Quantitative Analysis of Consistency in NoSQL Key-value Stores ⋆

Si Liu, Son Nguyen, Jatin Ganhotra, Muntasir Raihan Rahman, Indranil Gupta, and José Meseguer

Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract. The promise of high scalability and availability has prompted many companies to replace traditional relational database management systems (RDBMS) with NoSQL key-value stores. This comes at the cost of relaxed consistency guarantees: key-value stores only guarantee eventual consistency in principle. In practice, however, many key-value stores seem to offer stronger consistency. Quantifying how well consistency properties are met is a non-trivial problem. We address this problem by formally modeling key-value stores as probabilistic systems and quantitatively analyzing their consistency properties by statistical model checking. We present for the first time a formal probabilistic model of Apache Cassandra, a popular NoSQL key-value store, and quantify how much Cassandra achieves various consistency guarantees under various conditions. To validate our model, we evaluate multiple consistency properties using two methods and compare them against each other. The two methods are: (1) an implementation-based evaluation of the source code; and (2) a statistical model checking analysis of our probabilistic model.

1 Introduction

The promise of high scalability and availability has prompted many companies and organizations to replace traditional relational database management systems (RDBMS) with NoSQL key-value stores in order to store large data sets and tackle an increasing number of users. According to DB-Engines Ranking [2] by June 2015, three NoSQL datastores, namely MongoDB [3], Cassandra [1] and Redis [4], have advanced into the top 10 most popular database engines among 277 systems, highlighting the increasing popularity of NoSQL key-value stores. For example, Cassandra is currently being used at Facebook, Netflix, eBay, GitHub, Instagram, Comcast, and over 1500 more companies.

NoSQL key-value stores invariably replicate application data on multiple servers for greater availability in the presence of failures. Brewer's CAP theorem [12] implies that, under network partitions, a key-value store must choose between consistency (keeping all replicas in sync) and availability (latency). Many

⋆ Partially supported by NSF CNS 1319527, NSF 1409416, NSF CCF 0964471, and AFOSR/AFRL FA8750-11-2-0084.

key-value stores prefer availability, and thus they provide a relaxed form of consistency guarantees (e.g., eventual consistency [27]). This means key-value store applications can be exposed to stale values. This can negatively impact key-value store user experience. Not surprisingly, in practice, many key-value stores seem to offer stronger consistency than they promise. Therefore there is considerable interest in accurately predicting and quantifying what consistency properties a key-value store actually delivers, and in comparing in an objective, and quantifiable way how well properties of interest are met by different designs.

However, the task of accurately predicting such consistency properties is non-trivial. To begin with, building a large scale distributed key-value store is a challenging task. A key-value store usually embodies a large number of components (e.g., membership management, consistent hashing, and so on), and each component can be thought of as source code which embodies many complex design decisions. Today, if a developer wishes to improve the performance of a system (e.g., to improve consistency guarantees, or reduce operation latency) by implementing an alternative design choice for a component, then the only option currently available is to make changes to huge source code bases (e.g., Apache Cassandra [1] has about 345,000 lines of code). Not only does this require many man months, it also comes with a high risk of introducing new bugs, needs understanding in a huge code base before making changes, and is unfortunately not repeatable. Developers can only afford to explore very few design alternatives, which may in the end fail to lead to a better design.

In this paper we address these challenges by proposing a formally model-based methodology for designing and quantitatively analyzing key-value stores. We formally model key-value stores as probabilistic systems specified by *probabilistic rewrite rules* [5], and quantitatively analyze their properties by *statistical model checking* [24,30]. We demonstrate the practical usefulness of our methodology by developing, to the best of our knowledge for the first time, a formal probabilistic model of Cassandra, as well as of an alternative Cassandra-like design, in Maude [13]. Our formal probabilistic model extends and improves a nondeterministic one we used in [19] to answer *qualitative* yes/no consistency questions about Cassandra. It models the main components of Cassandra and its environment such as strategies for ordering multiple versions of data and message delay. We have also specified two consistency guarantees that are largely used in industry, *strong consistency* (the strongest consistency guarantee) and *read your writes* (a popular intermediate consistency guarantee) [26], in the QuaTEx probabilistic temporal logic [5]. Using the PVeStA [6] statistical model checking tool we have then quantified the satisfaction of such consistency properties in Cassandra under various conditions such as consistency level combination and operation issue time latency. To illustrate the versatility and ease with which different design alternatives can be modeled and analyzed in our methodology, we have also modeled and analyzed the same properties for an alternative Cassandra-like design.

An important question is how much trust can be placed on such models and analysis. That is, how reliable is the predictive power of our proposed methodol-

ogy? We have been able to answer this question for our case study as follows: (i) we have experimentally evaluated the same consistency properties for both Cassandra and the alternative Cassandra-like design[1]; and (ii) we have compared the results obtained from the formal probabilistic models and the statistical model checking with the experimentally-obtained results. Our analysis indicates that the model-based consistency predictions conform well to consistency evaluations derived experimentally from the real Cassandra deployment, with both showing that Cassandra in fact achieves much higher consistency (sometimes up to strong consistency) than the promised eventual consistency. They also show that the alternative design is not competitive in terms of the consistency guarantees considered. Our entire Maude specification, including the alternative design, has less than 1000 lines of code, which further underlines the versatility and ease of use of our methodology at the software engineering level.

