Analyzing Consistency Properties for Fun and Profit

Wojciech Golab, Xiaozhou Li, Mehul A. Shah

HP Laboratories
HPL-2011-6

Keyword(s):

data consistency, algorightms, key-value stores

Abstract:

Motivated by the increasing popularity of eventually consistent key-value stores as a commercial service, we address two important problems related to the consistency properties in a history of operations on a read/write register (i.e., the start time, finish time, argument, and response of every operation). First, we consider how to detect a consistency violation as soon as one happens. To this end, we formulate a specification for online verification algorithms, and we present such algorithms for several well-known consistency properties. Second, we consider how to quantify the severity of the violations, if a history is found to contain consistency violations. We investigate two quantities: one is the staleness of the reads, and the other is the commonality of violations. For staleness, we further consider time-based staleness and operation-count-based staleness. We present efficient algorithms that compute these quantities. We believe that addressing these problems helps both key-value store providers and users adopt data consistency as an important aspect of key-value store offerings.

External Posting Date: June 6, 2011 [Fulltext] Approved for External Publication
Internal Posting Date: June 6, 2011 [Fulltext]
To be published and presented at PODC 2011, San Jose, June 6-8, 2011.

Analyzing Consistency Properties for Fun and Prof t

Wojciech Golab
Hewlett-Packard Labs
Palo Alto, California, USA
wojciech.golab@hp.com

Xiaozhou (Steve) Li
Hewlett-Packard Labs
Palo Alto, California, USA
xiaozhou.li@hp.com

Mehul A. Shah
Hewlett-Packard Labs
Palo Alto, California, USA
mehul.shah@hp.com

HP Labs Technical Report HPL-2011-6

Abstract

Motivated by the increasing popularity of eventually consistent key-value stores as a commercial service, we address two important problems related to the consistency properties in a history of operations on a read/write register (i.e., the start time, f nish time, argument, and response of every operation). First, we consider how to detect a consistency violation as soon as one happens. To this end, we formulate a specif cation for online verif cation algorithms, and we present such algorithms for several well-known consistency properties. Second, we consider how to quantify the severity of the violations, if a history is found to contain consistency violations. We investigate two quantities: one is the staleness of the reads, and the other is the commonality of violations. For staleness, we further consider time-based staleness and operation-count-based staleness. We present eff cient algorithms that compute these quantities. We believe that addressing these problems helps both key-value store providers and users adopt data consistency as an important aspect of key-value store offerings.

1 Introduction

In recent years, large-scale key-value stores such as Amazon's S3 [2] have become commercially popular. A key-value store provides a simple get(key) and put(key,value) interface to the user. Providing strong consistency properties for these operations has become an increasingly important goal [8, 16, 30]. However, the implementations of many key-value stores are proprietary and, as such, opaque to the users. Consequently, the users cannot reason about the implementations so as to be conf dent about their correctness. Instead, they can only test the system empirically, and analyze the test results (e.g., traces of operations) to see if it is delivering the promised level of consistency. To be useful, such consistency analysis should address two important problems. One, the analysis should reveal consistency violations as soon as they happen so that corrective actions can be taken (e.g., tuning the consistency level for future operations [3]). Two, the analysis should quantify the severity of violations. If consistency is part of the Service Level Agreement (SLA), and the severity of violations can be quantif ed in some way, then some proportional compensation, monetary or otherwise, may be negotiated between the user and the service provider (hence the title of the paper).

We model a key-value store by a collection of read/write registers where each key identifies a register and the get/put requests translate into read/write operations on the appropriate register. Given a history of operations (i.e., the start time, finish time, argument, and response of every operation) on a read/write register, how do we determine whether the history satisfies certain consistency properties such as atomicity (i.e., linearizability) [18, 22]? The basic decision problem, which seeks only a yes/no answer, has been addressed in the literature [5, 14, 23]. In this paper, we are interested in two questions beyond the decision problem. First, how to detect a consistency violation as soon as it happens, rather than analyze the entire history potentially long after the fact? Second, if the history is found to violate the desired consistency property, how to quantify the severity of the violations? To our knowledge, these problems have not been addressed in literature, mainly because (1) storage as a service is new, and (2) traditionally, stores avoid inconsistency altogether rather than briefly sacrifice consistency for better availability.

In this paper, after laying out the model and definitions (Section 2), we present online consistency verification algorithms for several well-known consistency properties, namely safety, regularity, atomicity, and sequential consistency (Section 3). The distinctive feature of these algorithms is that they operate not by processing the entire history at once, but rather by processing a history incrementally as events (i.e., start or finish of an operation) occur, and reporting violations as they are detected. We note that our online algorithms do not control what or when operations are to be issued: they merely analyze the histories passively and report violations according to formal consistency property definitions.

We then propose several ways to quantify the severity of atomicity violations in a history (Sections 4 and 5). The first quantity we consider is the maximum staleness of all the reads (Section 4). Staleness attempts to capture how much older the value read is compared to the latest value written. We propose two definitions of "older than" in this context. One is based on the passage of time. The second is based on the number of intervening writes, a notion that coincides precisely with the k-atomicity concept proposed by Aiyer et al. [1]. We present algorithms that compute the maximum time-based staleness and, for special cases, operation-count-based staleness.

The second quantity for evaluating the severity of violations is the commonality of violations. Defining this concept precisely is nontrivial as violations are not easily attributed to individual operations. Instead, we define commonality as the minimum "proportion" of the history that must be removed in order to make the history atomic (Section 5). To simplify the problem computationally, we do not consider the removal of individual operations but rather of entire *clusters*—groups of operations that read or write the same value. We give two formulations. In the unweighted formulation, we treat all clusters equally and try to remove the smallest subset of clusters. We solve this problem using a greedy algorithm (Section 5.1). In the weighted formulation, we weigh clusters according to their size (i.e., number of operations in the cluster), and we try to remove a subset of clusters with minimum total weight. We solve this problem using a dynamic programming algorithm (Section 5.2). Finally, we survey related work (Section 6) and conclude the paper with a discussion of some open problems (Section 7). Due to space limitations, we defer all proofs of correctness to the appendices.

2 Model

A collection of client machines interact with a key-value store via two interfaces: get(key) and put(key,value), where key and value are uninterpreted strings. In order to determine whether the key-value store provides certain consistency properties, a client machine can timestamp when it sends out a get/put request and when it receives a response, and can record the value read or written. Since different client machines can access the same key, the individual client histories are sent to a centralized *monitor* where the individual histories

are merged. Furthermore, since accesses to different keys are independent of each other, the monitor groups operations by key and then examines whether each group satisfies the desired consistency properties. We further assume that all client machines have well-synchronized clocks (often accomplished by time synchronization protocols such as NTP), or have access to a global clock, so that client timestamps can be considered to represent real time. See the technical report [15] for additional discussions on this scenario and our assumptions.

We model a key-value store as a collection of read/write registers, each identified by a unique key. We consider a collection of operations on such a register. Each operation, either a read or a write, has a start time, finish time, and value. The value of a write is the value written to the register and the value of a read is the value obtained by the read. Note that the value of a write is known at the start of the write, but the value of a read is known only at the finish of the read. This distinction is important for online verification of consistency properties (Section 3). We assume that all writes assign a distinct value. We make this assumption for two reasons. First, in our particular application, all writes can be tagged with a globally unique identifier, typically consisting of the local time of the client issuing the write followed by the client's identifier. Therefore, this assumption does not incur any loss of generality. Second, when the values written are not unique, the decision problem of verifying consistency properties is NP-complete for several well-known properties, in particular atomicity and sequential consistency [7, 14, 27].

We next define some terminology and notations. An *event* is the start or finish of a read or write operation. We assume that the system has the ability to pair up start/finish events for the same operation. We denote the start and finish events of an operation op by $|op$ and $op|$ respectively, and the start and finish times of op by $op.s$ and $op.f$ respectively. The start of a read whose return value is not yet known is denoted by $|r(?)$. We assume that all start and finish times are unique. We say that operation op *time-precedes* (or simply precedes) operation op', written as $op < op'$, if $op.f < op'.s$. We say that op *time-succeeds* (or simply succeeds) op' iff op' time-precedes op. If neither $op < op'$ nor $op' < op$, then we say op and op' are *concurrent* with each other. A *history* is a collection of events that describes a collection of operations, some of which may not have finished. Without loss of generality, we assume that a history begins with an artificial pair of start/finish events for a write that assigns the initial value of the register. For a read, its *dictating write* is the (unique) write that assigns the value obtained by the read. Typically, every read in a history has a dictating write, otherwise either the history contains incomplete information or the system is buggy. For a write, the set of reads that obtain the value written is called the *dictated reads* of the write. A write can have any number of dictated reads (including zero). The time-precedence relation defines a partial order on the operations in a history H. A total order of the operations in H is called *valid* if it conforms to the time-precedence partial order.

More discussions can be found in Appendix D.

