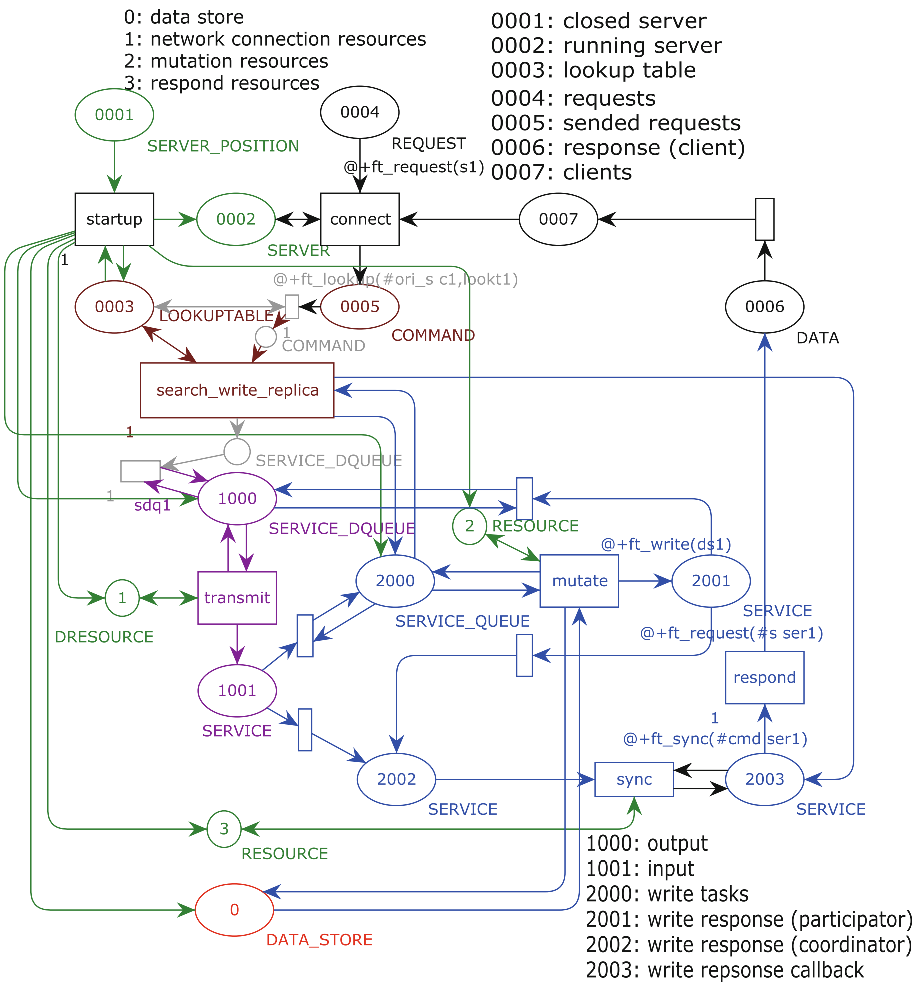


Fig. 4. A folded model for represent a cluster with any number of nodes (Color figure online)

the model. The model is drew by CPN Tools and users can download the source code of the model from Github. The elements (i.e., places and transitions) tinged with green color refer to the initiation process of servers, so that many resources are initialized. The dark color elements refer to determining which nodes having replicas. The purple color elements refer to the network propagation. The blue elements refer to the local write process and the synchronization process.

The model defines 15 col_sets and 16 complex functions (except for simple expressions on arcs). To make the model readable, we omit the expressions or functions on arcs, the guard functions on transitions, and the definition of col_sets. However, the time costs of important transitions are reserved. Place 0001, 0003, 0004, and 0007 have initial tokens. Users can read detailed model on Github.

4 Model Analysis

4.1 Performance Simulation

In CPN Tools, transitions in a model can have a property, time cost. It is introduced by Timed Petri Net. Suppose a transition has that property, and the time cost is t, then it means that when the transition is fired, it generates the output tokens after t time units. Using that property, we can simulate the time cost of a write process in the real cluster. Before simulation, users need to define the time costs of each transition in our model.

To simulate a cluster with n nodes using the model in Fig. 4, we just need to put n tokens with different values into the place "0001". In default, the consistent hashing ring will be generated automatically and stored in the place "0003" when the transition "startup" is fired. To simulate m requests, we need to put m tokens into the place "0004". After the simulation, Data items (i.e., tokens) will be stored in the place "0".

Suppose $\sigma_r(t)$ is the number of tokens in the place "0004" at time t (the clock is controlled by the CP-net model rather than a real clock in the real world), and $\sigma_i(t)$ is the number of tokens which belong to the node s_i in the place "0" at time t. Then the throughput T_{s_i} of the node s_i between time t_1 to t_2 is:

$$T_{s_i} = \frac{\sigma_i(t_2) - \sigma_i(t_1)}{t_2 - t_1}$$

The throughput T of the cluster is:

$$T = \frac{\sigma_r(t_2) - \sigma_r(t_1)}{t_2 - t_1} \tag{1}$$

Given a request, we mark that its related token d is put into the place "0004" at time $t_{req}(d)$, and generated into the place "0" at time $t_{res}(d)$. Then given a request set R, the average latency of requests is:

$$L(R) = \frac{\sum_{r \in R} (t_{res}(r) - t_{req}(r))}{|R|} \tag{2}$$

4.2 Parameter Tuning

A model which can be used for tuning parameters needs to have two characteristics: (1) the model can reflect the current configurations of a real system; (2) the model can tune parameters which users focus on. In our model, the first characteristic is implemented by the settings of time costs of each transition: we collect system log from real systems and then analyze the time costs of each events; then we use the time costs as transitions' time costs. After that, the model can reflect the performance of the real system well. In this way, though we do not know the specific values of all the parameters, the model has the ability to reflect the impact of current parameters' values.

