SIAM J. COMPUT.
Vol. 47, No. 2, pp. 420–455

COMPUTING k-ATOMICITY IN POLYNOMIAL TIME*

WOJCIECH GOLAB†, XIAOZHOU (STEVE) LI‡, ALEJANDRO LÓPEZ-ORTIZ§, AND NAOMI NISHIMURA§

In memory of Alex López-Ortiz

Abstract. The k-atomicity property can be used to describe the consistency of data operations in large distributed storage systems. The weak consistency guarantees offered by such systems are seen as a necessary compromise in view of Brewer's CAP principle. The k-atomicity property requires that every read operation obtains a value that is at most k updates (writes) old and becomes a useful way to quantify weak consistency if k is treated as a variable that can be computed from a history of operations. Specifically, the value of k quantifies how far the history deviates from the atomicity (linearizability) property for read/write registers. We address the problem of computing k indirectly by solving the k-atomicity verification problem (k-AV): given a history of read/write operations and a positive integer k, decide whether the history is k-atomic. Gibbons and Korach showed that in general this problem is NP-complete when $k = 1$ and hence not solvable in polynomial time unless P = NP. In this paper we present two algorithms that solve the k-AV problem for any $k \geq 3$ in special cases. Similarly to known solutions for $k = 1$ and $k = 2$, both algorithms assume that all the values written to a given object are distinct. The first algorithm places an additional restriction on the structure of the input history and solves k-AV in $O(n^2 + n \cdot k \log k)$ time, where n is the number of operations in the history. The second algorithm does not place any additional restrictions on the input but is efficient only when k is small and when concurrency among write operations is limited. Its time complexity is $O(n^2)$ if both k and our particular measure of write concurrency are bounded by constants.

Key words. consistency, atomicity, verification, distributed storage

AMS subject classifications. 68M14, 68Q25, 68N30

DOI. 10.1137/16M1056389

1. Introduction.

Distributed storage and data management systems empower a broad range of data-intensive services today including social networking, web search, e-mail, calendars, and online auctions. In an effort to cope with web-scale workloads in the face of immense competitive pressures, the designs of such services have shifted away from conventional relational databases and toward simpler but more scalable solutions. As a result, many practical systems offer simple key-value operations and basically available, soft state, eventual consistency (BASE) properties either instead of or side by side with conventional transactions and more powerful atomicity, consistency, isolation, durability (ACID) properties.

Eventually consistent key-value stores are an important breed of distributed storage systems that provide BASE properties [8, 35, 37]. Their distinguishing charac-

*Received by the editors January 12, 2016; accepted for publication (in revised form) February 7, 2018; published electronically April 10, 2018. A preliminary version of this manuscript appears in the proceedings of PODC 2015 [18].

http://www.siam.org/journals/sicomp/47-2/M105638.html

Funding: The first author was supported in part by the Google Faculty Research Awards program, the Amazon Web Services (AWS) Cloud Credits for Research program, and the Microsoft Azure for Research program. The first, third, and fourth authors were supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

†Electrical and Computer Engineering, University of Waterloo, Waterloo, ON N2L 3G1, Canada (wgolab@uwaterloo.ca).

‡Google Inc., Mountain View, CA 94043 (xzli@google.com).

§School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada (nishi@uwaterloo.ca).

teristic is the ability to remain available in the face of network partitions, in which disjoint subsets of hosts become isolated from one another and are unable to communicate. To ensure that storage operations can proceed despite such partitions, specialized replication techniques such as *sloppy quorums* [8, 22, 33] are used. As a side effect, eventually consistent systems can only provide weak consistency guarantees—a necessary compromise according to Brewer's CAP principle [5], which states that a distributed storage system cannot simultaneously provide strong consistency (C), high availability (A), and partition-tolerance (P). This observation has been formalized by Gilbert and Lynch as the impossibility of implementing an atomic read/write register in an asynchronous message passing system with unreliable communication channels [15]. A read/write register in this context corresponds to a key-value pair as follows: the key is the register's name and is immutable, whereas the value is the object's state and can be accessed using read and write operations.

As defined by Terry et al. [35] and Vogels [37], eventual consistency means that if an object is not updated and no failures occur for a sufficiently long period of time, then eventually all read operations on the object return the last updated value of that object. Although this definition captures nicely the essence of eventual consistency, it also leaves open important questions regarding the behavior of key-value stores in practice: How long is "eventual," and how consistent are reads when objects are updated continuously or failures never subside? In an attempt to answer the latter question, various notions of *data staleness* have been defined and analyzed. In this context, a value written to a key is considered stale once it is overwritten by another value, and so the staleness of a particular read is a measure of the "distance" between two write operations: the write that assigned the value read (called the *dictating write* of the read), and the write that assigned the latest value to the object under consideration.