3 Online verification of consistency properties

Determining whether a given history satisfies certain consistency properties such as atomicity has been addressed in the literature [5, 14, 23]. However, known solutions are *offline* algorithms in the sense that they analyze the entire history at once, even though a violation may occur in some short prefix of the history. In this section, we investigate how to detect a violation as soon as one happens, and we present efficient algorithms that achieve this goal for three well-known consistency properties: safety, regularity, and atomicity. We also discuss the complications associated with verifying sequential consistency in an online manner.

3.1 Specifying online verification algorithms

Online verification algorithms work by inspecting the start event and finish event of each operation one by one, in time order, and determine on-the-fly whether a violation has happened. In contrast to an offline algorithm, which simply indicates whether a history satisfies some consistency property, we would like an online algorithm to output more fine-grained information. For example, if a history initially satisfies the consistency property and then fails to satisfy it (after some long-enough prefix), then the output from the online algorithm should be different from the case where violations occur from the beginning. From a theoretical perspective, another attractive feature for an online algorithm is the ability to report meaningful information for infinitely long histories—a property that an offline algorithm cannot satisfy because its output summarizes upon termination the entire input history in one yes/no answer.

To meet the requirements discussed above, we define an online algorithm as one whose input is a sequence of events $H = \langle e_1, \ldots, e_n \rangle$, and whose output is a sequence $\Gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$, where $\gamma_i \in \{\textit{good}, \textit{bad}\}$. The output value γ_i provides information about the prefix $\langle e_1, \ldots, e_i \rangle$ of H, which we denote by H_i. For a given history H, we call H *good* with respect to a consistency property P (which is external to our specification) if H satisfies P, and *bad* with respect to P otherwise. Intuitively, $\gamma_i = \textit{bad}$ indicates that H_i is bad, and furthermore the consistency violation can be attributed in some way to the last event e_i. An output value $\gamma_i = \textit{good}$ indicates that e_i does not introduce any additional consistency violations, but does not say whether H_i is good or bad. This is because it is possible that a violation has occurred in some short prefix of H_i, and no other violations have occurred since then. The above intuitive notion is captured by the following formal specification for how a correct online verification algorithm should behave:

Specification 3.1 *Let $H = \langle e_1, \ldots, e_n \rangle$ be an input history for an online verification algorithm for some consistency property P. Let $\Gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$ be the sequence of good/bad output values produced by the algorithm, one for each event in H. For any i, let $\tilde{H}_i$ denote the history obtained by taking H_i and removing every event e_j, where $1 \leq j < i$, such that $\gamma_j = \textit{bad}$, along with its matching start/finish event. Then for any i such that $1 \leq i \leq n$, $\gamma_i = \textit{good}$ iff $\tilde{H}_i$ is good with respect to P.*

It is easy to show that, for safety, regularity, atomicity, and sequential consistency, $\gamma_i = \textit{bad}$ only if e_i is the finish event for a read. This is not because write operations cannot participate in violations, but rather because only reads can reveal such violations, namely through their return values. For this reason we adopt the convention in this section that violations are attributed to reads only. We also think of an online algorithm as deciding for each read whether it has caused a violation with respect to the consistency property under consideration. If so, the algorithm outputs *bad* when it processes the read's finish event, and subsequently continues as if the offending read did not happen at all. We remark that this way of counting violations, informally speaking, does not necessarily report the smallest possible set of violations. For example, consider the history depicted in Figure 1, which is not atomic. An online verification algorithm greedily considers $r_1(1)$ valid, and classifies $r_2(0)$ and $r_3(0)$ as violations. We could instead suppose that $w(1)$ takes effect before $w(0)$, and attribute the violation to $r_1(1)$. Deciding which option is better at the time when $r_1(1)$ finishes would require seeing into the future. Thus, we cannot expect an online verification algorithm to make the best decision on-the-fly.

For the properties of safety, regularity and atomicity, it is also straightforward to show that any online verification algorithm ALG satisfying Specification 3.1 also has the following properties: (1) Validity: For any good history H, the output vector Γ of ALG on H contains all *good*. (2) Completeness: For any bad history H, the output vector Γ of ALG on H contains at least one *bad*. The same is not true for sequential consistency (i.e., validity does not hold), and we comment on that in more detail in Section 3.6.

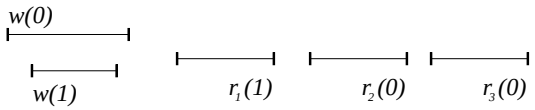

Figure 1: Online verif cation of a non-atomic history.

3.2 Efficiency of online verification algorithms

A straightforward approach to devising an online verif cation algorithm is to simply apply an off ine algorithm repeatedly on successively longer pref xes of the history. Assuming that the online algorithm discards all reads that cause violations, it is correct in the sense of satisfying Specif cation 3.1, provided of course that the off ine algorithm is correct. However, this approach may be ineff cient. For example, consider the atomicity verif cation algorithm of Gibbons and Korach [14] (simply referred to as the GK algorithm henceforth), which runs in $O(n \log n)$ time on an n-operation history. If we use the GK algorithm for online checking, then each stage of checking takes $O(n' \log n')$ time, where n' is the length of the pref x, and altogether the algorithm takes $O(n^2 \log n)$ time.

The key to eff cient online verif cation lies in managing the bookkeeping information in various data structures (e.g., zones [14], a value graph [23], or an operation graph [5]). Instead of throwing away and reconstructing these data structures each time the history grows by one event, we modify the data structures and try to perform computation on them incrementally. Furthermore, to reduce the time and space complexity of the computation we try to discard any information that is no longer needed. For example, suppose that we are checking for atomicity. Consider two successive writes $w(0)w(1)$. Upon the f nish of $w(1)$, we observe that reads starting after that time cannot return the value 0 without causing a violation. (Recall our assumption that each write assigns a unique value.) Therefore, $w(0)$ can be "wiped from the books" once $w(1)$ ends, if there are no ongoing reads at that time. This makes any future read returning 0 appear as though it lacks a dictating write, causing the algorithm to report a violation for that read. Another observation is that a f nished read can always be discarded, as long as the constraints that the read places on the order of writes are properly recorded. These ideas are explained in detail in the sections below.

3.3 Verifying safety

Safety is one of the weakest consistency properties. A history is *safe* iff there exists a valid total order on the operations such that (1) a read not concurrent with any writes returns the value of the latest write before it in the total order, and (2) a read concurrent with one or more writes may return an arbitrary value.

The online safety verif cation algorithm is presented in Algorithm 1. We use shorthands such as "$e_i = |w(a)$" to mean "event e_i is the start of a write of value a" (line 3). The algorithm maintains a set of values. These values are those that may be obtained by ongoing or future reads, and we call these values *allowable values*. For each allowable value a, the algorithm maintains a variable $w[a]$ that keeps track of the start time and f nish time of $w(a)$ (line 8). As a slight abuse of notation, we use square brackets to denote variables and braces to denote operations; this convention is adopted throughout the rest of the paper. We use "$w[b] < w[a]$" to mean the write of b precedes that of a (line 9). The algorithm maintains some additional data structures: (1) I: the set of reads that can be ignored (because they are concurrent with some writes), (2) R: the set of pending reads, and (3) nw: the number of pending writes. The algorithm creates new allowable values as new writes are seen. However, the algorithm also discards old values (line 18) as soon as those values are determined to be not allowable. The algorithm outputs *bad* when a read obtains a value that is not in the set of allowable values (line 19).

Algorithm 1: Online safety verification

Input: sequence of events $\langle e_1, e_2, \dots \rangle$
Output: sequence of *good*/*bad* values $\langle \gamma_1, \gamma_2, \dots \rangle$
Init: $I = R = \emptyset$, $nw = 0$

upon event e_i **do**
  $\gamma_i :=$ *good*;
  if $e_i = |w(a)$ **then**
    $nw := nw + 1$;
    $I := I \cup R$
  else if $e_i = w(a)|$ **then**
    $nw := nw - 1$;
    create $w[a]$; $w[a].(s, f) := w(a).(s, f)$;
    foreach $(b : w[b] < w[a])$ **do** discard $w[b]$ **end**
  else if $e_i = |r(?)$ **then**
    add $r(?)$ to R;
    if $nw > 0$ **then** add $r(?)$ to I **end**
  else if $e_i = r(a)|$ **then**
    remove $r(a)$ from R;
    if $r \in I$ **then**
      remove $r(a)$ from I
    else
      if $\exists w[a]$ **then** $\forall b, b \neq a$: discard $w[b]$
      else $\gamma_i :=$ *bad*
      end
    end
  end;
  output γ_i
end

3.4 Verifying atomicity

We say that a history is *atomic* iff there exists a valid total order on the operations such that every read returns the value of the latest write before it in the total order. Offline verification of atomicity has been addressed in the literature before [5, 14, 23]. The online algorithm presented in this section uses core ideas from an offline algorithm, and some additional ideas on discarding obsolete values. Fundamentally, it does not matter which offline algorithm we begin with, but in what follows, we adopt the GK algorithm [14]. This is because the GK algorithm lends itself most conveniently to incremental computation. More discussion on offline algorithms can be found in the related work section (Section 6).