As for the second characteristic, the model in this paper can only be used for tuning (1) the numbers of the concurrent write threads, the network connections, and concurrent synchronization threads; (2) the timeout parameter; (3) different workloads and quorum settings; and (4) data partitioning configuration. The first one can be achieved by modifying the number of tokens in the place "2", "1" and "3". The second one is not shown in Fig. 4. Users can find the related transitions in a more complex model on Github. The third one can be achieved by adding different tokens in the place "0004". By setting the tokens with different time values, we can control the number of concurrent requests. By setting the tokens with different "key" values, we can control the data distribution of requests. By setting the tokens with different *clevel* values, we can control the quorum settings. The 4th one can be achieved by modifying tokens in the place "0003".

4.3 Consistency Detection

By checking all the tokens in the place "0" on each servers, we can detect whether there is replica inconsistency easily. If a data item has two replicas, then it needs to be stored on the places "0" of two servers. Given a data item (key, value, version)= (k, v, vr), we suppose the replicas of it are on server s_i and s_j, and the related tokens in the places 0 of s_i and s_j are $token_i = (k, v_i, vr_i)$ and $token_j = (k, v_j, vr_j)$. then there are 3 situations:

(1) Neither $token_i$ or $token_j$ exists, which means both the two servers s_i and s_j have not finished the writing operation.
(2) $token_i$ (or $token_j$) does not exist, which means s_i (or s_j) has not finished the writing operation while s_j (or s_i) has finished it. If $token_i$ does not exist, we consider that s_j has the latest version of the data item. Otherwise s_i has the latest version.
(3) $v_i \neq v_j$ or $vr_i \neq vr_j$, which means the two replicas are not consistent. If $vr_i > vr_j$, then s_i has the latest version, otherwise s_i has a stale version.

Figure 5 shows an example of the third situation. In the figure, we omit the irrelevant part of the model. The data item whose key is b has two

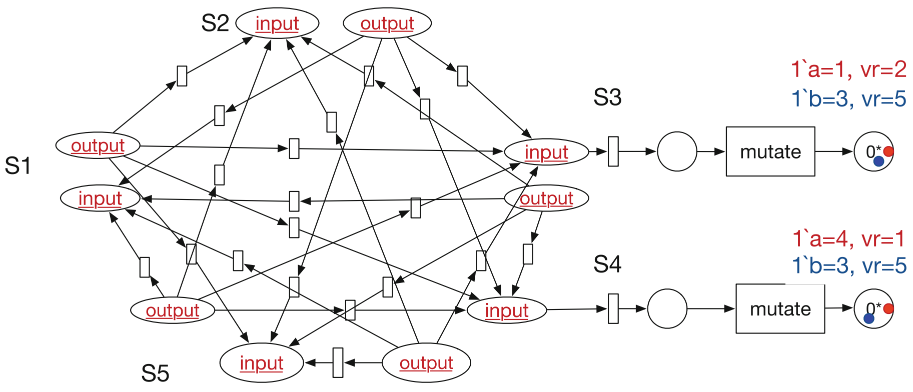


Fig. 5. An example of inconsistent data items

consistent replicas. The data item whose key is a has two inconsistent replicas. Because data item a on server s_4 has a smaller version than s_3, we consider that the data item on s_4 is more stale. The reason of the inconsistency may be that server s_4 lost one or more writing operations for the data item.

Detecting the consistency of replicas is meaningful because it helps us accelerating the reading process. Normally, Cassandra provides eventual consistency for high availability and low latency [2]: as described in Sect. 2, if $w + r > k$, where k is the number of replicas, users can get a strong consistent result. However, for each reading requests, it requires the cluster reads r replicas, where the $r - 1$ reading results are meaningless if all of the replicas are consistent. Therefore, if we know the replicas are consistent, we can set $r = 1$ to accelerate reading processes while maintaining the strong consistency property. Therefore, monitoring whether the replicas are consistent and when they become consistent are important.

By monitoring the changes of tokens of place 0 on all the servers, we can observe the inconsistency of all the data items. It is easy to be implemented by CPN language and CPN Tools, while some other Petri Net languages which do not have the concept of "colour" are hard to support it. That is one of the main reason that we choose CPN.

4.4 Consistency Measurement

Because of the importance of replica consistency, many consistency metrics are proposed. From the client-centric view, there are k-staleness, t-visibility [3], session consistency [27] and so on. These metrics are a little easier than data-centric metrics, it is because they consider the target systems as black boxes. From the data-centric view, there are atomicity, regularity, and safeness [14]. Besides, inconsistency time window is also proposed in [7,16] respectively, while the definitions of them are slightly different. In this paper, we use the CPN model to show what the inconsistency time window in [16] is and how to measure it.