Our main contributions include:

- We present a formal methodology for the quantitative analysis of key-value store designs and develop, to the best of our knowledge for the first time, a *formal executable probabilistic model* for the Cassandra key-value store and for an alternative Cassandra-like design.
- We present, to the best of our knowledge for the first time, a statistical model checking analysis for quantifying consistency guarantees, namely, *strong consistency* and *read your writes*, in Cassandra and the alternative design.
- We demonstrate the good predictive power of our methodology by comparing the model-based consistency predictions with experimental evaluations from a real Cassandra deployment on a real cluster. Our results indicate similar consistency trends for the model and deployment.

2 Preliminaries

2.1 Cassandra Overview

Cassandra [1] is a distributed, scalable, and highly available NoSQL database design. It is distributed over collaborative servers that appear as a single instance to the end client. Data are dynamically assigned to several servers in the cluster (called the *ring*), and each server (called a *replica*) is responsible for different ranges of the data stored as key-value pairs. Each key-value pair is stored at multiple replicas for fault-tolerance.

In Cassandra a client can perform *read* or *write* operations to query or update data. When a client requests a read/write operation to a cluster, the server it is connected to will act as a *coordinator* to forward the request to all replicas that hold copies of the requested key-value pair. According to the specified *consistency level* in the operation, the coordinator will reply to the client with a value/ack after collecting *sufficient* responses from replicas. Cassandra supports tunable

[1] We implemented the alternative Cassandra-like design by modifying the source code of Apache Cassandra version 1.2.0.

consistency levels with `ONE`, `QUORUM` and `ALL` the three major ones, e.g., a `QUORUM` read means that, when a *majority* of replicas respond, the coordinator returns to the client the most recent value. This strategy is thus called *Timestamp-based Strategy* (TB) in the case of processing reads. Note that replicas may return different timestamped values to the coordinator. To ensure that all replicas agree on the most current value, Cassandra uses in the background the *read repair* mechanism to update those replicas holding outdated values.

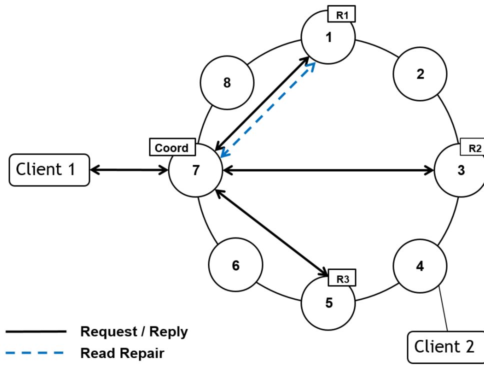

Fig. 1. Cassandra deployed in a single cluster of 8 servers with replication factor 3

Fig. 1 shows an example Cassandra deployed in a single data center cluster of eight nodes with three replicas and consistency level `QUORUM`. The read/write from client `1` is forwarded to all three replicas `1`, `3` and `5`. The coordinator `7` then replies to client `1` after receiving the first two responses, e.g., from `1` and `3`, to fulfill the request without waiting for the reply from `5`. For a read, upon retrieving all three possibly different versions of values, the coordinator `7` then issues a read repair write with the highest timestamped value to the outdated replica, `1`, in this example. Note that various clients may connect to various coordinators in the cluster, but requests from any client on the same key will be forwarded to the same replicas by those coordinators.

2.2 Rewriting Logic and Maude

Rewriting logic [21] is a semantic framework to specify concurrent, object-oriented systems as *rewrite theories* $(\Sigma, E \cup A, R)$, where $(\Sigma, E \cup A)$ is a *membership equational logic theory* [13], with Σ an algebraic signature declaring sorts, subsorts, and function symbols, E a set of conditional equations, and A a set of equational axioms. It specifies the system's state space as an algebraic data type; R is a set of *labeled conditional rewrite rules* specifying the system's *local transitions*, each of which has the form $[l] : t \longrightarrow t'$ **if** *cond*, where *cond* is a condition or guard and l is a label. Such a rule specifies a transition from an instance of t to the corresponding instance of t', provided the condition holds.

The Maude system [13] executes rewrite theories specified as Maude modules, and provides a rich collection of analysis tools. In this paper we consider distributed systems such as Cassandra made up of objects communicating with each other via asynchronous message passing. The distributed state of such a system is formalized as a *multiset* of objects and messages, and a state transition is *multiset rewriting*. In an object-oriented module, an object of the form `< id : class | a1 : v1, a2 : v2, ..., an : vn >` is an instance (with a unique

name `id`) of the `class` that encapsulates the attributes `a1` to `an` with the current values `v1` to `vn`. Upon receiving a message, an object can change its state and send messages to other objects. For example, the rewrite rule (with label `l`)

```
rl [l] :  m(O,z)  < O : C | a1 : x, a2 : O' >
     =>   < O : C | a1 : x + z, a2 : O' >  m'(O',x + z) .
```

defines a transition where an incoming message `m`, with parameters `O` and `z`, is consumed by the target object `O` of class `C`, the attribute `a1` is updated to `x + z`, and an outgoing message `m'(O',x + z)` is generated.