In this paper, we focus on a formal notion of *version-based staleness*, which is defined by counting write operations. Specifically, we consider the *k-atomicity* property of Aiyer, Alvisi, and Bazzi [1], which is a generalization of Lamport's atomicity property for read/write registers [23]. Both properties are defined over an *execution history*, which is a collection of read/write operations tagged with distinct start and finish times. An operation in such a history "happens before" another operation if the former finishes before the latter starts, and a history is called k-atomic if there exists a total order over the operations that extends the "happens before" partial order, and in which every read returns a value written by one of the last k writes preceding the read in the total order.

The k-atomicity verification problem (k-AV) is to decide for a given execution history and integer $k \geq 1$ whether the history is k-atomic. An efficient solution to this problem makes it possible to compute the smallest k for which a given history is k-atomic, which quantifies the maximum degree of staleness observed by clients accessing an eventually consistent storage system. Thus, solving k-AV makes it possible to analyze the actual consistency provided by such systems in arbitrary workloads, including ones where writes occur continuously and concurrently with each other.

Our main contribution in this paper is a pair of algorithms that solve the k-AV problem for arbitrary $k \geq 3$. Both algorithms assume that all the values written to a given object in a history are distinct, which establishes a precise "reads from" mapping between read and write operations. The first algorithm, called GPO ("greedy plus obligations"), places an additional restriction on the structure of the input history and solves k-AV for arbitrary k in $O(n^2 + n \cdot k \log k)$ time, where n is the number of operations in the history. The second algorithm, called CGS ("configuration graph

search"), does not place any additional restrictions on the input but is efficient only when k is small and when the concurrency among write operations is restricted. Its time complexity is $O(n^2)$ if both k and our specific measure of write concurrency are bounded from above by constants. As explained in section 7, we expect these algorithms to perform efficiently in practice.

2. Related work. The fundamental problem of defining consistency properties for shared objects was first addressed by Misra [27] and Lamport [23]. Lamport defines a number of key concepts in this area, including the *happens before* and $\longrightarrow$ (i.e., precedes) relations, and the atomicity property for registers [23]. The "happens before" relation is an irreflexive partial order over instantaneous events that captures the following notion of causality: given two events e_1, e_2 in Einstein's four-dimensional space-time, we say that e_1 happens before e_2 if it is possible for a pulse of light emitted at e_1 to arrive at e_2, and hence for e_1 to physically cause e_2. Operations on objects are modeled as collections of events, and given two operations op_1, op_2 we say that $op_1 \longrightarrow op_2$ if every event of op_1 happens before every event of op_2. In the presence of a global clock, events can be totally ordered according to time, in which case $op_1 \longrightarrow op_2$ means that op_1 finishes before op_2 starts. Lamport's atomic read/write register satisfies the property that any collection of operations applied to the object can be arranged (conceptually) in some total order that extends $\longrightarrow$ and in which every read returns the value assigned by the latest write [23]. Misra's axioms [27] provide an alternative definition of such a register and explicitly address the case of multiple concurrent writers, whereas Lamport's definitions are given for the single-writer case but generalize easily to multiple writers. Herlihy and Wing's *linearizability* property [20] is a generalization of Lamport's atomicity property to arbitrary typed shared objects, and to scenarios that include pending (i.e., incomplete) operations that may appear to take effect before they finish. Herlihy and Wing's *happens before* relation over pairs of operations is a special case of Lamport's $\longrightarrow$ relation when events are totally ordered.

A variety of relaxed consistency properties have been defined in an attempt to describe the behavior of weakly consistent distributed storage systems. In general these properties capture the staleness of the values returned by read operations, which we can think of more precisely as the "distance" between operations, as explained in section 1. Two notions of staleness have been defined in the literature: *time-based* and *version-based*. A time-based staleness of t time units (measured using a wall clock) means that a read returned a value that was considered fresh at most t time units earlier. Torres-Rojas, Ahamad, and Raynal formalize this notion by defining *timed consistency* properties in distributed message passing systems as generalizations of sequential consistency and causal consistency [36]. Similarly, Golab, Li, and Shah generalize the atomicity property for read/write registers by defining Δ-atomicity [17] and Γ-atomicity [19]. On the other hand, a version-based staleness of k versions means that a read returned the value assigned by one of the last k writes that precede the read in some total ordering of the write operations. Aiyer, Alvisi, and Bazzi formalize this notion as the *k-atomicity* property, which is also inspired by Lamport's atomic register [1]. A collection of read/write operations is k-atomic if the operations can be arranged in some total order that extends $\longrightarrow$ and in which every read returns the value assigned by one of the k latest writes.