Inconsistent time window: It refers to the duration time of the inconsistent between two replicas. Given a writing operation $W[x]1$ (i.e., set item $x = 1$), and the item x has two replica s_i and s_j, if s_i finished $W[x]1$ at time t_i while s_j finished it at t_j, we say the inconsistent time window of s_i and s_j on data item x is

$$itw_x(s_i, s_j) = |t_i - t_j|. \tag{3}$$

Figure 6 shows an example. Similar with Fig. 5, the irrelevant part of the model is omitted. The figure is a snapshot that the blue and red tokens $token_b$ and $token_r$ have been written successfully in server s_4 at time t_0. At that time, $token_b$ and $token_r$ were not written in server s_3. Therefore, it belongs to the second inconsistency situation. There are two cases: the token ($token_r$) has enqueued the mutation queue, but it is waiting for enjoying service; the token ($token_b$) has not been propagated to the server (s_3). By monitoring the changes of place 0, we will know the exact time that the tokens are written successfully. Then we can calculate the inconsistency time window by Eq. 3.

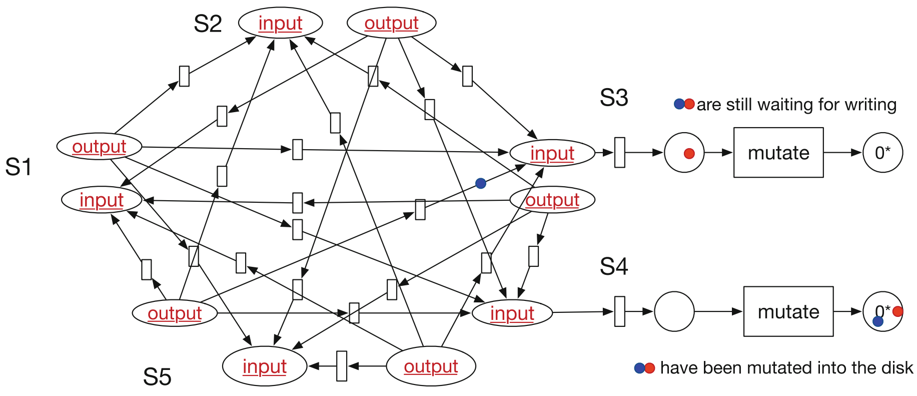

S1
S2
S3
S4
S5
input
output
mutate
0*
are still waiting for writing
have been mutated into the disk

Fig. 6. An example of inconsistent time window (Color figure online)

5 Application: QuoVis

To help users understanding the read and write process of Cassandra well, we build a visualization simulator[4]. The tool is driven by our CPN model and CPN Tools's execution engine (by using Access/CPN library [31][5]). Besides, the tool provides some analysis plugins for helping users simulating and tuning the performance, and analyzing the consistency.

We firstly introduce the architecture of QuoVis. Then we describe the basic simulation and visualization of QuoVis. Then we demonstrate 3 analysis plugins. The first plugin shows how to simulate the hardware performance and tune the system parameters with QuoVis (Sects. 5.3 and 5.4). The second plugin helps users to choose a better data partitioning strategy by quantitatively comparing the performance difference (Sect. 5.5). With the third plugin, we illustrate how to use the model for consistency detection (Sect. 5.6). Besides, we discuss about that whether the quorum protocol can always guarantee eventual consistency in Cassandra.

5.1 System Design

QuoVis is designed based on a modular architecture with an extensible plugin interface, as shown in Fig. 7. The underlying model is the kernel of the simulator. The simulator engine has the ability to execute the model step-by-step and can provide the runtime information to other modules. When executing the model, the simulator maintains a virtual global clock (i.e., the time in CPN Tools execution engine) to simulate the time elapsed for each step in a read or

[4] A demo video is available from https://www.youtube.com/watch?v=nvIBDUubFvM.

[5] We found the performance is terrible for simulating a large scale clusters, and thereby we have built a faster execution engine. The engine is not the focus of this paper so that we omit it.