2.3 Statistical Model Checking and PVeStA

Distributed systems are probabilistic in nature, e.g., network latency such as message delay may follow a certain probability distribution, plus some algorithms may be probabilistic. Systems of this kind can be modeled by *probabilistic rewrite theories* [5] with rules of the form:

$$[l] : t(\overrightarrow{x}) \to t'(\overrightarrow{x}, \overrightarrow{y}) \;\; if \;\; cond(\overrightarrow{x}) \;\; with \;\; probability \;\; \overrightarrow{y} := \pi(\overrightarrow{x})$$

where the term t' has additional new variables $\overrightarrow{y}$ disjoint from the variables $\overrightarrow{x}$ in the term t. Since for a given matching instance of the variables $\overrightarrow{x}$ there can be many (often infinite) ways to instantiate the extra variables $\overrightarrow{y}$, such a rule is *non-deterministic*. The probabilistic nature of the rule stems from the probability distribution $\pi(\overrightarrow{x})$, which depends on the matching instance of $\overrightarrow{x}$, and governs the probabilistic choice of the instance of $\overrightarrow{y}$ in the result $t'(\overrightarrow{x}, \overrightarrow{y})$ according to $\pi(\overrightarrow{x})$. In this paper we use the above PMaude [5] notation for probabilistic rewrite rules.

Statistical model checking [24,30] is an attractive formal approach to analyzing probabilistic systems against temporal logic properties. Instead of offering a yes/no answer, it can verify a property up to a user-specified level of confidence by running Monte-Carlo simulations of the system model. For example, if we consider strong consistency in Cassandra, a statistical model-checking result may be "The Cassandra model satisfies strong consistency 86.87% of the times with 99% confidence". Existing statistical verification techniques assume that the system model is purely probabilistic. Using the methodology in [5,15] we can eliminate non-determinism in the choice of firing rules. We then use PVeStA [6], an extension and parallelization of the tool VeStA [25], to statistically model check purely probabilistic systems against properties expressed by QuaTEx probabilistic temporal logic [5]. The expected value of a QuaTEx expression is iteratively evaluated w.r.t. two parameters α and δ provided as input by sampling until the size of (1-α)100% confidence interval is bounded by δ. In this paper we will compute the expected probability of satisfying a property based on definitions of the form `p() = BExp ; eval E[# p()] ;`, where `#` is the next operator, `BExp` is a consistency-specific predicate, and `p()` is a state predicate returning the value either `1.0` or `0.0` after checking whether the current state satisfies `BExp` or not.

3 Replicated Data Consistency

Distributed key-value stores usually sacrifice consistency for availability (Brewer's CAP theorem [12]), advocating the notion of weak consistency (e.g., Cassandra promises eventual consistency [1]). However, studies on benchmarking eventually consistent systems have shown that those platforms seem in practice to offer more consistency than they promise [28,11]. Thus a natural question derived from those observations is "what consistency does your key-value store actually provide?" We summarize below the prevailing consistency guarantees advocated by Terry [26]. We will focus on two of them (*strong consistency* and *read your writes*) in the rest of this paper.

- Strong Consistency (SC) ensures that each read returns the value of the last write that occurred before that read.
- Read Your Writes (RYW) guarantees that the effects of all writes performed by a client are visible to her subsequent reads.
- Monotonic Reads (MR) ensures a client to observe a key-value store increasingly up to date over time.
- Consistent Prefix (CP) guarantees a client to observe an ordered sequence of writes starting with the first write to the system.
- (Time-) Bounded Staleness (BS) restricts the staleness of values returned by reads within a time period.
- Eventual Consistency (EC) claims that if no new updates are made, eventually all reads will return the last updated value.

Note that SC and EC lie at the two ends of the consistency spectrum, while the other intermediate guarantees are not comparable in general [26].

In [19,20] we investigated SC, RYW and EC from a qualitative perspective using standard model checking, where they were specified using *linear temporal logic* (LTL). The questions we answered there are simply yes/no questions such as "Does Cassandra satisfy strong consistency?" and "In what scenarios does Cassandra violate read your writes?". We indeed showed by counterexamples that Cassandra violates SC and RYW under certain circumstances, e.g., successive write and read with the combinations of lower consistency levels. Regarding EC, the model checking results of our experiments with bounded number of clients, servers and messages conforms to the promise. We refer the reader to [19,20] for details.

In this paper we look into the consistency issue for Cassandra in terms of SC and RYW from a quantitative, statistical model checking perspective. To aid the specification of the two properties (Section 5.1), we now restate them more formally. As all operations from different clients can be totally ordered by their issuing times, we can first view, from a client's perspective, a key-value store S as a history $H = o_1, o_2, ..., o_n$ of n read/write operations, where any operation o_i can be expressed as $o_i = (k, v, c, t)$, where t denotes the *global* time when o_i was issued by client c, and v is the value read from or written to on key k. We can then define both consistency properties based on H:

– We say S satisfies SC if for any read $o_i = (k, v_i, c_i, t_i)$, provided there exists a write $o_j = (k, v_j, c_j, t_j)$ with $t_j < t_i$, and without any other write $o_h = (k, v_h, c_h, t_h)$ such that $t_j < t_h < t_i$, we have $v_i = v_j$. Note that c_h, c_i and c_j are not necessarily different;
– We say S satisfies RYW if either (1) S satisfies SC, or (2) for any read $o_i = (k, v_i, c_i, t_i)$, provided there exists a write $o_j = (k, v_j, c_j, t_j)$ with $c_i = c_j$ and $t_j < t_i$, and with any other write $o_h = (k, v_h, c_h, t_h)$ such that $c_i \neq c_h$ and $t_j < t_h < t_i$, we have $v_i = v_j$.