Several storage systems and modeling frameworks incorporate relaxed consistency properties. The TACT (tunable availability/consistency tradeoffs) framework of Yu and Vahdat defines version-based and time-based staleness informally as *order error*

and *staleness* [39]. Lee and Welch [24] describe a probabilistic consistency framework in the context of *randomized registers* implemented using probabilistic quorums, introduced earlier by Malkhi, Reiter, and Wright [25]. The analysis of randomized registers refers implicitly to version-based staleness by deriving bounds on the probability that a stale replica exists for a data item that has been overwritten a certain number of times. The probabilistically bounded staleness (PBS) framework of Bailis et al. uses similar principles to derive bounds on time-based and version-based staleness [3]. The authors formalize both notions of staleness in the simplified model where write operations do not overlap in time with other read or write operations. Ardekani and Terry [2] as well as Terry et al. [34] propose storage systems that can be configured using service level agreements (SLAs) to provide bounds on time-based staleness.

Techniques for calculating consistency metrics, such as the ones presented in this paper, complement the body of work on tunable storage systems and consistency modeling in several ways. First, they enable an empirical understanding of consistency in practical storage systems, ranging from measuring the convergence time of the replication protocol [4, 30, 38] to analyzing the complex effect of the workload and system configuration on the consistency actually observed by client applications [19, 26]. Second, precise analysis of consistency enables meaningful verification. For example, a customer who negotiates an SLA with a service provider could use a k-atomicity decision procedure to determine whether the SLA was indeed honored. This is possible even if the SLA refers to some alternative notion of staleness, for example one that refers to the internal implementation details of the system (e.g., where data versions are determined by timestamp ordering), as long as the provider's definition of staleness bears a precise relationship to k-atomicity. Similarly, consistency analysis could be used to validate mathematical models of consistency in environments that do not conform exactly to the simplifying assumptions on which the models are based. For example, PBS [3] assumes that network latencies have well-defined and independent distributions and was validated in an environment where the latencies were controlled artificially to meet the assumption; its accuracy in a real-world environment remains to be verified.

Decision problems pertaining to the consistency properties discussed herein are formalized using concepts borrowed from the study of shared memory multiprocessors. The input to the decision problem is an *execution history*—a sequence of invocation and response events corresponding to operations applied to a collection of shared objects. The events in a history define the invocations and responses of operations on objects. Gibbons and Korach show that deciding whether such a history is atomic is NP-complete in the general case [14]. However, in the special case when each write operation on a given object assigns a distinct value, polynomial-time decision algorithms for atomicity have been developed. Under this assumption, Misra's axioms can be used to characterize atomicity as the absence of cycles in a *conflict graph* whose vertices represent values and whose edges represent precedence constraints imposed by the ordering of reads and writes in the history [27]. For a history with n operations the time complexity of this algorithm is $O(n^2)$. Gibbons and Korach instead followed the approach of clustering operations according to the value read or written and characterized atomicity as the absence of conflicts between pairs of such clusters [14]. Their algorithm has time complexity $O(n \log n)$.

The decision problem for the k-atomicity property is called the k-atomicity verification (k-AV) problem. Given a history and an integer $k \geq 1$, the objective is to decide whether the history is k-atomic. The *locality* property of k-atomicity states that the input history is k-atomic if and only if for each object accessed, the subhis-

tory of operations on that specific object is k-atomic [16, 20]. Accordingly, solutions to k-AV assume without loss of generality that all operations are applied to the same object. A solution to k-AV can be used to compute the *k-value* of a history, defined as the smallest k for which the history is k-atomic. The k-value equals 1 if and only if the history is atomic in Lamport's sense [23], whereas larger k-values indicate deviation from atomicity and hence from linearizability. In general the higher the k-value, the greater the deviation.