The GK algorithm works as follows. For a history, the set of operations that take the same value is called a *cluster*. Let $C(a)$ denote the cluster for value a. Let $\bar{s}(a)$ be the maximum start time of all the operations in $C(a)$, that is, $\bar{s}(a) = \max\{op.s : op \in C(a)\}$. Let $\underline{f}(a)$ be the minimum finish time of all the operations in $C(a)$, that is, $\underline{f}(a) = \min\{op.f : op \in C(a)\}$. The *zone* for a value a, denoted by $Z(a)$, is the closed interval of time between $\underline{f}(a)$ and $\bar{s}(a)$. If $\underline{f}(a) < \bar{s}(a)$, this zone is called a *forward zone* and spans from $\underline{f}(a)$ to $\bar{s}(a)$. Otherwise, the zone is called a *backward zone* and spans from $\bar{s}(a)$ to $\underline{f}(a)$. We use $Z.l$ and

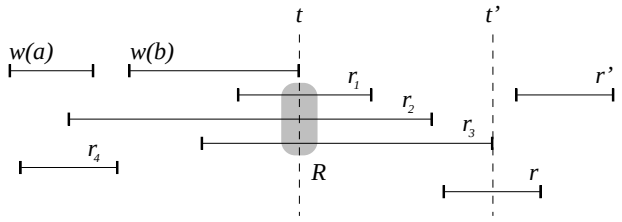

Figure 2: When to discard bookkeeping information.

$Z.r$ to denote the left endpoint and right endpoint of zone Z. In other words, if $Z(a)$ is a forward zone, then $Z(a).l = \underline{f}(a)$ and $Z(a).r = \bar{s}(a)$. If $Z(a)$ is a backward zone, then $Z(a).l = \bar{s}(a)$ and $Z(a).r = \underline{f}(a)$. We write $Z_1 < Z_2$ iff $Z_1.r < Z_2.l$. We use $\overrightarrow{Z}$ to denote a forward zone and $\overleftarrow{Z}$ to denote a backward zone.

We say that two zones Z_1 and Z_2 *conflict* with each other, denoted by $Z_1 \not\sim Z_2$, iff they are both forward zones and they overlap, or one is a forward zone and the other is a backward zone contained entirely in the former forward zone. Two zones are *compatible* with each other, written as $Z_1 \sim Z_2$, iff they do not conf ict. According to this def nition, two backward zones never conf ict. Gibbons and Korach [14] show that a history is atomic iff (1) every read has a dictating write, (2) no read precedes its dictating write, and (3) all pairs of zones are compatible.

Our online algorithm is based on the GK algorithm, and has the ability to discard obsolete information. The technique for identifying such information is based on the following observation, illustrated in Figure 2. Consider a write $w(a)$, which is succeeded by another write $w(b)$. Let R be the set of ongoing reads when $w(b)$ f nishes. We observe that at the time when all the reads in R have f nished, no ongoing or future reads can obtain the value a without causing a violation. To see this, let t be $w(b)$'s f nish time and t' be the largest f nish time of all the reads in R. Consider an ongoing read r at t'. This read does not belong to R because at t', all reads in R have f nished. Therefore, r starts after time t, yet at time t, value a has been superseded by value b. Thus, r should not return a. Next, consider a read r' that starts after t'. Since $t < t'$, by the same argument, r' should not return value a either. In other words, at time t', we can discard all information related to value a, and if subsequently a read returns a, we can immediately report a violation. Based on this observation, we def ne the α *set of value* a, denoted by $\alpha(a)$, to be the set of ongoing reads at the earliest f nish time of all the writes that succeed $w(a)$. For the example in Figure 2, $\alpha(a) = \{r_1, r_2, r_3\}$. In an online algorithm, we can use a variable $\alpha[a]$ to keep track of which reads in $\alpha(a)$ are still ongoing, and once this set becomes empty, value a can be discarded. Due to the nature of online algorithms, $\alpha(a)$ is initially undef ned (because the algorithm has not seen any writes that succeed $w(a)$); for that period of time the $\alpha[a]$ variable is nil (not $\emptyset$).

Clearly, how much space can be saved depends on the history. However, we observe that new writes cause new variables to be created, but in the meantime, they increase the chances that old variables can be discarded. Therefore, we expect that in long histories, space saving in practice is substantial.

Algorithm 2 implements the above ideas. The f nish event of a read is determined to be *bad* under two circumstances: (1) when a read obtains a value not allowable (line 19), and (2) when an updated zone conf icts with an existing, fully formed zone (line 24). In the latter case the algorithm undos the update to the zone so that it can "pretend" that this bad event didn't happen. In both cases, the present read is removed from R and α-sets (lines 28 to 32), so the effect of the matching start event is also eliminated (recall Specif cation 3.1). The rest of the algorithm mainly adds and removes the bookkeeping information as new events arrive, following the ideas described above. For example, upon the start of a write (line 3), the algorithm creates the corresponding data structures, leaving $\underline{f}$ of the new zone to be ∞ (i.e., "undef ned")

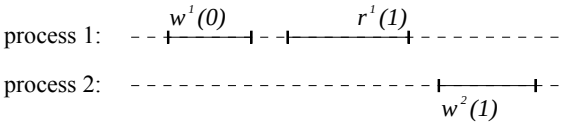

Figure 3: A sequentially consistent history that is not safe.

as no f nish event for that zone has been seen. As other events are seen, the algorithm initializes (line 12) and updates (line 30) the α-sets accordingly so that old values can be discarded in a timely manner.

3.5 Verifying regularity

We say that a history is *regular* iff there exists a valid total order of the operations such that a read returns the value of the latest write before it in the total order, except that a read concurrent with one or more writes may return the value of one of the writes concurrent with the read. Online verif cation of regularity is similar to verif cation of atomicity, except that we discard immediately any read that returns the value of some write concurrent with the read. We omit the details due to space limitations.

3.6 Verifying sequential consistency

We say that a history is *sequentially consistent* iff there exists a total order on the operations such that (1) the total order is consistent with process order (i.e., operations by the same process are placed in the same order as they are issued), and (2) a read returns the value of the latest write before it in the total order. The total order need not be valid (i.e., conform to the real-time partial order) as long as property (1) holds. In order to def ne process order, we assume that each process issues one operation at a time.

Note that to verify sequential consistency, we need to know which process issues which operation in the history, in contrast to the previous three consistency properties. In this section, we assume that the history includes such information, and that we know in advance the full set of processes that may issue operations. Both assumptions can be realized easily in practice.

Off ine verif cation of sequential consistency is NP-complete if writes can assign duplicate values [14, 27], but admits straightforward solutions if we assume that each write assigns a unique value. For example, we can use an operation graph approach [5]. In particular, we can model each operation as a vertex in a directed graph. We add edges to this graph in the following three steps: (1) $op \rightarrow op'$ if op and op' are from the same process and op precedes op', (2) $w(a) \rightarrow r(a)$ for all values a, and (3) $w(a) \rightarrow w(b)$ if $w(a) \rightarrow r(b)$, for all values a and b. It is easy to show that the history is sequentially consistent iff each read has a dictating write and the resulting graph is a DAG.

Online verif cation of sequential consistency poses unique challenges owing to weaker constraints on the total order of operations, which need not conform to the real-time partial order. The fundamental problem is illustrated by the history depicted in Figure 3. This history violates safety, regularity, and atomicity because the real-time partial order of operations forces $r^1(1)$ before $w^2(1)$ in any total ordering, meaning that $r^1(1)$ precedes its dictating write (the superscripts denote the processes issuing the operations). On the other hand, the history is sequentially consistent, and so in online verif cation we would like the algorithm to output a sequence of *good* values for this particular input. (See the "Validity" property def ned in Section 3.1.) This is problematic because when the online algorithm sees the pref x $w^1(0)r^1(1)$, there is no dictating write for $r^1(1)$ and so Specif cation 3.1 stipulates $\gamma_4 = \mathit{bad}$.

We work around this problem by making the following simplifying assumption: in the real-time partial order of operations, a read never precedes its dictating write. In practical terms, a key-value store can break this assumption only if there is a software bug causing reads to return values that have not yet been written, or if there is signif cant clock skew among servers, making a read appear to precede its dictating write when events are collected at a centralized monitor. We ignore these possibilities as they are orthogonal to the core problem of determining what consistency property a key-value store actually provides when it is designed (correctly) to provide some weaker property such as eventual consistency.

Even with our simplifying assumption, we still face the problem that in online verif cation of sequential consistency, timing information cannot be used to determine which operations can be discarded. Instead, the following rule can be used to discard obsolete operations: As soon as there is an operation $op(a)$ such that each process has executed some operation that is downstream of op in the DAG (possibly $op(a)$ itself), then any operation upstream of $op(a)$'s dictating write (which can be $op(a)$ itself if $op(a)$ is a write) can be discarded from the graph. This rule is correct because no ongoing or future reads can return a or any value whose dictating write is upstream of $op(a)$'s dictating write. To see this, suppose otherwise, and let b be the value returned by the read such that b's dictating write is upstream of a's. Then from the above rules for adding edges, it is easy to construct a cycle involving $w(b)$ and $w(a)$, indicating a violation of sequential consistency. The verif cation algorithm for sequential consistency that implements this rule under our simplifying assumption is straightforward, and we omit the details due to space limitations.