4 Probabilistic Modeling of Cassandra Designs

This section describes a formal probabilistic model of Cassandra including the underlying communication model (Section 4.1) as well as an alternative Cassandra-like design (Section 4.2). The entire executable Maude specifications are available at `https://sites.google.com/site/siliunobi/qest15-cassandra`.

4.1 Formalizing Probabilistic Communication in Cassandra

In [19] we built a formal executable model of Cassandra summarized in Section 2.1. Specifically, we modeled the ring structure, clients and servers, messages, and Cassandra's dynamics. Moreover, we also introduced a *scheduler* object to schedule messages by maintaining a global clock `GlobalTime`[2] and a queue of inactive/scheduled messages `MsgQueue`. By activating those messages, it provides a deterministic total ordering of messages and allows synchronization of all clients and servers, aiding formal analysis of consistency properties (Section 5.1).

Fig. 2. Visualization of rewrite rules for forwarding requests from a coordinator to the replicas

To illustrate the underlying communication model, Fig. 2 visualizes a segment of the system transitions showing how messages flow between a coordinator and the replicas through the scheduler in terms of rewrite rules. The delayed messages (of the form `[...]`) `[D1, repl1 <- Msg1]` and `[D2, repl2 <- Msg2]`,

[2] Though in reality synchronization can never be exactly reached due to clock skew [17], cloud system providers use NTP or even expensive GPS devices to keep all clocks synchronized (e.g., Google Spanner). Thus our abstraction of a global clock is reasonable.

targeting replicas `repl1` and `repl2`, are produced by the coordinator at global time `T` with the respective message delays `D1` and `D2`. The scheduler then enqueues both messages for scheduling. As the global time advances, messages eventually become active (of the form {...}), and are appropriately delivered to the replicas. For example, the scheduler first dequeues `Msg1` and then `Msg2` at global time `T + D1` and `T + D2` respectively, assuming `D1 < D2`. Note that messages can be consumed by the targets only when they are active.

As mentioned in Section 2.3, we need to eliminate nondeterminism in our previous Cassandra model prior to statistical model checking. This can be done by transforming nondeterministic rewrite rules to purely probabilistic ones. Below we show an example transformation, where both rules illustrate how the coordinator reacts upon receiving a read reply `ReadReplySS` from a replica, with `KV` the returned key-value pair of the form `(key,value,timestamp)`, `ID` and `A` the read and client's identifiers, and `CL` the read's consistency level, respectively. The coordinator `S` adds `KV` to its local buffer, and returns to `A` the highest timestamped value determined by `tb` via the message `ReadReplyCS`, provided it has collected the consistency-level number of responses determined by `cl?`.

In the nondeterministic version `[...-nondet]`, the outgoing message is equipped with a delay `D` nondeterministically selected from the delay set `delays`. We keep the set unchanged so that standard model checking will explore all possible choices of delays each time the rule is fired. For example, if `delays: (2.0,4.0)`, two read replies will be generated nondeterministically with the delays `2.0` and `4.0` time units respectively, each of which will lead to an execution path during the state space exploration.

```
crl [on-rec-rrep-coord-nondet] :
    {T, S <- ReadReplySS(ID,KV,CL,A)}
    < S : Server | buffer: BF, delays: (D,DS), AS >
 => < S : Server | buffer: BF', delays: (D,DS), AS >
    (if cl?(CL,BF') then
     [D, A <- ReadReplyCS(ID,tb(BF'))]
     else none fi)
 if BF' := add(ID,KV,BF) .
```

```
crl [on-rec-rrep-coord-prob] :
    {T, S <- ReadReplySS(ID,KV,CL,A)}
    < S : Server | buffer: BF, AS >
 => < S : Server | buffer: BF', AS >
    (if cl?(CL,BF') then
     [D, A <- ReadReplyCS(ID,tb(BF'))]
     else none fi)
 if BF' := add(ID,KV,BF)
    with probability D := distr(...) .
```

We transform the above rule to the probabilistic version `[...-prob]`, where the delay `D` is distributed according to the parameterized probability distribution function `distr(...)`. Once the rule fires, only one read reply will be generated with a probabilistic real-valued message delay.

Likewise all nondeterministic rules in our previous model can be transformed to purely probabilistic rewrite rules. Furthermore, as explained in [5,15], the use of continuous time and the actor-like nature of the specification ensure that *only one probabilistic rule is enabled at each time instant*, thus eliminating any remaining nondeterminism from the firing of rules.

4.2 Alternative Strategy Design

Two major advantages of our model-based approach are: (1) the ease of designing new strategies in an early design stage, and (2) the ability to predict their effects

before implementation. Here we illustrates the first part by presenting as an alternative design the *Timestamp-agnostic Strategy* (TA). The key idea is that, instead of using timestamps to decide which value will be returned to the client as TB does (Section 2.1), TA uses the values themselves to decide which replica has the latest value. For example, if the replication factor is 3, then for a `QUORUM` read, the coordinator checks whether the values returned by the first two replicas are identical: if they are, the coordinator returns that value; otherwise it waits for the third replica to return a value. If the third value matches one of the first two values, the coordinator returns the third value. So for a `QUORUM` read TA guarantees that the coordinator will reply with the value that has been stored at a majority of replicas. For an `ALL` read, the coordinator compares all three values; if they are all the same, it returns that value. Notice that TA and TB agree on processing `ONE` reads.