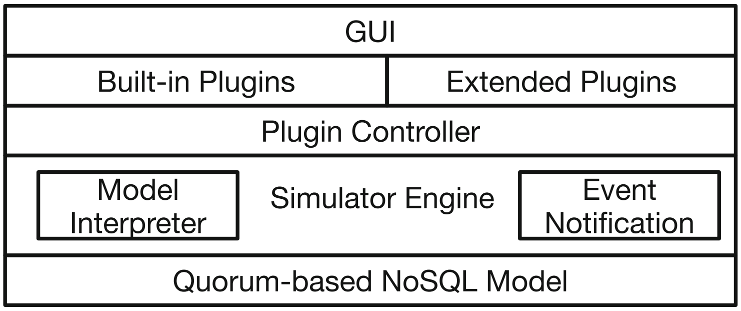


Fig. 7. QuoVis system architecture

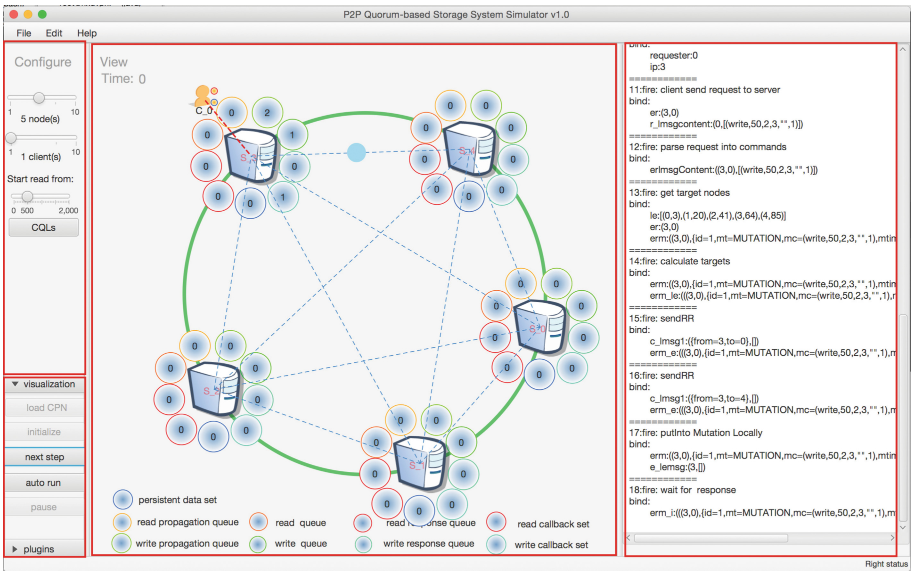
write process. The elapsed time is irrelevant to the real-world time. Hence, the simulation result is independent on the performance on the underlying hardware which QuoVis runs on.

The abilities of tuning Cassandra parameters and other analysis are provided as plugins. Some built-in plugins have been developed. Users can create new plugins to extend the built-in abilities. Plugin controller is responsible for plugin registration and management. Plugins can be loaded dynamically. To avoid plugins asking information from the simulator frequently, an event notification module is in place to allow plugins to subscribe only the transitions and their binding data values (i.e., tokens) of their interests. The GUI module is designed to visualize the running status of the model. Plugins can also customize the GUI.

5.2 Process Visualization

This section demonstrates how QuoVis simulates a Cassandra cluster with an arbitrary number of nodes and settings, and can be used to visualize each step of a read or write process. A screenshot of the main screen of the simulator is shown in Fig. 8. Users can configure the number of servers, clients and other details on the left pane. Users can customize the reading and writing requests via an advanced menu there. By setting the starting time of the first read request, QuoVis can simulate a cluster which has loaded data before the read workload comes. Users can also control the simulation progress step by step through the simulation, and watch the simulation trace details on the right pane.

The animation of the simulation is shown on the middle pane for visualization purposes. Users can, for example, animate and monitor how the cluster with the consistent hashing ring is constructed and how a read or write process is running. For each node, there are nine circles that represent some important places in the model. Normally, these places represent queues in Cassandra model. Users can observe how many items in a queue and even obtain the details of the items there when the mouse pointer is placed on top of its corresponding circle. When a message moves from one queue to a queue of another node, an animation of a small blue bubble moving from its corresponding circle to another circle will be played to reflect this message propagation. The blue bubble represents a token



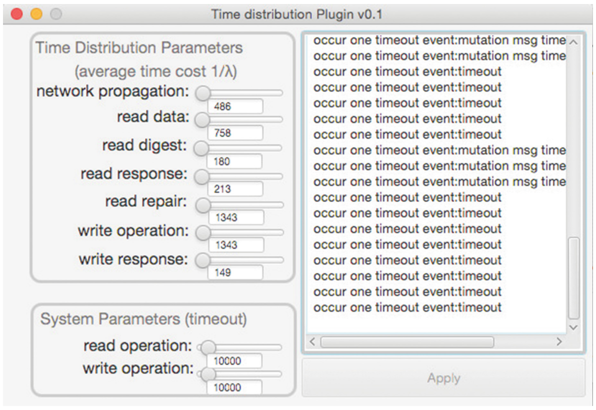
```
        requester:0
        ip:3
============
11:fire: client send request to server
bind:
        er:(3,0)
        r_lmsgcontent:(0,[(write,50,2,3,"",1)])
============
12:fire: parse request into commands
bind:
        erlmsgContent:((3,0),[(write,50,2,3,"",1)])
============
13:fire: get target nodes
bind:
        le:[(0,3),(1,20),(2,41),(3,64),(4,85)]
        er:(3,0)
        erm:((3,0),{id=1,mt=MUTATION,mc=(write,50,2,3,"",1),mtin
============
14:fire: calculate targets
bind:
        erm:((3,0),{id=1,mt=MUTATION,mc=(write,50,2,3,"",1),mtin
        erm_le:(((3,0),{id=1,mt=MUTATION,mc=(write,50,2,3,"",1),r
============
15:fire: sendRR
bind:
        c_lmsg1:({from=3,to=0},[])
        erm_e:(((3,0),{id=1,mt=MUTATION,mc=(write,50,2,3,"",1),m
============
16:fire: sendRR
bind:
        c_lmsg1:({from=3,to=4},[])
        erm_e:(((3,0),{id=1,mt=MUTATION,mc=(write,50,2,3,"",1),m
============
17:fire: putInto Mutation Locally
bind:
        erm:((3,0),{id=1,mt=MUTATION,mc=(write,50,2,3,"",1),mtin
        e_lemsg:(3,[])
============
18:fire: wait for  response
bind:
        erm_i:(((3,0),{id=1,mt=MUTATION,mc=(write,50,2,3,"",1),m
```

Fig. 8. The main screen of the simulator (Color figure online)

in our model. Overall this simulation and visualization demonstrate the most common application of our simulator.