4 Quantifying staleness

What can we do if we discover that a history contains consistency violations? We can try to quantify the severity of the violations. In this paper we consider two quantities: staleness of reads and commonality of violations. This section addresses the former, and the next section addresses the latter. Informally, the *staleness of a read* quantif es the "distance" between the write operation that assigns the value returned by the read, and the operation that writes the latest value (in some valid total order of the operations). We can then def ne the *staleness of a history* as the maximum staleness over all the reads in the history. In this paper we consider two natural ways to formalize the notion of "distance": (1) by measuring the passage of time, and (2) by counting the number of intervening writes. We elaborate on these two approaches in the subsections that follow. (Note: From here on, the algorithms that we consider are no longer online algorithms.)

4.1 Time-based staleness

In this section, we discuss Δ-*atomicity*, a consistency property that we feel is appropriate for reasoning about eventually consistent read/write storage systems. This property is a generalization of atomicity, and is def ned for any non-negative real number Δ. Informally, Δ-atomicity allows a read to return either the value of the latest write preceding it, or the value of the latest write as of "Δ time units ago." Thus, if $\Delta > 0$, Δ-atomicity is strictly weaker than atomicity [22], and if $\Delta = 0$, it is identical to atomicity. We now give a more precise def nition.

We f rst observe that some histories may contain the following "bad" reads: (1) a read obtains a value that has never been written, and (2) a read precedes its own dictating write. We call a history *simple* iff it contains neither anomaly. For non-simple histories, we def ne their staleness to be ∞. It is straightforward to check if a history is simple, and so in what follows we only consider simple histories.

For a simple history H, let H_Δ be the history obtained from H by decreasing the start time of each

read by Δ time units. We say that H is Δ-*atomic* iff H_Δ is atomic. Therefore, given a history H and Δ, checking if H is Δ-atomic is reduced to computing H_Δ from H and checking if H_Δ is atomic. (For $\Delta = 0$, the reduction is trivial.) The following captures some useful properties of Δ-atomicity.

Fact 4.1 *(1) Two compatible zones remain compatible if we decrease the start times of the reads in these zones by arbitrary amounts. (2) For any simple history H, there exists a $\Delta \geq 0$ such that H is Δ-atomic. (3) For any simple history H and $0 \leq \Delta \leq \Delta'$, if H is Δ-atomic then it is also Δ'-atomic.*

We state in the following lemma an alternative (and somewhat more intuitive) definition of Δ-atomicity:

Lemma 4.2 *A simple history H is Δ-atomic iff there exists an assignment of a unique timestamp to each operation such that: each timestamp is within the operation's time interval, and a read with timestamp t obtains the value of the write with the greatest timestamp $t' < t - \delta_t$ for some δ_t such that $0 \leq \delta_t \leq \Delta$.*

For the remainder of this section, we focus on the problem of computing for any simple history H the smallest $\Delta \geq 0$ that makes H Δ-atomic, and hence makes H_Δ atomic. Since shifting the start times of read operations (by increasing Δ) may break the assumption that start and finish times are unique (see Section 2), we must carefully handle corner cases where two zones share an endpoint. To that end, we adopt the convention that two forward zones are compatible if they overlap at exactly one point, and a forward zone is compatible with any backward zone that shares one or both endpoints with the forward zone.

To compute the optimal Δ, we propose a solution based on the GK algorithm for verifying atomicity [14] (see Section 3.4). Given a simple history H, we first compute the set of zones $\mathcal{Z}$. For each pair of distinct zones $Z_1, Z_2 \in \mathcal{Z}$, we assign a score $\chi(Z_1, Z_2) \geq 0$, which quantifies the severity of the conflict between Z_1 and Z_2, and has the property that $\chi(Z_1, Z_2) = \chi(Z_2, Z_1)$. Intuitively, $\chi(Z_1, Z_2)$ is the minimum value of Δ that eliminates any conflict between Z_1 and Z_2.

To understand how χ is computed, consider first the effect of decreasing the starting times of all reads in H by Δ. For a zone that does not contain any reads, there is no effect. For a forward zone, which necessarily contains at least one read, the right endpoint of the zone shifts to the left, up to the limit where the forward zone collapses into a single point. Once this limit is reached, the zone becomes a backward zone and behaves as we describe next. For any backward zone containing at least one read, the left endpoint of the zone shifts to the left, up to the limit where the left endpoint coincides with the start of the dictating write. Beyond this limit there is no effect. Thus, for large enough Δ, all zones become backward zones, and there are no conflicts.

The score function $\chi(Z_1, Z_2)$ is defined precisely as follows. Let $Z_1 \cap Z_2$ denote the time interval corresponding to the intersection of Z_1 and Z_2, and let $|Z_1 \cap Z_2|$ denote length of this intersection interval.

- If $Z_1 \sim Z_2$, then $\chi(Z_1, Z_2) = 0$.
- If Z_1, Z_2 are conflicting forward zones, then $\chi(Z_1, Z_2) = |Z_1 \cap Z_2|$. (Intuitively, to resolve the conflict we shift the right endpoint of the zone that finishes earliest to the left, until either this zone becomes a backward zone, or its right endpoint meets the left endpoint of the other zone.)
- If Z_1 is a forward zone and Z_2 is a conflicting backward zone that contains at least one read and whose dictating write begins before $Z_1.l$, then

  $$\chi(Z_1, Z_2) = \min\left(Z_1.r - Z_2.r, Z_2.l - Z_1.l\right).$$

  (Intuitively, to resolve the conflict we shift $Z_1.r$ and $Z_2.l$ to the left by the smallest amount ensuring that Z_1 no longer contains Z_2.)

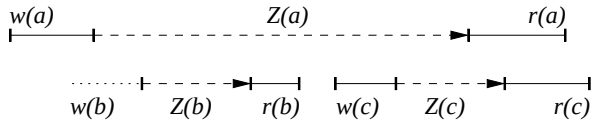

Figure 4: A history that may or may not be 2-atomic.

- If Z_1 is a forward zone and Z_2 is a conf icting backward zone of any other kind, then $\chi(Z_1, Z_2) = Z_1.r - Z_2.r$. (Intuitively, to resolve the conf ict we shift $Z_1.r$ to the left until Z_1 no longer contains Z_2. Shifting $Z_2.l$ does not help.)

It follows from the discussion above that increasing Δ can only eliminate existing conf icts, and never creates new ones. Consequently, choosing $\Delta = \max\{\chi(Z_1, Z_2) : Z_1, Z_2 \in \mathcal{Z}\}$ eliminates simultaneously all conf icts among the zones in H_Δ, and ensures that H_Δ is atomic. Furthermore, no smaller Δ has the latter property. These results are captured in Theorem 4.3 below.

Theorem 4.3 *Let H be a simple history and $\mathcal{Z}$ be the set of zones corresponding to the clusters in H (see Section 3.4). Define $\Delta_{opt} = \max\{\chi(Z_1, Z_2) : Z_1, Z_2 \in \mathcal{Z}\}$. Then $H_{\Delta_{opt}}$ is atomic. Furthermore, $\Delta_{opt} = \min\{\Delta : H_\Delta \text{ is atomic}\}$.*

Computing Δ_{opt} is a straightforward exercise of tabulating the scores for all pairs of distinct zones and taking the maximum of the scores.

4.2 Operation-count-based staleness

A different way to quantify the staleness of a read is to count how many writes intervene between the read and its dictating write. For this purpose, Aiyer et al. [1] have def ned the notion of k-atomicity. A history is called *k-atomic* iff there exists a valid total order of the operations such that every read obtains the value of one of the k latest writes before it in the total order. By this def nition, ordinary atomicity is identical to 1-atomicity. We are not aware of any algorithm, online or off ine, that verif es whether a history is k-atomic, for $k > 1$. In this section, we present the f rst 2-atomicity verif cation algorithm (an off ine algorithm), albeit it only works for a special class of histories.

Our f rst attempt is to extend existing 1-atomicity verif cation algorithms to $k > 1$, but it is not obvious to us how to do that. For example, consider the GK algorithm [14]. When $k = 2$, it is no longer suff cient to just look at the zones as in the GK algorithm. To see this, consider the history depicted in Figure 4, where the start time of $w(b)$ is left unspecif ed. By the GK algorithm, this history is not atomic (because there are overlapping forward zones), but whether or not it is 2-atomic depends on the start time of $w(b)$. If $w(b)$ starts after $w(a)$ f nishes, then the history is not 2-atomic, because $w(a)$ is separated from $r(a)$ by $w(b)$ and $w(c)$. However, if $w(b)$ starts before $w(a)$ f nishes, then the history is 2-atomic, in which case the total order would be $w(b)w(a)r(b)w(c)r(a)r(c)$.