To formalize TA (or other alternative strategies) we only need to specify the corresponding functions of the returned values from the replicas buffered at the coordinator, as we defined `tb` for TB, without redefining the underlying model. We omit the specification (available online) for simplicity. Note that our component-based model also makes it possible to dynamically choose the optimal strategy in favor of consistency guarantees. More precisely, once we have a pool of strategies and their respective strengths in consistency guarantees (which can be measured by statistical model checking), the coordinator can invoke the corresponding strategy-specific function based on the client's preference. For example, given strategies S1/S2 offering consistency properties C1/C2, if a client issues two consecutive reads with desired consistency C1, C2, respectively, the coordinator will generate, e.g., the C1-consistent value for the preceding read, by calling the strategy function for S1.

5 Quantitative Analysis of Consistency in Cassandra

How well do our Cassandra model and its TA alternative design satisfy SC and RYW? Does TA provide more consistency than TB based on our model? Are those results consistent with reality? We propose to investigate these questions by statistical model checking (Section 5.1) and by implementation-based evaluation (Section 5.2) of both consistency properties in terms of two strategies.

5.1 Statistical Model Checking Analysis

Scenarios. We define the following setting for our experimental scenarios of statistical model checking:

- We consider a single cluster of 4 servers, and the replication factor of 3.
- All replicas are initialized with default key-value pairs.
- Each read/write can have consistency level `ONE`, `QUORUM` or `ALL`, and all operations concern the same key.
- We consider the lognormal distribution for message delay with the mean μ `= 0.0` and standard deviation σ `= 1.0` [10].

– All consistency probabilities are computed with a 99% confidence level of size at most 0.01 (Section 2.3).

Fig. 3 shows the two scenarios, with each parallel line denoting one session of one client. Regarding SC, we consider a scenario of three consecutive operations, W1, W2 and R3, issued by three different clients, respectively, where `L1` and `L2` are the issuing latencies between them. We choose to experiment with consistency level `ONE` for both W1 and W2 to evaluate different consistency levels for R3. Thus we name each subscenario (TB/TA-O/Q/A) depending on the target strategy and R3's consistency level, e.g., (TB-Q) refers to the case checking SC for TB with R3 of `QUORUM`.

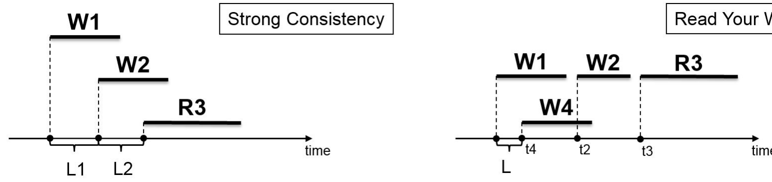

Fig. 3. Experimental Scenarios of Statistical Model Checking of SC and RYW

Regarding RYW, we consider a scenario with four operations, where W1, W2 and R3 are issued by one client and *strictly ordered* (a subsequent operation will be blocked until the preceding one on the same key finishes) while W4 is from the other client[3]. The issuing latency `L` is tunable, which can vary the issuing time of W4. Thus we can derive the corresponding cases in RYW's definition (Section 3), and specify and analyze the property accordingly. We choose to experiment with consistency level `ONE` for both W1 and W4 to evaluate different combinations of consistency levels for W2 and R3. The only possible cases violating RYW are, if we forget W4 for the moment, (R3,W2) = (O,O)/(O,Q)/(Q,O) due to the fact that a read is guaranteed to see its preceding write from the same client, if `R + W > RF` with `R` and `W` the respective consistency levels and `RF` the replication factor. Thus we name each subscenario (TB/TA-OO/OQ/QO/...) depending on the target strategy and the combination of consistency levels. For simplicity, we let W2 and R3 happen immediately upon their preceding operations finish.

Formalizing Consistency Properties. Based on the consistency definitions (Section 3) and the above scenario, SC is satisfied if R3 reads the value of W2. Thus we define a parameterized predicate `sc?(A,A',O,O',C)` that holds if we can match the value returned by the subsequent read `O` (R3 in this case) from client `A` with that in the preceding write `O'` (W2) from client `A'`. Note that the attribute `store` records the associated information of each operation issued by the client: operation `O` was issued at global time `T` on key `K` with returned/written value `V` for a read/write.