The k-AV problem was first considered by Golab, Li, and Shah [17]. Like the problem of deciding atomicity, k-AV is NP-complete in the general case, assuming that k is part of the input. Golab, Li, and Shah present the first polynomial-time solutions to 2-AV under the assumption that each write operation on a given object assigns a distinct value, which is the same assumption that makes 1-AV tractable [16]. One algorithm, called LBT, runs in $O(n^2)$ time for a history with n operations. It leverages ideas inspired by Misra [27], as well as the technique of limited backtracking [10]. The other algorithm, called Forward Zones First (FZF), runs in $O(n \log n)$ time, and borrows the clustering idea of Gibbons and Korach [14]. Roughly speaking, both algorithms attempt to establish k-atomicity by arranging the write operations in a history in a total order constrained by the "happens before" relation and by the positions of read operations relative to the writes. The key observation with respect to the latter constraint is that placing a write in some position of the total order restricts which writes may be placed in neighboring positions. In general, this restriction either dictates that the next $k-1$ positions must include some particular subset of up to $k-1$ writes or negates k-atomicity by requiring that the same positions must include k or more particular writes. In the first case, if the subset of up to $k-1$ writes is nonempty, then it has $(k-1)!$ permutations, meaning that the next write is determined uniquely when $k = 2$ but not when $k > 2$. LBT and FZF exploit this uniqueness to achieve polynomial running time for $k = 2$ and do not generalize easily to higher k.

The k-AV problem has a simple solution when k is greater than or equal to the number of write operations (e.g., $k \geq n$) since in that case the writes can be ordered arbitrarily subject to the "happens before" constraint, in contrast to the case of smaller k. More precisely, it suffices to verify that each read returns the value assigned by some write whose start time is less than the finish time of the read. This can be achieved in $O(n \log n)$ time as follows: (1) use a balanced binary tree to group the operations by value, representing each group as a linked list; (2) verify that each group contains at least one write, and compute for each group the minimum start time of all the writes in the group; and (3) verify for each group that no read finishes before the minimum write start time computed earlier. Part (1) can be executed in $O(n \log n)$ steps, while $O(n)$ steps in total suffice for parts (2) and (3).

The k-AV problem is somewhat similar to the graph bandwidth problem (GBW). Given a graph G and a positive integer k, GBW is to decide whether it is possible to arrange the vertices of G at distinct positions on a line such that any two vertices that are adjacent in G are separated by at most $k-1$ other vertices on the line. GBW is NP-complete when k is part of the input [29] and is solvable in polynomial time when k is fixed [13, 31]. The algorithm of Saxe uses dynamic programming and runs in $O(n^{k+1})$ time, where n is the number of vertices [31]. Kleitman and Vohra [21] present a GBW algorithm that runs in $O(n \log n)$ time but is correct only for interval graphs.

Our graph-theoretic characterization of k-AV, which is presented later on in section 3 (see Lemma 3.8), has a similar flavor to GBW in the sense that it refers to a

linear arrangement of vertices that is constrained by edges. The main difference is that we consider constraints with respect to two different but related sets of edges, with one set imposing "hard" constraints similar to edges in a topological ordering, and the other set imposing "soft" constraints similar to edges in GBW. Furthermore, our edges are directed and the union of the two edge sets has a structure similar to an interval graph, whereas GBW is defined over arbitrary undirected graphs. Regarding time complexity, we are not aware of a polynomial-time Turing reduction from GBW to k-AV or vice versa. Such a reduction from k-AV to GBW would imply that k-AV is solvable in polynomial time for fixed k.

In the area of parameterized complexity, introduced by Downey and Fellows [9], a problem is associated with one or more *parameters*; even for a problem without a known polynomial-time algorithm, it may be possible to confine the potentially exponential (or worse) blow-up to a function of the parameter(s) only. As in classical complexity, parameterized complexity also has the notions of a hierarchy of complexity classes, reductions, and hardness. More on parameterized complexity can be found in subsequent monographs [7, 11, 28].

Portions of this paper devoted to the CGS algorithm, which has exponential time complexity in the worst case, are related to fixed-parameter tractability. More formally, for a parameterized problem P with inputs of the form (x, k), where $|x| = n$ and k is a positive integer, P is *fixed-parameter tractable* (or in FPT) if it can be decided in $f(k)n^c$ time, where f is an arbitrary function and c is a constant independent of both n and k. A fixed-parameter algorithm is of particular interest when k is known to be a small constant independent of n, hence yielding a polynomial algorithm for such instances.