5.3 Performance Simulation

When a user uses the simulator to simulate a NoSQL cluster in a production environment, the user needs to input the hardware parameters to describe the production environment at first. In QuoVis, we use the time costs of each step to describe hardware performance. For example, if we want to use QuoVis to simulate and analyze a Cassandra cluster on Amazon EC2 with 5 *t2.small*



Time Distribution Parameters (average time cost 1/λ)

network propagation:	486
read data:	758
read digest:	180
read response:	213
read repair:	1343
write operation:	1343
write response:	149

System Parameters (timeout)

read operation:	10000
write operation:	10000

```
occur one timeout event:mutation msg time
occur one timeout event:mutation msg time
occur one timeout event:timeout
occur one timeout event:timeout
occur one timeout event:timeout
occur one timeout event:timeout
occur one timeout event:timeout
occur one timeout event:mutation msg time
occur one timeout event:mutation msg time
occur one timeout event:mutation msg time
occur one timeout event:timeout
occur one timeout event:timeout
occur one timeout event:timeout
occur one timeout event:timeout
occur one timeout event:timeout
occur one timeout event:timeout
occur one timeout event:timeout
occur one timeout event:timeout
```

Fig. 9. The plugin for tuning time cost and timeout.

servers, we can first run Cassandra on the cluster and then sample some system logs to calculate the time costs. The top left of Fig. 9 shows the time cost of each step on the cluster.

After getting the time costs, users can use QuoVis to simulate Cassandra on Amazon EC2. For example, we can use the model for performance evaluation. Figure 10 demonstrates the throughput simulation results with different cluster sizes by the CP-net model. The results are normalized to $[1, \infty)$ by dividing the throughput result by that of a 5-node cluster. That is, if the throughput of a 5-node cluster is 1000 ops/s, then the throughputs of a 10-node, 15-node and 20-node clusters are 1190, 1210 and 1350.

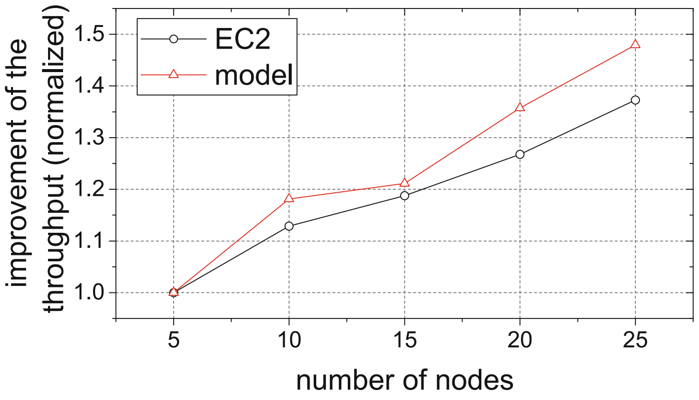


Fig. 10. Throughput prediction of clusters with different sizes and a comparison with real systems

In this experiment, the time distributions are collected and calculated by system log in a 5-nodes cluster. When we change the size of the cluster in our model, we do not change the time distributions of each transition. The above approach may make some errors to our simulation results. However, we think the error is minor: by deploying real clusters with different sizes on Amazon EC2, we get the throughput on real clusters. The comparison with the model is also shown in Fig. 10. According to the comparison, the maximum error of the model is about 9% (i.e., the cluster with 25 nodes).

Using our model, we can predict how many nodes an application needs. For example, in Figure 10, suppose the application requirement is 1250 ops/s. Firstly, we can deploy a 5-nodes cluster and collect system log for initializing our model. Then we use the model to predict performance with larger clusters. According to the prediction in Figure 10, we recommend that the cluster needs more than 20 nodes ($1350 > 1250$).

Though we can use the time costs of 5 nodes to simulate a cluster with 25 nodes within 9% errors, it does not means we can simulate larger cluster without modifying the time costs while keeping a small error. For example, if we want to predict the performance of a cluster with 60 nodes, we would better to get the real time costs of each transition on a 30-nodes cluster first. Fortunately, it is acceptable in practice: scaling out a cluster is iterative, and it is unlikely that scaling out a cluster from 5 nodes to 500 nodes one round.

5.4 Parameter Optimization: Timeout

To get the best performance, users need to determine various system parameters such as timeout value, maximum number of threads, etc. We consider the timeout value as a specific example. A high timeout value delays the detection of system exception or node failure, and a low timeout value can misjudge a normal process as an exception that affect the system correctness. Since the cluster is newly setup, users are unlikely to identify suitable values for the system parameters quickly. Instead of trial and error, restart and benchmark the system, it would be desirable to be able to just enter different parameter values and simulate the benchmarks.