[3] Section 3 describes two disjoint cases for RYW, which we mimic with tuneable `L`: if `t4 < t2`, only W2 is RYW-consistent; otherwise both W2 and W4 are RYW-consistent.

```
op sc? : Address Address Nat Nat Config -> Bool .
eq sc?(A,A',O,O',< A  : Client | store : ((O, K,V,T), ...), ... >
                 < A' : Client | store : ((O',K,V,T'), ...), ... > REST) = true .
```

Likewise we define for RYW a parameterized predicate `ryw?(A,A',O1,O2,O3,C)` that holds if we can match the value returned by the subsequent read `O2` (R3 in this case) with that in the preceding write `O1` by itself (W2 in this case), or in a more recent write `O3` (W4 in this case if issued after W2) determined by `T3 >= T1`).

```
 op ryw? : Address Address Nat Nat Nat Config -> Bool .
 eq ryw?(A,A',O1,O2,O3,< A  : Client | store : ((O1,K,V,T1), (O2,K,V, T2), ...),
         ... > REST) = true .
ceq ryw?(A,A',O1,O2,O3,< A  : Client | store : ((O1,K,V,T1), (O2,K,V',T2), ...), ... >
         < A' : Client | store : ((O3,K,V',T3), ...), ... > REST) = true if T3 >= T1 .
```

Analysis Results for SC. Fig. 4 shows the resulting probability of satisfying SC, where the probability (of R3 reading W2) is plotted against the issuing latency (`L2`) between them. Regarding TB, from the results (and intuitively), given the same issuing latency, increasing the consistency level provides higher consistency; given the same consistency level, higher issuing latency results in higher consistency (as the replicas converge, a sufficiently later read (R3) will return the consistent value up to 100%). Surprisingly, `QUORUM` and `ALL` reads start to achieve SC within a very short latency around `0.5` and `1.5` time units respectively (with `5` time units for even `ONE` reads).

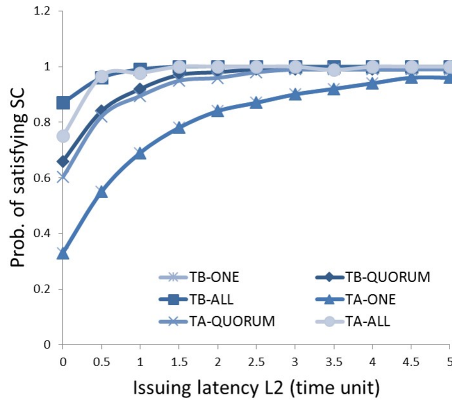

Fig. 4. Probability of Satisfying SC by Statistical Model Checking

On the other hand, all observations for TB apply to TA in general. In fact, for `QUORUM` and `ALL` reads, the two strategies perform almost the same, except that: (1) for `ALL` reads, TB provides noticeably more consistency than TA within an extremely short latency of `0.5` time units; and (2) for `QUORUM` reads, TB offers slightly more consistency than TA within `2.5` time units.

Based on the results it seems fair to say that both TB and TA provide high SC, especially with `QUORUM` and `ALL` reads. The consistency difference between the two strategies results from the overlap of R3 and W2. More precisely, since the subsequent read has higher chance to read multiple versions of the key-value pair with lower issuing latency, TA, only relying on the version itself, will return the matched value that is probably stale.

Analysis Results for RYW. Fig. 5-(a) shows the resulting probability of satisfying RYW, where the probability (of R3 reading W2 or a more recent value) is plotted against the issuing latency (`L`) between W1 and W4. From the results it is straightforward to see that scenarios (TB-OA/QQ/AO/AA) guarantee RYW due to the fact "`R3 + W2 > RF`". Since we have already seen that the Cassandra model satisfied SC quite well, it is also reasonable that all combinations of consistency levels provide high RYW consistency, even with the lowest combination `(O,O)` that already achieves a probability around 90%. Surprisingly, it appears that a `QUORUM` read offers RYW consistency nearly 100%, even after a preceding write with the low consistency level down to `ONE` (scenario (TB-OQ)). Another observation is that, in spite of the concurrent write from the other client, the probability of satisfying RYW stays fairly stable.

Fig. 5-(b) shows the comparison of TA and TB regarding RYW, where for simplicity we only list three combinations of consistency levels from R3's perspective with W2's consistency level fixed to `ONE` (in fact, with W2's consistency level increases, the corresponding scenarios will provide even higher consistency). In general, all observations for TB apply to TA, and it seems fair to say that both TA and TB offer high RYW consistency. Certainly TA and TB agree on the combination `(O,O)`. However, TA cannot offer higher consistency than TB in any other scenario, with TA providing slightly lower consistency for some points, even though TA's overall performance is close to TB's over issuing latency. One reason is that TA does not respect the fact "`R + W > RF`" in general (e.g., two strictly ordered `Quorum` write and read cannot guarantee RYW).

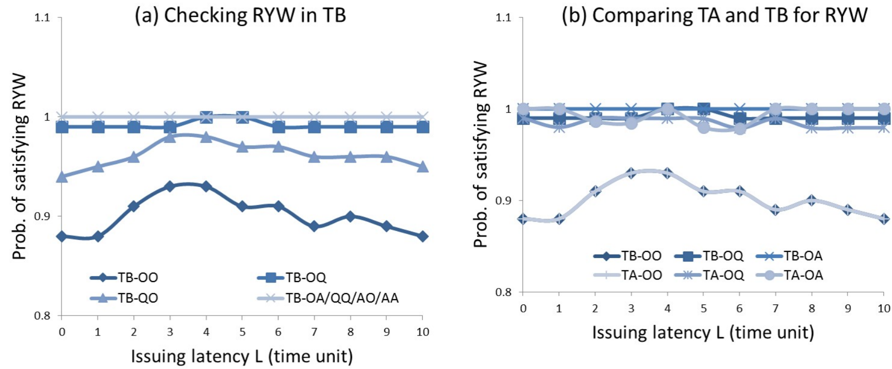

Fig. 5. Probability of Satisfying RYW by Statistical Model Checking

Remark. In summary, our Cassandra model actually achieves much higher consistency (up to SC) than the promised EC, with `QUORUM` reads sufficient to provide up to 100% consistency in almost all scenarios. Comparing TA and TB, it seems fair to say that TA is not a competitive alternative to TB in terms of SC or RYW, even though TA is close to TB in most cases.