3. Preliminaries. Following the model of Herlihy and Wing [20], we define an *execution history* (or *history* for short) as a sequence of *events* where each event is either the invocation or the response of an operation on a read/write register. Each operation in the history has both an invocation and a response event, and all operations are applied to the same object (see discussion of locality in section 2). Each event is tagged with a unique time, and events appear in the history in increasing order of their times. The exact values of these times are not important—only their relative order—and so we assume without loss of generality that the ith event (counting from $i = 1$) occurs at time i. Invocation events also record the operation type (i.e., read versus write) and the argument (for writes only), whereas response events record the return value (for reads only). The *value* of an operation op is its argument if op is a write, and its response if op is a read. Given a read r and a write w we call w a *dictating write* of r if r and w share the same value. In that case we call r a *dictated read* of w. A write may have any number of dictated reads, but we require (see Assumption 3.2) that each read has a unique dictating write. In the remainder of the paper, the total number of operations and the number of writes in the input history are denoted by n and n_w, respectively.

Given an operation op, we denote by $s(op)$ and $f(op)$ the start time and finish time of op, respectively. Given a pair of operations op_1, op_2 in a history H, we denote Herlihy and Wing's *happens before* relation, which is a partial order, by $<_H$ (i.e., $op_1 <_H op_2$ if and only if $f(op_1) < s(op_2)$). We say that op_1 and op_2 are *concurrent* if neither $op_1 <_H op_2$ nor $op_2 <_H op_1$, and we use the notation $op_1 \not<_H op_2$ to denote that $op_1 <_H op_2$ is false. A partial order $<_S$ over the operations in a history H *is compatible with* the partial order $<_H$, denoted $<_S \sim <_H$, if the following property holds for every pair of operations op_1, op_2 in H: if $op_1 <_H op_2$ and $<_S$ orders op_1

relative to op_2, then $op_1 <_S op_2$. (Note that $<_S$ is not required to order op_1 relative to op_2 even if $<_H$ does so.) Given a total order $<_T$ over the operations in a history H, $<_T \sim <_H$ holds if every pair of operations op_1, op_2 in H satisfies $op_1 <_H op_2 \Rightarrow op_1 <_T op_2$. We call $<_T$ a *k-atomic total order for H* (or *k-atomic ordering of the operations in H*) if $<_T \sim <_H$ and every read operation returns the value assigned by one of the last k writes that precede it in $<_T$. A *k-atomic value order T* for a history H is a permutation of the values written in H that is consistent with some k-atomic total order $<_T$ for H (i.e., if $w(v) <_T w(v')$, then v precedes v' in T). We will show later on in Lemma 3.8 how to determine whether a given sequence of values T is a k-atomic value order for a history H.

DEFINITION 3.1. *A history H is k-atomic if and only if there exists a k-atomic total order $<_T$ for H.*

In the remainder of the paper, we make the following assumption to circumvent the NP-completeness of k-AV.

ASSUMPTION 3.2. *For any history H, every read operation in H has exactly one dictating write.*

Without Assumption 3.2 the k-AV problem is NP-complete, assuming that k is part of the input, because 1-AV is NP-complete [14]. Assumption 3.2 can be enforced in practical storage systems by embedding a unique identifier, for example based on the current time and client ID, in each value.

Under Assumption 3.2 we adopt the following notational convention: for any history H and any value v read or written in H, $w(v)$ denotes the unique dictating write of v in H. Similarly we use $r(v)$ to denote a read of v in H. If more than one read of v exists in H, then the particular operation corresponding to $r(v)$ is specified in the context.

We make two additional assumptions to simplify our algorithms and their analysis. First, we assume that a value cannot be read before it is written.

ASSUMPTION 3.3. *For any history H, if r is a read in H and w is its dictating write, then $r \not<_H w$.*

Second, we assume that a read never finishes before its dictating write.

ASSUMPTION 3.4. *For any history H, if r is a read in H and w is its dictating write, then $f(w) < f(r)$.*

Intuitively, Assumption 3.4 ensures that if r happens before some other read r', then w also happens before r', which simplifies the characterization of k-atomicity presented later on in this section (see Definitions 3.6–3.7 and Lemma 3.8).

Any history that violates Assumption 3.3 automatically fails to satisfy k-atomicity for any $k \geq 1$. Assumption 3.3 can be tested in $O(n \log n)$ steps by clustering operations by their value and checking for each value that no read precedes its dictating write. On the other hand, Assumption 3.4 implies no loss of generality because any history H that satisfies Assumptions 3.2 and 3.3 can be transformed efficiently into a history H' that in addition satisfies Assumption 3.4 in a manner that preserves k-atomicity. This is done using a normalization procedure presented in Algorithm 1. Theorem 3.5 asserts the correctness of this procedure.