With QuoVis, users can easily try different parameter values to simulate different scenarios of a NoSQL cluster. Firstly, users need to initialize the QuoVis model by entering the hardware performance (i.e., the time costs) as above. After the initialization, we can use QuoVis to find the suitable system parameters. For example, in Fig. 9, we set the timeout value to 10000, which leads to too many timeout events. As a result, more than 1/10 of the writing requests fail. Therefore, many write operations have not been successfully replicated in all the replicas, i.e., the replicas are inconsistent. Users can then increase the timeout value until an acceptable number of timeout events is resulted.

5.5 Parameter Optimization: Consistent Hashing Ring

How data items are placed in a cluster can significantly affect the overall performance [17], and users can control the data partitioning strategy by modifying the consistent hashing ring. Using QuoVis, we can easily compare different data partitioning strategies and visualize the results. In this section, we use QuoVis to simulate different data partitioning strategies and predict the throughput. The plugin GUI is omitted here because of the limitation of pages.

To calculate the accuracy of the simulation result, we benchmark Cassandra throughput using 5 data partitioning strategies on Amazon EC2. Figure 11 shows the comparison results. The results are normalized to 0∼1 by dividing the throughput with the first partition strategy $P1$. Prediction from QuoVis is consistent with the throughput of the actual cluster. For example, QuoVis predicts that the throughput of the cluster with $P2$ strategy is about 86% of the throughput from the uniform strategy ($P1$). The result in the real cluster shows that the percentage is about 83%. By observation, the maximum error is about 8% ($P5$).

In addition, we can also use QuoVis to find out the reason that causes the decline of throughput. Via the visual process shown of $P5$ in QuoVis, we found that the mutation queues of node s_1 and s_5 are long and the sizes of the persistent

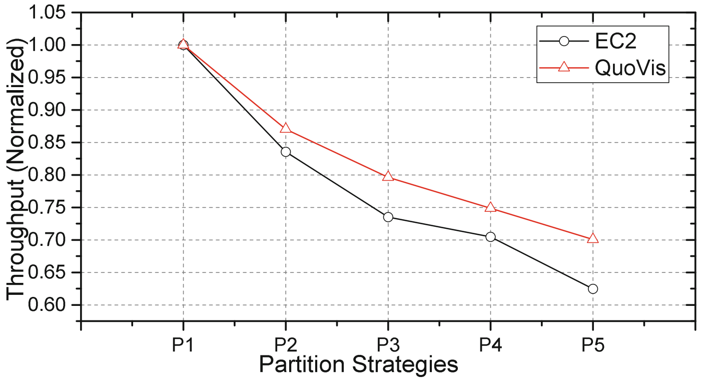


Fig. 11. data partitioning strategies - a real cluster vs QuoVis with 2 replicas. The results are normalized. The data partitions of servers s_1:s_2:s_3:s_4:s_5 are: uniform (P1), 15:15:15:15:40 (P2), 12:12:13:13:50 (P3), 10:10:10:10:60 (P4) and 7:7:8:8:70 (P5).

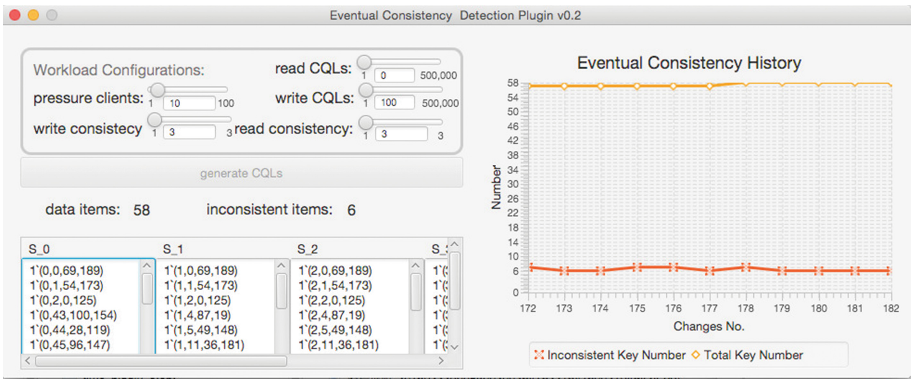
data sets of s_1 and s_5 are the largest. Therefore, we conjecture that s_1 and s_5 are the performance bottlenecks. This comparison has two implications: a uniform data partitioning is better in these settings; or node s_1 and s_5 should be upgraded; or the *concurrent write number* parameters on s_1 and s_5 should be optimized to achieve better performance if we must use this data partitioning strategy.

5.6 Consistency Detection

Eventual consistency history plugin supports detecting the replica consistency. Figure 12 shows the screenshot of the plugin. For each data item, if any replica contains a different value, we label the data item as inconsistent. When the *persistent data set* of a node changes, the plugin records the number of inconsistent data items and displays the records in a line chart (i.e., the red line with 'x' symbol in the chart).

Figure 12 shows an example that the system is not yet consistent after all writing requests are finished. In Fig. 12, sometimes the up and down of the red line means that some data inconsistencies are temporary. If the number of inconsistencies is greater than 0 at the end, it means that some data items are inconsistent permanently. For example, for a writing request, the message will queue up in a propagation queue, a write queue, the propagation queue again and then a response queue in turn. Since each node has its own progress in digesting its messages (e.g., some may have more messages), so temporary inconsistencies will occur.