Our model, including the alternative design, is less than 1000 lines of code and the time to compute the probabilities for the consistency guarantees is 15 minutes (worst-case). The upper bound for model runtime depends on the confidence level of our statistical model checker (99% confidence level for all our experiments).

5.2 Implementation-based Evaluation of Consistency

Experimental Setup. We deploy Cassandra on a single Emulab [29] server, which means that the coordinator and replicas are separate processes on the same server. We use YCSB [14] to inject read/write workloads. For RYW tests, we use two separate YCSB clients. Our test workloads are read-heavy (that are representative of many real-world workloads such as Facebook's photo storage [9]) with 90% reads, and we vary consistency levels between `ONE`, `QUORUM`, and `ALL`. We run Cassandra and YCSB clients for fixed time intervals and log the results. Based on the log we calculate the percentage of reads that satisfy SC/RYW. Note that other configurations follow our setup for statistical model checking.
Analysis Results for SC. We show the resulting, experimentally computed probability of strongly consistent reads against `L2` (Fig.3) for deployment runs regarding the two strategies (only for `QUORUM` and `ALL` reads). Overall, the results indicate similar trends for the model predictions (Fig.4) and real deployment runs (Fig.6-(a)): for both model predictions and deployment runs, the probability is higher for `ALL` reads than for `QUORUM` reads regarding both strategies, especially when `L2` is low; consistency does not vary much with different strategies.
Analysis Results for RYW. We show the resulting probability of RYW consistent reads against `L` (Fig.3) for deployment runs regarding two strategies. Again, the results indicate similar trends for the model predictions (Fig.5) and real deployment runs (Fig.6-(b)). Both the model predictions and deployment runs show very high probability of satisfying RYW. This is expected since for each client the operations are mostly ordered, and for any read operation from a client, we expect any previous write from the same client to be committed to all replicas. For the deployment runs, we observe that we get 100% RYW consistency, except for scenario (TB-OO), which matches expectations, since `ONE` is the lowest consistency level and does not guarantee anything more than EC. This also matches our model predictions in Fig.5, where we see that the probability of satisfying RYW for scenario (TB-OO) is lower compared to other cases.

Remark. Both the model predictions and implementation-based evaluations reach the same conclusion: Cassandra provides much higher consistency than the promised EC, and TA does not improve consistency compared to TB. Note that the actual probability values from both sides might differ due to factors like hard-to-match experimental configurations, the inherent difference between statistical model checking and implementation-based evaluation[4], and processing

[4] Implementation-based evaluation is based on a single trace of tens of thousands of operations, while statistical model checking is based on sampling tens of thousands of

delay at client/server side that our model does not include. However, the important observation is that the resulting trends from both sides are similar, leading to the same conclusion w.r.t. consistency measurement and strategy comparison.

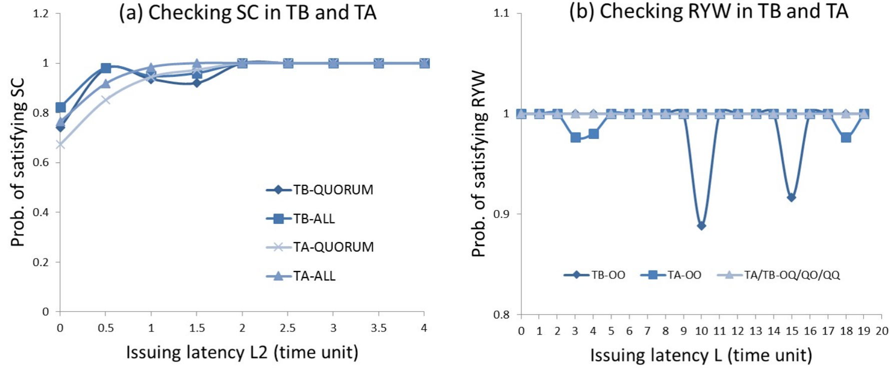

Fig. 6. Probability of Satisfying SC/RYW by Deployment Run

6 Related Work and Concluding Remarks

Model-based Performance Analysis of NoSQL stores. [22] presents a queueing Petri net model of Cassandra parameterized by benchmarking only one server. The model is scaled to represent the characteristics of read workloads for different replication strategies and cluster sizes. Regarding performance, only response times and throughput are considered. [16] benchmarks three NoSQL databases, namely Cassandra, MongoDB and HBase, by throughput and operation latency. Two simple high-level queuing network models are presented that are able to capture those performance characteristics. Compared to both, our probabilistic model embodies the major components and features of Cassandra, and serves as the basis of statistical analysis of consistency with multiple clients and servers. Our model is also shown to be able to measure and predict new strategy designs by both statistical model checking and the conformance to the code-based evaluation. Other recent work on model-based performance analysis includes [8], which applies multi-formalism modeling approach to the Apache Hive query language for NoSQL databases.