In what follows, we present an off ine 2-atomicity verif cation algorithm for nice histories. We call a history *nice* if (1) it is simple (see Section 4.1), (2) each write has at least one dictated read, and (3) each read succeeds its dictating write. We f rst observe that, for these histories, we can assume without loss of generality that each write has exactly one dictated read, because otherwise, we can condense the history by keeping, for each write, the dictated read that starts last and remove the other dictated reads. If we can construct a k-atomic total order for the condensed history, then we can add back the removed reads to the total order (by adding back the removed read somewhere between its dictated write and that write's

surviving dictated read) while preserving k-atomicity. On the other hand, if the condensed history is not 2-atomic, then neither is the original history. Therefore, it suffices to consider nice histories where each write has exactly one dictated read.

Given a nice history, the algorithm's overall strategy is to construct a total order by laying out the writes from right to left in the total order, generally in the order of decreasing finish times, but in the meantime being mindful about the additional constraints imposed by previously placed writes. The full algorithm is presented in Algorithm 3.

All operations start as "unpicked," and the algorithm picks the operations from the history and puts them into the total order, which is initially empty. When the algorithm starts, it picks a write $w(a)$ with the largest finish time as the rightmost write in the total order. Once it picks $w(a)$, it computes the set S, the set of unpicked reads that succeed $w(a)$ in the history. Note that those reads in S have to follow $w(a)$ in the total order, given that $w(a)$ is the next write to be laid out. The algorithm then prepends $w(a)$ and S to the total order such that all the reads in S follow $w(a)$ in the total order. Let $R = S \setminus \{r(a)\}$. If R is not empty, then it imposes additional constraints on what the next write should be, because of the 2-atomicity requirement. In particular, if $|R| > 1$, then it means that, in order to keep the 2-atomicity requirement, we have to lay out multiple writes at the next step, an obviously impossible task. Hence, the algorithm outputs *bad*. If $|R| = 1$, then the algorithm is obliged to lay out the dictating write for the lone read in R: there is no other choice. If $|R| = 0$, then the algorithm is free to pick any unpicked write and it picks the one with the largest finish time. The intuition of picking such a write is that, compared to other choices, such a write forces the fewest number of reads that have to be included in S, which in turn makes the algorithm more likely to continue. The algorithm continues until all operations are picked.

5 Quantifying commonality

In some sense, the staleness notion that we consider in the previous section focuses on the worst violation in a history. In this section, we consider how common violations are in a history. To this end, our first attempt is to partition the operations in a history into two classes, "good" and "bad," such that the removal of bad operations makes the remaining sub-history atomic. Then we can compute the smallest subset of bad operations and treat its size as the number of atomicity violations. However, classifying operations as good or bad is problematic because atomicity violations are not easily attributed to specific operations. Consider for example the history $w(a)w(b)r(a)$. On the one hand, $r(a)$ is bad because it returns a value other than the one most recently written. On the other hand, we can also blame $w(b)$, which completes before $r(a)$ but does not appear to take effect. Thus, of the three operations that exhibit the atomicity violation, there are two that, if removed individually, make the remaining sub-history atomic.

This example motivates a method of classifying operations as good or bad other than the one based on individual operations. To that end, we propose to group operations by their values. In the terminology of the GK algorithm [14], the set of operations that take the same value (i.e., a write plus zero or more reads) is called a *cluster*. We propose to classify entire clusters as good or bad, and compute an optimal subset of clusters whose removal makes the remaining sub-history atomic. There are two ways to define "optimal" in this context. In the unweighted formulation, each cluster is counted equally, and we try to maximize the number of clusters leftover. In the weighted formulation, we use the number of operations as the weight of a cluster, and we try to maximize the total weight of the clusters leftover. In what follows, we present a greedy algorithm for the former problem, and a dynamic programming algorithm for the latter. We note that these algorithms are not online algorithms.

Our algorithms operate on simple histories, which are defined in Section 3. Given an arbitrary history,

a preprocessing stage is used to obtain simple history—any cluster containing one or more reads but no dictating write is removed, and any cluster where a read precedes its dictating write is removed. These steps are necessary in the context of algorithms that select or discard entire clusters, and they are done identically for the two algorithms we present.

5.1 The unweighted formulation

Let H be a simple history, and let $\mathcal{Z}_H$ be the set of zones corresponding to the clusters of operations in H. We call a set of zones *compatible* if no two zones in this set conflict with each other. Conflicts between pairs of zones are defined as in the GK algorithm [14], which is explained in Section 3.4. Our goal is to find a maximum-size compatible subset of $\mathcal{Z}_H$, which yields an atomic sub-history with the largest possible number of clusters. Our algorithm first picks all the backward zones, and discards any forward zones that conflict with any of the backward zones. The algorithm then selects a maximum compatible subset of the remaining forward zones. The latter sub-problem can be solved optimally using another greedy algorithm [17], which works as follows. It first sorts the remaining forward zones in increasing order of their right endpoints. It then picks the first unpicked forward zone, removes any forward zones that conflict with the forward zone just picked, and repeats until there are no more unpicked forward zones. The running time is dominated by the time needed to sort the operations and zones, and hence the algorithm can run in $O(n \log n)$ time on an n-operation history. The full algorithm is shown in Algorithm 4. The correctness of this algorithm is stated in Theorem 5.1.

Theorem 5.1 *Given a history H, Algorithm 4 outputs a maximum size subset of clusters that form an atomic sub-history of H.*

5.2 The weighted formulation

Keeping a cluster containing only one operation sounds very different from keeping a cluster containing a hundred operations, and yet in the previous section we do not favor one choice over the other. To account for this disparity, we present a dynamic programming algorithm that identifies a subset of clusters to keep that has the maximum total number of operations. Informally, this approach approximates a solution to the more general problem of finding the smallest subset of operations that must be removed in order to make a history atomic.

Suppose there are m zones in the given history H. Define the *weight* of a zone Z, denoted by $\pi(Z)$, to be the number of operations in that zone. We order these zones in increasing order of their right endpoints and denote them by Z_1 to Z_m. (These endpoints are unique as we assume that start/finish times are unique.) Let $\Pi(i)$, where $1 \leq i \leq m$, denote the maximum total weight of any compatible subset of $\{Z_1, ..., Z_i\}$. We make the following observation about the connection between $\Pi(i)$ and $\Pi(i-1)$. If Z_i is a backward zone, then it is always better to keep Z_i than to discard it, because by the ordering of the zones, Z_i does not conflict with any zones in $\{Z_1, ..., Z_{i-1}\}$. Therefore, $\Pi(i) = \Pi(i-1) + \pi(Z_i)$. If Z_i is a forward zone, then the algorithm has to consider whether it is better to keep Z_i or to discard it. If the algorithm discards Z_i, then $\Pi(i) = \Pi(i-1)$. However, if the algorithm keeps Z_i, then the analysis is slightly more involved.

For any i, let $f(i)$, where $1 \leq f(i) < i$, be the largest index such that $Z_{f(i)}$ precedes (hence does not conflict with) Z_i, or 0 if all zones $\{Z_1, ..., Z_{i-1}\}$ overlap with Z_i. Also let $L(i) = \{\ell : f(i) < \ell < i$ and Z_ℓ is a backward zone that does not conflict with $Z_i\}$. We observe that, if the algorithm keeps forward

zone Z_i, then

$$\Pi(i) = \Pi(f(i)) + \pi(Z_i) + \sum_{\ell \in L(i)} \pi(Z_\ell).$$

Therefore, for a forward zone Z_i, the algorithm picks the max of the above quantity and $\Pi(i-1)$. The complete idea is presented in Algorithm 5. The correctness of the algorithm is stated in Theorem 5.2.

Theorem 5.2 *Given a history H, Algorithm 5 outputs a subset of clusters with maximum total weight that form an atomic sub-history of H.*

In terms of efficiency, extracting the zones takes $O(n \log n)$ time (assuming that the operation endpoints are initially unsorted), sorting the zones takes $O(m \log m)$ time, finding $f(i)$ takes $O(\log m)$ time, and computing $L(i)$ takes $O(m)$ time. Therefore, this algorithm runs in $O(n \log n + m^2)$ time.

6 Related work

Many consistency properties have been proposed before, and we focus on a few well-known ones in this paper. Misra [23] is the first to consider what axioms a read/write register should abide by in order to provide atomic behavior, although the term atomic is not coined there. Lamport [22] first coins the term atomic; the same paper proposes safety and regularity. Herlihy and Wing [18] extend the notion to general data types and define the concept of linearizability. For read/write registers, atomicity and linearizability are equivalent definitions. Lamport [21] proposes sequential consistency.

In the literature, several notions have been proposed to allow an operation or transaction to violate stringent consistency properties, up to a certain limit [1, 20, 26, 28]. The Δ-atomicity property considered in this paper is different from those proposed before, and is motivated by the desire to have a simple number that relates a non-atomic history to a similar atomic one. Yu and Vahdat [32] propose a continuous consistency model that includes a time-based staleness concept similar in spirit to ours, but defined with respect to database replicas rather than individual operations.