However, the timeout parameter limits the waiting time for a writing operation, the message has to depart from the queue if the time is up. If a message is timeout, the replica will be inconsistent permanently. Therefore, to keep replicas consistent eventually, we need to set the timeout value big enough or run on high performance hardware and network.



Eventual Consistency Detection Plugin v0.2

Workload Configurations:

pressure clients: 1 10 100

write consistecy 1 3 3

read CQLs: 1 0 500,000

write CQLs: 1 100 500,000

read consistency: 1 3 3

generate CQLs

data items: 58

inconsistent items: 6

S_0	S_1	S_2	S_
1\`(0,0,69,189)	1\`(1,0,69,189)	1\`(2,0,69,189)	1\`(
1\`(0,1,54,173)	1\`(1,1,54,173)	1\`(2,1,54,173)	1\`(
1\`(0,2,0,125)	1\`(1,2,0,125)	1\`(2,2,0,125)	1\`(
1\`(0,43,100,154)	1\`(1,4,87,19)	1\`(2,4,87,19)	1\`(
1\`(0,44,28,119)	1\`(1,5,49,148)	1\`(2,5,49,148)	1\`(
1\`(0,45,96,147)	1\`(1,11,36,181)	1\`(2,11,36,181)	1\`(

Fig. 12. A case of that not all the replicas eventually reach to consistent status (Color figure online)

The timeout strategy indicates that eventual consistency is not satisfied with only writing operations. Most systems address this issue by triggering a repair when they receive read requests. For example, Cassandra uses *read repair* mechanism to repair these inconsistent data items automatically. However, this repair always incurs performance penalty. We can use QuoVis to tune the timeout parameters to avoid this data consistency issue.

By using Eq. 3, we can calculate the inconsistency time window. In this simulation, each data item has 3 replicas. The write consistency level is 2. There are 20000 different data items. When one node finishes writing a replica of a data item, we calculate the inconsistency time window by Eq. 3. After all the data items are written, we draw the inconsistency time window distribution for visualization.

Figure 13 shows the result. The black line is the distribution of the *itw*s between two participators. The red line is the distribution of the *itw*s between

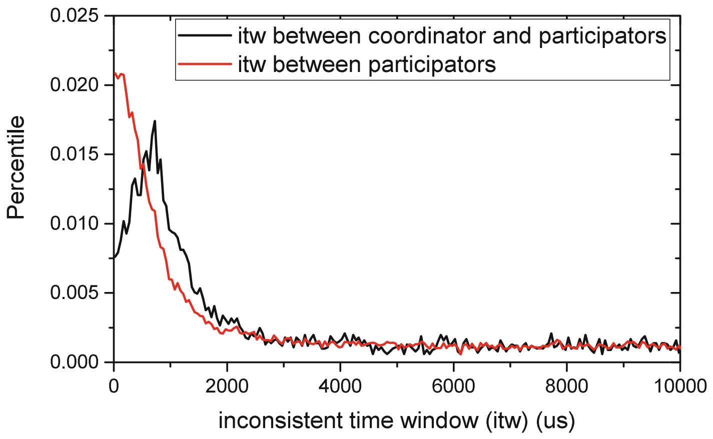


Fig. 13. The distribution of the inconsistent time window (Color figure online)

coordinators and its participators. Both the *itw* distribution have long-tails. Therefore, we omit the *itw* whose value is greater than 10000. The black line has higher probability to reach a small *itw* than the red line, so that we say the *itw* between participators is better than the *itw* between coordinators and its participators. It is because that writing on a coordinator node saves a time cost of "transmit" transition.

6 Conclusion

Distributed NoSQL systems are popular, but tuning and analyzing these systems can be complicated and tedious. We propose using CPN to model NoSQL systems and for performance and consistency analysis. We have presented our recently built simulator called QuoVis for analyzing and tuning a quorum-based NoSQL system, Cassandra. The simulator is supported by the engine of CPN Tools and Access/CPN. Several usage scenarios have been presented to show the effectiveness of our simulator for deploying, maintaining (including visualizing and monitoring) a distributed NoSQL system. Since our simulator is built based on an extensible (plugin-based), modular architecture, its functionalities can be easily extended. Finally, although the underlying model of our simulator is based on Cassandra, we believe that it can be easily customized for other NoSQL systems.

Acknowledgement. The paper is supported by The National Key R&D Program of China (2016YFB1000701) and NSFC (61325008).

References

1. Aguilera-Mendoza, L., Llorente-Quesada, M.T.: Modeling and simulation of hadoop distributed file system in a cluster of workstations. In: Cuzzocrea, A., Maabout, S. (eds.) MEDI 2013. LNCS, vol. 8216, pp. 1–12. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41366-7_1
2. Bailis, P., Ghodsi, A.: Eventual consistency today: limitations, extensions, and beyond. Commun. ACM **56**(5), 55–63 (2013)
3. Bailis, P., Venkataraman, S., Franklin, M.J., Hellerstein, J.M., Stoica, I.: Probabilistically bounded staleness for practical partial quorums. VLDB **5**(8), 776–787 (2012)
4. Bao, X., Liu, L., Xiao, N., Zhou, Y., Zhang, Q.: Policy-driven configuration management for NoSQL. In: 2015 IEEE 8th International Conference on Cloud Computing (CLOUD), pp. 245–252, June 2015
5. Barbot, B., Kwiatkowska, M.: On quantitative modelling and verification of DNA walker circuits using stochastic Petri Nets. In: Devillers, R., Valmari, A. (eds.) PETRI NETS 2015. LNCS, vol. 9115, pp. 1–32. Springer, Cham (2015). doi:10.1007/978-3-319-19488-2_1
6. Bermbach, D.: Benchmarking Eventually Consistent Distributed Storage Systems. KIT Scientific Publishing, Karlsruhe (2014)