Experimental Consistency Benchmarking in NoSQL stores. [23,28,11] propose active and passive consistency benchmarking approaches, where operation logs are analyzed to find consistency violations. [7] proposes probabilistic notions

Monte-Carlo simulations of several operations (that can be considered as a segment of the trace) up to a certain statistical confidence.

of consistency to predict the data staleness, and uses Monte-Carlo simulations to explore the trade-off between latency and consistency in Dynamo-style partial quorum systems. Their focus is more on developing the theory of consistency models. However, we focus on building a probabilistic model for a key-value store like Cassandra itself, and our objective is to compare the consistency benchmarking results with the model-based predictions from our statistical model checking.

Our main focus in this paper has been twofold: (i) to predict what consistency Cassandra can provide by using statistical model checking; and (ii) to demonstrate the predictive power of our model-based approach in key-value store design by comparing statistical model checking predictions with implementation-based evaluations. Our analysis is based on a formal probabilistic model of Cassandra. To the best of our knowledge, we are the first to develop such a formal model.

In this paper we have only looked into two specific consistency guarantees. A natural next step would be to specify and quantify other consistency models by statistical model checking. Depending on the perspective (key-value store providers, users, or application developers), different metrics (e.g., throughput and operation latency) can be used to measure key-value store performance. We also plan to refine our model in order to quantify those metrics. While showing scalability is not the goal of this paper, we believe our results will continue to hold at larger scales. There are resource challenges related to scaling the model checking to larger scales (e.g., parallelizing it in the proper way), and we plan to solve this in our future work. More broadly, our long-term goal is to develop a library of formally specified executable components embodying the key functionalities of NoSQL key-value stores (not just Cassandra), as well as of distributed transaction systems [18]. We plan to use such components and the formal analysis of their performance to facilitate efficient exploration of the design space for such systems and their compositions with minimal manual effort.

References

1. Cassandra, `http://cassandra.apache.org`
2. DB-Engines, `http://db-engines.com/en/ranking`
3. MongoDB, `http://www.mongodb.org`
4. Redis, `http://redis.io`
5. Agha, G.A., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. Electr. Notes Theor. Comput. Sci. 153(2), 213–239 (2006)
6. AlTurki, M., Meseguer, J.: Pvesta: A parallel statistical model checking and quantitative analysis tool. In: CALCO. pp. 386–392 (2011)
7. Bailis, P., Venkataraman, S., Franklin, M.J., Hellerstein, J.M., Stoica, I.: Probabilistically bounded staleness for practical partial quorums. Proc. VLDB Endow. 5(8), 776–787 (2012)
8. Barbierato, E., Gribaudo, M., Iacono, M.: Performance evaluation of NoSQL bigdata applications using multi-formalism models. Future Generation Comp. Syst. 37, 345–353 (2014)

9. Beaver, D., Kumar, S., Li, H.C., Sobel, J., Vajgel, P.: Finding a needle in haystack: Facebook's photo storage. In: OSDI 2010. pp. 47–60 (2010)
10. Benson, T., Akella, A., Maltz, D.A.: Network traffic characteristics of data centers in the wild. In: IMC. pp. 267–280 (2010)
11. Bermbach, D., Tai, S.: Eventual consistency: How soon is eventual? An evaluation of amazon s3's consistency behavior. In: Middleware. p. 1 (2011)
12. Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC. p. 7 (2000)
13. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, LNCS, vol. 4350. Springer (2007)
14. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SOCC. pp. 143–154 (2010)
15. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Statistical model checking for composite actor systems. In: WADT. pp. 143–160 (2012)
16. Gandini, A., Gribaudo, M., Knottenbelt, W.J., Osman, R., Piazzolla, P.: Performance evaluation of nosql databases. In: EPEW. pp. 16–29 (2014)
17. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
18. Liu, S., Rahman, M.R., Ganhotra, J., Ölveczky, P.C., Gupta, I., Meseguer, J.: Formal Modeling and Analysis of RAMP Transaction Systems (2015), `https://sites.google.com/site/siliunobi/ramp`
19. Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of cassandra in maude. In: ICFEM. pp. 332–347 (2014)
20. Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of Cassandra in Maude (2014), `https://sites.google.com/site/siliunobi/icfem-cassandra`
21. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. pp. 73–155 (1992)
22. Osman, R., Piazzolla, P.: Modelling replication in nosql datastores. In: QEST. pp. 194–209 (2014)
23. Rahman, M.R., Golab, W., AuYoung, A., Keeton, K., Wylie, J.J.: Toward a principled framework for benchmarking consistency. In: HotDep (2012)
24. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: CAV. pp. 266–280 (2005)
25. Sen, K., Viswanathan, M., Agha, G.A.: VESTA: A statistical model-checker and analyzer for probabilistic systems. In: QEST. pp. 251–252 (2005)
26. Terry, D.: Replicated data consistency explained through baseball. Commun. ACM 56(12), 82–89 (2013)
27. Vogels, W.: Eventually consistent. Commun. ACM 52(1), 40–44 (2009)
28. Wada, H., Fekete, A., Zhao, L., Lee, K., Liu, A.: Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In: CIDR. pp. 134–143 (2011)
29. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: OSDI. pp. 255–270 (2002)
30. Younes, H.L.S., Simmons, R.G.: Statistical probabilistic model checking with a focus on time-bounded properties. Inf. Comput. 204(9), 1368–1409 (2006)