To the best of our knowledge, all existing consistency verification algorithms [5, 14, 23] are offline algorithms. Misra [23] presents an elegant algorithm for verifying whether a history is atomic. Given a history, Misra's algorithm defines a "before" relation on the values (of the operations) that appear in the history as follows: (1) a before a if $r(a) < w(a)$ (i.e., a read precedes its dictating write) or there is a $r(a)$ but not $w(a)$, (2) a before b if there exist two operations $op(a), op'(b)$ such that $op(a) < op'(b)$, and (3) a before c if a before b and b before c. A history is atomic iff the "before" relation is irreflexive, anti-symmetric, and transitive. Misra's algorithm can also be viewed as constructing a directed graph called the *value graph*, where each vertex represents a value, and an edge $a \rightarrow b$ exists iff a at some point appears before b (i.e., there exist $op(a), op(b)$ such that $op(a) < op(b)$. Then a history is atomic iff (1) it is simple, and (2) its value graph is a DAG.

Despite the apparent dissimilarity, Misra's algorithm and the GK algorithm have an inherent connection. It is not hard to show that, in Misra's algorithm, for simple histories, if the value graph contains a cycle, then the smallest cycle is of length 2 (i.e., there exist two values a, b such that $a \rightarrow b$ and $b \rightarrow a$). Therefore, it suffices to examine, for each pair of values a, b, whether there are operations that take these two values that appear before each other. And this interpretation translates directly into the GK algorithm's approach of inspecting zone pairs (see Section 3.4). However, similar to the GK algorithm, for verifying 2-atomicity, it is insufficient to examine the value graph in Misra algorithm: it is not hard to construct two histories, one 2-atomic but the other not, that share the same value graph.

Anderson et al. [5] propose off ine verif cation algorithms for safety, regularity, and atomicity, and test the Pahoehoe key-value store [4] using a benchmark similar to YCSB [9]. It is found that consistency violations increase with the contention of accesses to the same key, and that for benign workloads, Pahoehoe provides atomicity most of the time.

The complexity of verif cation has been investigated for several consistency properties [7, 14, 27]. Taylor [27] shows that verifying sequential consistency is NP-Complete. Gibbons and Korach [14] show that, in general, verifying sequential consistency (VSC) and verifying linearizability (VL) are both NP-Complete problems. They also consider several variants of the problem and show that, for most variants, VSC remains NP-Complete yet VL admits eff cient algorithms for some variants. Cantin et al. [7] show that verifying memory coherence, which is equivalent to VSC for one memory location, is still NP-Complete. However, as we discussed in Section 3.6, if write values are unique, then VSC on a single memory location is solvable easily in polynomial time.

In recent years, key-value stores such as Amazon's S3 [2] have become popular storage choices for large-scale Internet applications. According to Brewer's CAP principle [6], among consistency, availability, and partition-tolerance, only two of these three properties can be attained simultaneously. Since partition-tolerance is a must for modern Internet applications, most key-value stores favor availability over consistency. For example, Amazon's S3 [2] and Dynamo [10] only promise eventual consistency [29]. However, more recently, data consistency is becoming a more important consideration, and various key-value stores have proposed ways to provide consistency properties stronger than just eventual consistency [8, 16, 30]. Finally, sometimes data consistency is indispensable, even when an application favors availability. For example, the creation of a bucket in Amazon's S3 [2] is an atomic operation so that no two users can create two buckets of the same name. For such an operation, data consistency is critical and hence in Amazon, availability can be sacrif ced if need be. Consequently, many cloud systems are starting to provide atomic primitives that applications can use to implement strong consistency.

Wada et al. [31] investigate the consistency properties provided by commercial storage systems and made several useful observations. However, the consistency properties they investigated are the client-centric properties such as read-your-write or monotonic-read, which are easy to check. In contrast, the consistency properties we consider in this paper are the data-centric ones and are stronger and harder to verify. Fekete et al. [11] investigate how often integrity violations are produced by varying degrees of isolation in database systems, but those violations are easy to verify.

On the surface, the k-atomicity verif cation problem is somewhat similar to the graph bandwidth problem (problem GT40 of Garey and Johnson [13]). For general graphs, if k is part of the input, then the problem is NP-Complete [12, 24], but if k is f xed, then the problem admits a polynomial-time solution [25]. However, for interval graphs, even if k is part of the input, the problem is polynomial-time solvable [19]: the 2-atomicity algorithm presented in Section 4.2 is similar in spirit to the algorithm therein.

7 Concluding remarks

In this paper, we have addressed several problems related to the verif cation of consistency properties in histories of read/write register operations. In particular, we have considered how to perform consistency verif cation in an online manner. In addition, we have proposed several ways to quantify the severity of violations in case a history is found to contain consistency violations. We have also presented algorithms for computing those quantities. In practice, the online verif cation algorithms enable systems to monitor the consistency provided in real time so that corrective actions can be taken as soon as violations are detected. Quantifying the severity of violations enables customers and service providers to negotiate compensations

proportional to the severity of violations. On the other hand, we have not addressed several problems in their full generality, such as the k-atomicity verif cation problem. We hope to address them in future work.

Acknowledgments

We are thankful to the anonymous referees for their feedback, and to Dr. Ram Swaminathan of HP Labs for his careful proofreading of this paper.

References

[1] A. Aiyer, L. Alvisi, and R. A. Bazzi. One the availability of non-strict quorum systems. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC)*, pages 48–62, September 2005.

[2] Amazon's Simple Storage Service. Available at `http://aws.amazon.com/s3`.

[3] Amazon's SimpleDB. Available at `http://aws.amazon.com/simpledb`.

[4] E. Anderson, X. Li, A. Merchant, M. A. Shah, K. Smathers, J. Tucek, M. Uysal, and J. J. Wylie. Eff cient eventual consistency in Pahoehoe, an erasure-coded key-blob archive. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 181–190, January 2010.

[5] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. Wylie. What consistency does your key-value store actually provide? In *Proceedings of the Sixth Workshop on Hot Topics in System Dependability (HotDep)*, October 2010.

[6] E. Brewer. Towards robust distributed systems, 2000. Available at `http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf`.

[7] J. F. Cantin, M. H. Lipasti, and J. E. Smith. The complexity of verifying memory coherence and consistency. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):663–671, July 2005.

[8] Cassandra. Available at `http://incubator.apache.org/cassandra/`.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, June 2010.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating System Principles (SOSP)*, pages 205–220, October 2007.

[11] A. Fekete, S. N. Goldrei, and J. P. Asejo. Quantifying isolation anomalies. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB)*, pages 467–478, August 2009.

[12] M. R. Garey, R. L. Graham, D. S. Johnson, and D. E. Knuth. Complexity results for bandwidth minimization. *SIAM Journal on Applied Mathematics*, 34(3):477–495, May 1978.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, NY, 1979.

[14] P. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26:1208–1244, August 1997.

[15] W. Golab, X. Li, and M. A. Shah. Analyzing consistency properties for fun and prof t. Technical Report HPL-2011-6, Hewlett-Packard Laboratories, 2011. Available at `http://www.hpl.hp.com/techreports/2011/HPL-2011-6.pdf`.

[16] Google Storage for Developers. Available at `http://code.google.com/apis/storage`.

[17] U. I. Gupta, D. T. Lee, and J. Y.-T. Leung. Eff cient algorithms for interval graphs and circular-arc graphs. *Networks*, 12:459–467, Winter 1982.

[18] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[19] D. J. Kleitman and R. V. Vohra. Computing the bandwidth of interval graphs. *SIAM Journal on Discrete Mathematics*, 3(3):373–375, August 1990.

[20] N. Krishnakumar and A. J. Bernstein. Bounded ignorance in replicated systems. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems (PODS)*, pages 63–74, May 1991.

[21] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[22] L. Lamport. On interprocess communication, Part I: Basic formalism and Part II: Algorithms. *Distributed Computing*, 1(2):77–101, June 1986.

[23] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, January 1986.

[24] C. H. Papadimitriou. The NP-completeness of the bandwidth minimization problem. *Computing*, 16(3):263–270, September 1976.

[25] J. Saxe. Dynamic-programming algorithms for recognizing small-bandwidth graphs in polynomial time. *SIAM Journal on Algebraic and Discrete Methods*, 1(4):363–369, December 1980.

[26] A. Singla, U. Ramachandran, and J. Hodgins. Temporal notions of synchronization and consistency in Beehive. In *Proceedings of the Ninth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 211–220, June 1997.

[27] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent progreams. *Acta Informatica*, 19(1):57–84, April 1983.

[28] F. J. Torres-Rojas, M. Ahamad, and M. Raynal. Timed consistency for shared distributed objects. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 163–172, May 1999.

[29] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009.

[30] Voldemort. Available at `http://project-voldemort.com/`.

[31] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storages: the consumers' perspective. In *Proceedings of the Fifth Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2011.

[32] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, August 2002.