7. Bermbach, D., Tai, S.: Eventual consistency: how soon is eventual? An evaluation of amazon s3's consistency behavior. In: Proceedings of the 6th Workshop on Middleware for Service Oriented Computing, p. 1. ACM (2011)
8. Bushik, S.: A Vendor-Independent Comparison of NoSQL Databases: Cassandra, HBase, MongoDB, Riak. Network World, 22 October 2012
9. van Der Aalst, W.M., Ter Hofstede, A.H., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14**(1), 5–51 (2003)
10. Di, S., Cappello, F.: GloudSim: Google trace based cloud simulator with virtual machines. Softw. Pract. Experience **45**(11), 1571–1590 (2015)
11. Feinberg, A.: Project voldemort: reliable distributed storage. In: Proceedings of the 10th IEEE International Conference on Data Engineering (2011)
12. Gaudel, Q., Ribot, P., Chanthery, E., Daigle, M.J.: Health monitoring of a planetary rover using hybrid particle Petri Nets. In: Kordon, F., Moldt, D. (eds.) PETRI NETS 2016. LNCS, vol. 9698, pp. 196–215. Springer, Cham (2016). doi:10.1007/978-3-319-39086-4_13
13. Gifford, D.K.: Weighted voting for replicated data. In: Proceedings of the Seventh ACM Symposium on Operating Systems Principles, pp. 150–162. ACM (1979)
14. Golab, W., Li, X., Shah, M.A.: Analyzing consistency properties for fun and profit. In: Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp. 197–206. ACM (2011)
15. Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F.B., Babu, S.: Starfish: a self-tuning system for big data analytics. CIDR **11**, 261–272 (2011)
16. Huang, X., Wang, J., Bai, J., Ding, G., Long, M.: Inherent replica inconsistency in Cassandra. In: 2014 IEEE International Congress on Big Data, pp. 740–747. IEEE (2014)
17. Huang, X., Wang, J., Zhong, Y., Song, S., Yu, P.S.: Optimizing data partition for scaling out NoSQL cluster. Concurrency Comput. Pract. Experience **17**, 5793–5809 (2015)
18. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the 29th ACM Symposium on Theory of Computing, pp. 654–663. ACM (1997)
19. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Syst. Rev. **44**(2), 35–40 (2010)
20. Liao, W., Hou, K., Zheng, Y., He, X.: Modeling and simulation of troubleshooting process for automobile based on Petri Net and flexsim. In: Qi, E., Shen, J., Dou, R. (eds.) The 19th International Conference on Industrial Engineering and Engineering Management, pp. 1141–1153. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38391-5_121
21. Mace, J., Roelke, R., Fonseca, R.: Pivot tracing: dynamic causal monitoring for distributed systems. In: Proceedings of the 25th SOSP, pp. 378–393. ACM (2015)
22. Majidi, F., Harounabadi, A.: Presentation of an executable model for evaluation of software architecture using blackboard technique and formal models. JACST **4**(1), 23–31 (2015)
23. Montresor, A., Jelasity, M.: Peersim: A scalable P2P simulator. In: 2009 IEEE Ninth International Conference on Peer-to-Peer Computing, pp. 99–100. IEEE (2009)
24. Osman, R., Piazzolla, P.: Modelling replication in NoSQL datastores. In: Norman, G., Sanders, W. (eds.) QEST 2014. LNCS, vol. 8657, pp. 194–209. Springer, Cham (2014). doi:10.1007/978-3-319-10696-0_16

25. Shi, J., Qiu, Y., Minhas, U.F., Jiao, L., Wang, C., Reinwald, B., Özcan, F.: Clash of the titans: MapReduce vs. Spark for large scale data analytics. VLDB **8**(13), 2110–2121 (2015)
26. Shi, J., Zou, J., Lu, J., Cao, Z., Li, S., Wang, C.: MRTuner: a toolkit to enable holistic optimization for MapReduce jobs. VLDB **7**(13), 1319–1330 (2014)
27. Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M.J., Theimer, M.M., Welch, B.B.: Session guarantees for weakly consistent replicated data. In: 1994 Proceedings of the Third International Conference on Parallel and Distributed Information Systems, pp. 140–149. IEEE (1994)
28. Thomas, R.H.: A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. Database Syst. (TODS) **4**(2), 180–209 (1979)
29. Wagenhals, L.W., Liles, S.W., Levis, A.H.: Toward executable architectures to support evaluation. In: 2009 International Symposium on Collaborative Technologies and Systems, pp. 502–511, May 2009
30. Wang, K., Kulkarni, A., Lang, M., Arnold, D., Raicu, I.: Using simulation to explore distributed key-value stores for extreme-scale system services. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 9. ACM (2013)
31. Westergaard, M.: Access/CPN 2.0: a high-level interface to coloured Petri Net models. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 328–337. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21834-7_19