A Correctness proofs for Section 3

Theorem A.1 *The safety verification algorithm in Figure 1 satisfies Specification 3.1 with respect to the safety property.*

Proof sketch: We note that the online safety verif cation algorithm (simply referred to as ALG in this proof) only outputs bad for the f nish events of reads, and that neither the start nor the f nish event of a read causes changes of the data structures used by ALG to determine whether other events are good or bad. In particular, the start/f nish event of a read only affects I (which is used to keep track of which reads should be ignored) and R (which is used to keep track of which reads are ongoing). Neither I nor R plays a role in determining whether other reads are good or bad. Therefore, the last output of ALG on a history H is the same as the last output of ALG on $\tilde{H}$, where $\tilde{H}$ is the history with all the bad events (i.e., those events that cause ALG to output bad) removed. Therefore, it suff ces to show that on a history H whose length $(|H| - 1)$-pref x is good (i.e., ALG outputs good for all the f rst $|H| - 1$ events), ALG outputs good iff H is good (i.e., iff the last event of H is good).

We next argue that ALG produces exactly the same last output as a correct off ine safety verif cation algorithm (which doesn't discard any information), that is, ALG reports a violation iff H's last event is bad. Suppose ALG outputs bad. The cause can only be that the last event is the f nish of a read $r(a)$ and $w[a]$ does not exist. This can be because either $w(a)$ has not started or $w[a]$ is once created but is later discarded. In the f rst case, H is not safe because there is a read that precedes its dictated write. In the second case, H is not safe because of the three operations $w(a)$, $w(b)$, and $r(a)$, appear in sequence in the history (recall that, by the def nition of safety and by ALG, $r(a)$ is not concurrent with any writes).

On the other hand, if ALG outputs good, then we can simulate the running of ALG and generate a total order of the operations that satisf es the safety property. In particular, we generate the total order as follows: (1) when ALG discards some writes, we add these discarded writes to the total order, in arbitrary order, (2) after processing the f nish event of a read, we add the read to the total order. It is not hard to show that a total order generated this way satisf es the safety property. Therefore, ALG satisf es Specif cation 3.1. □

Here we give a brief remark on the eff ciency of this algorithm. Let Λ_r denote the maximum number of concurrent reads at any time and let Λ_w be the maximum number of concurrent writes at any time. Then we observe that, in terms of space eff ciency, $|R| \leq \Lambda_r$ and $|w[*]| \leq \Lambda_w$. The processing time of each event is dominated by set membership testing, which runs in $O(\log \max(\Lambda_r, \Lambda_w))$ time. Therefore, as long as Λ_r and Λ_w are not high, this algorithm is eff cient. Otherwise, $w[*]$ can be implemented eff ciently using hashing.

Theorem A.2 *The online atomicity verification algorithm in Figure 2 satisfies Specification 3.1 with respect to the atomicity property.*

Proof sketch: The proof is similar in structure to the previous proof. To see that a bad event do not alter the algorithm's data structures beyond the processing of the event, note that we undo the updates to the zone variables when the latest read causes the zones to conf ict with each other. Therefore, when given a history with bad events, ALG reports the bad events, but continues as if the bad events never happen.

To see the equivalence of ALG to the GK algorithm, simply note that, in the case where ALG reports a violation because of the absence of $w[a]$, or in the case where ALG reports violation when two zones conf ict, the GK algorithm would have detected a conf ict too. Furthermore, these are the only two cases where the GK algorithm would have detected a conf ict. Therefore, ALG is equivalent to GK for the output

of the last event. Therefore, ALG satisfies Specification 3.1. □

B Proofs for Section 4

B.1 Proofs for Section 4.1

Proof of Fact 4.1: (1) Decreasing the start times of reads can only shrink a forward zone by moving its right endpoint (potentially turning a forward zone into a backward zone), or stretch a backward zone by moving its left endpoint. This can only eliminate conflicts between two zones; it cannot create new conflicts. (2) Decreasing the start times of reads by a sufficiently large amount forces all zones to become backward zones. By the GK algorithm, backward zones do not conflict with each other. (3) Follows from (1) and the definition of Δ-atomicity. □

Proof of Lemma 4.2: First, suppose that H is Δ-atomic. Then by definition, H_Δ is atomic. Therefore, there exists a valid total order on the operations in H_Δ such that a read obtains the value of the latest write that precedes it in the total order. Then there exists some assignment of timestamps to operations in H_Δ, such that the timestamp for each operation is a time value between the operation's start and finish, and such that the ordering of operations by increasing timestamp is identical to the above total order. It remains to relate the timestamps just defined for H_Δ to the timestamps for H referred to by Lemma 4.2. If a read in H_Δ has a timestamp less than Δ time units after its start, then the timestamp of this read in H is its start time in H. Otherwise, the read's timestamp in H is the same as its timestamp in H_Δ. The timestamp of a write in H is also the same as its timestamp in H_Δ. It is easy to verify that the timestamps proposed for operations in H have the desired properties.

Conversely, suppose that there is an assignment of timestamps for operations in H as described in Lemma 4.2. It follows easily that if we shift the left endpoints of all read operations by Δ then the resulting history is atomic. In other words, H_Δ is atomic, and hence H is Δ-atomic. □

Proof of Theorem 4.3: By Fact 4.1, all pairs of zones in $H_{\Delta_{\text{opt}}}$ are compatible, and so $H_{\Delta_{\text{opt}}}$ is atomic. Furthermore, for any $\Delta < \Delta_{\text{opt}}$, there is some pair of zones in H_Δ that conflict, making H_Δ not atomic. Thus, Δ_{opt} is the smallest possible. □

B.2 Proofs for Section 4.2

The intuitive understanding of the algorithm (simply referred to as ALG for the sake of conciseness) is that ALG places the writes from right to left. When there are no other constraints, it picks a write $w(a)$ that finishes last among the currently unpicked writes. Once ALG lays out $w(a)$, it computes the set of reads R that should follow the write in this total order. Note that this is the "best" place to put those reads in R, given that $w(a)$ is the next write to be laid out. However, if R is not empty, then it imposes additional constraints on what the next write should be laid out, because of the 2-atomicity requirement. In particular, if $|R| > 1$, then it means that, in order to keep the 2-atomicity requirement, we have to lay out multiple writes at the next place, an obviously impossible task. Hence, ALG gives up. If $|R| = 1$, then ALG is obliged to lay out the dictating write for the lone read in R. There is no other option. If $|R| = 0$, then ALG is free to pick any unpicked write and it picks the one with the largest finish time. The intuition of picking such a write is that, compared to other choices, such a write forces fewer number of reads that has to follow it in the total

order, potentially reducing the rw-distance (i.e., the number of writes between a write and its dictated read) for other values. To establish the correctness of the algorithm in Figure 3, we first establish the following key lemma, which holds for arbitrary k.

Lemma B.1 *If a nice history is k-atomic, then there exists a k-atomic total order where the last write in the total order is the write that finishes last among all the writes in the history.*

Proof: Let $w(a)$ be the write that finishes last among all the writes in the history. Let T be a k-atomic total order for the history. We first observe that $w(a)$ is one of the last k writes in the total order, otherwise the rw-distance of value a is $> k$, violating the k-atomicity definition. If $w(a)$ is the last write in T, then we are done. Otherwise, let W be the sequence of writes that follow $w(a)$ in the total order. We claim that we can swap W and $w(a)$, and repack the reads (i.e., adjust the placement of the reads, see Section 4.2 for the definition of a *packed* total order) accordingly, and the new total order T' will remain k-atomic.

To see this, note that repacking only affects those reads that are placed to the right of $w(a)$ in T. Among these reads, the repacking may increase the rw-distance for those values in W, but will only decreasing or maintain the rw-distance for other values, because $w(a)$ finishes last among all the writes. For the former set of values, since their writes all appear in the last k writes, the repacking of their reads, no matter where they end up, will not violate k-atomicity. □

We next prove the correctness of the algorithm.

Theorem B.2 *The 2-atomicity verification algorithm in Figure 3 generates a 2-atomic total order iff the given nice history is 2-atomic.*

Proof: For the sake of conciseness, we call our algorithm ALG. If ALG produces a total order, then the total order is 2-atomic, because ALG observes all constraints. The non-trivial part is to show that if the history is 2-atomic, then ALG will indeed generate a total order.

We prove this by contradiction. Suppose the given history is 2-atomic but ALG reports a violation somewhere during its execution. Since the history is 2-atomic, there exists a 2-atomic total order. Suppose ALG makes the first crucial mistake when it lays out $w(a)$, namely, there are no 2-atomic total orders that end with $w(a)W$, where W is the sequence of writes previously laid out by ALG, but there are 2-atomic total orders that end with W. Such a $w(a)$ exists because if ALG has been correct so far and reports a violation next, then it can only be because $|R| > 1$ or there are unpicked writes that succeed a picked write. In either case, there is no way to extend the partial layout to a 2-atomic total order, contradicting the assumption that there are 2-atomic total orders that end with W.

If $W = \emptyset$ (i.e., ALG is wrong from the beginning), then by Lemma B.1, there is a 2-atomic total order that lays out $w(a)$ first. A contradiction. If $W \neq \emptyset$, then let $w(b)$ be the first write in W. Consider the R set computed after laying out $w(b)$. Since ALG continues after laying out $w(b)$, we know that $|R| = \emptyset$ and there are no unpicked writes that succeed $w(b)$. Therefore, there are no constraints on what to lay out next. In other words, ALG starts afresh for all unpicked operations. By Lemma B.1, there exists a 2-atomic total order that lays out $w(a)$ next. A contradiction again. □

C Proofs for Section 5

Proof: (of Theorem 5.1) Consider the conflict graph for the history where a vertex represents a zone and an edge between two vertices means the two zones conflict with each other. The problem of identifying a

maximum-size compatible subset zones is equivalent to f nding a maximum independent set in this graph. Let F be the forward vertices (i.e., vertices for forward zones), B be the backward vertices (i.e., vertices for backward zones), F_1 be those vertices in F that have neighbors in B, F_2 be those vertices in F that do not have neighbors in B, B_1 be those vertices in B that have neighbors in F (or more precisely, in F_1), and B_2 be those vertices in B that do not have neighbors in F.

We note that vertices in B are independent because backward zones do not conf ict with each other. Also note that for any vertex b in B_1, the neighbors of b in F_1 form a clique, because they are all forward zones that contain zone b and hence overlap with each other. We claim that the algorithm presented in Figure 4, called ALG, keeps the maximum number of zones. Clearly, this claim is true for B_2 because ALG keeps all of B_2. For $B_1 \cup F_1$, no algorithm can keep more than $|B_1|$ zones, because of the clique observation above and the pigeon-hole principle. Therefore, by keeping all of B_1, ALG also keeps the maximum number of zones for $B_1 \cup F_1$. Finally, for F_2, since ALG does not pick any zones from F_1, ALG is free to pick as many zones from F_2 as possible provided they do not conf ict with each other. ALG does exactly this by using an optimal algorithm to pick the maximum-size compatible subset of F_2. Therefore, ALG is optimal for F_2 as well, and in summary, optimal overall. □

D Dealing with failures and missing information

The history constructed by a monitor from notif cations issued by clients may contain various anomalies that either preclude the use of known consistency verif cation algorithms, or else confound the results of such algorithms. In this section we identify these anomalies and discuss workarounds.

Suppose that at some time t, the monitor materializes a history H based on the execution fragments observed so far, which may be missing certain events. Our goal is to show how the monitor can convert H into a simple history H' (see Section 4.1 for the def nition of simple histories) that represents well the events occurring in H. A simple history has a special structure that meets the preconditions of most consistency verif cation algorithms. By modifying H to form H', we inevitably risk introducing of false negatives or false positives in the verif cation process, which examines H' in order to deduce properties of H. Our strategy will be to construct H' in such a way that we introduce false positives only. That is, we ensure that if H' violates a consistency property (i.e., the verif cation algorithm produces a negative outcome) then H does as well. This is reasonable because in general H only represents partial information about the read and write operations applied during an execution of the system, and so we cannot deduce from H alone that this collection of operations satisf es some consistency property. We can only identify features in H proving that a consistency property is violated even if H is an incomplete view of the big picture.

We now describe in detail the construction of H' from H. A common occurrence is that H contains start events without matching f nish events, because the f nish notif cation had not yet been issued by a client by time t. Similarly, H may contain a f nish event without an start, because the start notif cation reached the monitor before the monitor began collecting data. We can obtain a simple history H from H' as explained below.

If a read start in H is missing a matching f nish, we simply discard it because it provides no useful information. If a write start is missing a f nish, we discard it if no read in H returns the value written, otherwise we complete it with a f nish with timestamp t. In other words, in the f rst case we guess that the write did not take effect, and in the second case we guess that it did. If a read or a write has a f nish but not a start, we would like to reconstruct the start event if possible. Although there is nothing in our model that makes this possible, we can easily modify the system to make this possible by attaching redundant data to a

f nish notif cation, namely the timestamp of the corresponding start and its argument, if any. Finally, some start/f nish pair may be missing from H entirely. If the missing operation is a read, then there is no way to detect this at the monitor. However, if the dictating write for a read is missing, then we wish to avoid a false negative. To that end, we remove any read operation that lacks a dictating write. It is easy to verify that the construction of H' from H described above yields a simple history.

Algorithm 2: Online atomicity verif cation

Input: sequence of events $\langle e_1, e_2, \dots \rangle$
Output: sequence of *good*/*bad* values $\langle \gamma_1, \gamma_2, \dots \rangle$
Init: $R = \emptyset$; no $w[\cdot]$, $Z[\cdot]$, or $\alpha[\cdot]$ variables

upon event e_i **do**
  $\gamma_i :=$ *good*;
  if $e_i = |w(a)$ **then**
    create $w[a]$, $Z[a], \alpha[a]$;
    $w[a].(s, f) := (w(a).s, \infty)$;
    $Z[a].(\underline{f}, \bar{s}) := (\infty, w(a).s)$;
    $\alpha[a] :=$ nil
  else if $e_i = w(a)|$ **then**
    $w[a].f := w(a).f$;
    $Z[a].\underline{f} := \min(Z[a].\underline{f}, w(a).f)$;
    foreach $(b : w[b] < w[a] \wedge \alpha[b] =$ nil$)$ **do**
      $\alpha[b] := R$;
      if $\alpha[b] = \emptyset$ **then** discard $w[b], Z[b], \alpha[b]$ **end**
    end
  else if $e_i = |r(?)$ **then**
    add $r(?)$ to R
  else if $e_i = r(a)|$ **then**
    if $\nexists\, w[a]$ **then**
      $\gamma_i :=$ *bad*
    else
      $Z[a].\bar{s} := \max(Z[a].\bar{s}, r(a).s)$;
      $Z[a].\underline{f} := \min(Z[a].\underline{f}, r(a).f)$;
      if $(\exists b : Z[b].\underline{f} \neq \infty \wedge Z[a] \not\sim Z[b])$ **then**
        $\gamma_i :=$ *bad* ;
        undo updates to $Z[a]$ for e_i
      end
    end;
    remove $r(a)$ from R ;
    foreach $(b : r(a) \in \alpha[b])$ **do**
      remove $r(a)$ from $\alpha[b]$;
      if $\alpha[b] = \emptyset$ **then** discard $w[b], Z[b], \alpha[b]$ **end**
    end
  end;
  output γ_i
end

Algorithm 3: Off ine 2-atomicity verif cation

Input: condensed nice history H
Output: whether or not H is 2-atomic
Init: $R = S = \emptyset$; none of the operations are picked

while $\exists$ unpicked writes **do**
  if $|R| > 1$ **then output** *bad*
  else
    if $|R| = 1$ **then**
      $w(a) :=$ dictating write for the lone $r(a) \in R$
    else
      $w(a) :=$ unpicked write with largest f nish time
    end
    if $\exists$ unpicked $w(b) : w(a) < w(b)$ **then**
      output *bad*
    end
    $S :=$ {unpicked reads $r : w(a) < r$};
    // $r(a)$ may or may not be in S
    pick $w(a)$ and S, prepend them to the total order;
    // order of reads in S is unimportant
    $R := S \setminus \{r(a)\}$
  end
end;
output *good*

Algorithm 4: Max subset of compatible clusters

Input: simple history H
Output: maximum size subset of compatible clusters in H

$\mathcal{C} :=$ set of clusters for operations in H;
$\mathcal{Z} :=$ set of zones corresponding to $\mathcal{C}$;
$R :=$ subset of clusters in $\mathcal{C}$ with backward zones;
foreach $\overrightarrow{Z(a)} \in \mathcal{Z} : (\exists\ \overleftarrow{Z(b)} \in \mathcal{Z} : Z(a) \not\sim Z(b))$ **do**
  remove $C(a)$ from $\mathcal{C}$
end;
$R' :=$ max compatible subset of $\mathcal{C}$ with forward zones;
// a standard greedy algorithm can compute R'
$R := R \cup R'$;
output R

Algorithm 5: Compatible clusters with max total weight

Input: simple history H

Output: subset of compatible clusters in H with maximum total number of operations

$\Pi[0] := 0$;
$R[0] := \emptyset$;
$Z_{1:m} :=$ zones in H sorted by increasing right endpoints;
$C_{1:m} :=$ clusters corresponding to $Z_{1:m}$;
for $i := 1$ **to** m **do**
  if $\overrightarrow{Z_i}$ **then**
    if $(\exists j : 1 \leq j < i \wedge Z_j < Z_i)$ **then**
      $f :=$ max of such j
    else
      $f := 0$
    end;
    $L := \{\ell : f < \ell < i \wedge \overleftarrow{Z_\ell} \wedge Z_\ell \sim Z_i\}$;
    $\Pi[i] := \max\{\Pi[i-1], \Pi[f] + \pi(Z_i) + \sum_{\ell \in L} \pi(Z_\ell)\}$;
    if $\Pi[i] = \Pi[i-1]$ **then**
      $R[i] := R[i-1]$
    else
      $R[i] := R[f] \cup \{C_i\} \cup \{C_l : l \in L\}$
    end
  else
    $\Pi[i] := \Pi[i-1] + \pi(Z_i)$;
    $R[i] := R[i-1] \cup \{C_i\}$
  end
end;
output R