THEOREM 3.5. *If the input history H of the normalization procedure (Algorithm 1) satisfies the preconditions (Assumptions 3.2–3.3), then the output history H' satisfies the postconditions (Assumptions 3.2–3.4, and H is k-atomic if and only if H' is k-atomic).*

Input: history H that satisfies Assumptions 3.2–3.3.
Output: history H' that satisfies Assumptions 3.2–3.4, and is k-atomic if and only if H is.

```
1 H' := H
2 for every value v in H do
3 |   r(v) := read of v with earliest finish time in H
4 |   if f(r(v)) < f(w(v)) then
5 |   |   shift the response event for w(v)'s counterpart in H' to the position
  |   |     immediately preceding the response event of r(v)'s counterpart in H'
6 |   end
7 end
```

Algorithm 1: Normalization procedure for histories.

Proof. Suppose that H satisfies Assumptions 3.2–3.3. It suffices to show that line 5 either preserves or introduces the desired properties as H is transformed into H'. To begin with we must show that H' is a history. Initially $H' = H$ by line 1 where H is the input history. At line 5 a write operation w is modified by shifting its response event, and H' remains a history unless the response event is shifted beyond the corresponding invocation event, in which case $f(r) < s(w) < f(w)$ holds and contradicts Assumption 3.3.

Next, consider Assumptions 3.2–3.4. Assumption 3.2 is preserved since the transformation keeps all the dictating writes, does not modify any reads, and does not introduce any new reads. Assumption 3.3 is preserved since the transformation does not introduce any new operations or change the relative order of the response of a read with respect to the invocation of its dictating write. Assumption 3.4 is established directly by line 5.

It remains to show that H is k-atomic if and only if H' is k-atomic. Suppose that H is k-atomic and let $<_T$ be a k-atomic total order for H. To prove that H' is k-atomic it suffices to show that $<_T \sim <_{H'}$. Suppose otherwise. Then H' contains operations op_1, op_2 such that $op_1 <_T op_2$ and $op_2 <_{H'} op_1$. Since $<_T \sim <_H$ and $op_1 <_T op_2$, it follows that $op_2 \not<_H op_1$. Since $op_2 <_{H'} op_1$ but $op_2 \not<_H op_1$, it also follows that op_2 is a write whose response event was shifted at line 5 and placed immediately before the response of some dictated read op_3. First note that $op_2 <_T op_3$ because a dictating write precedes its dictated reads in any k-atomic total order. If $op_3 = op_1$, then this implies $op_2 <_T op_1$, which contradicts $op_1 <_T op_2$. Otherwise $op_3 \neq op_1$ and in that case $op_3 <_{H'} op_1$ because $op_2 <_{H'} op_1$ and the response of op_3 is the next event in H' after the response of op_2. This implies $op_3 <_H op_1$ since op_3 is a read, and so $op_3 <_T op_1$ because $<_T \sim <_H$. Since $op_2 <_T op_3$ and $op_3 <_T op_1$ hold, $op_2 <_T op_1$ follows transitively, which once again contradicts $op_1 <_T op_2$.

Conversely, suppose that H' is k-atomic. Then there is a k-atomic total order $<_{T'}$ for H' and $<_{T'} \sim <_{H'}$. It follows from line 5 that for any pair of operations op_1 and op_2, $op_1 <_H op_2 \Rightarrow op_1 <_{H'} op_2$, and so $<_{T'} \sim <_{H'}$ in this case implies $<_{T'} \sim <_H$. As a result, H is k-atomic, as needed. ∎

In general, our algorithms assume that $2 \leq k \leq n$. As discussed in section 2, known algorithms can solve the cases $k \leq 2$ and $k \geq n$ in polynomial time, but do not work correctly when $2 < k < n$.

To simplify the presentation and analysis of our k-AV algorithms we first describe

two graph-theoretic representations of the input history. These representations are described in Definition 3.6, which is inspired by Misra's *before* relation on values [27], and in Definition 3.7. We use both definitions to characterize k-atomicity in Lemma 3.8, which we use extensively later in sections 4 and 5.

DEFINITION 3.6. *The* read value graph *for a history H, denoted $G_R(V, E_R)$, is a directed graph where V is the set of values written in H, and E_R contains an edge (v, v') if and only if $v \neq v'$ and there exists a read $r(v')$ in H such that $w(v) <_H r(v')$.*

DEFINITION 3.7. *The* write graph *for a history H, denoted $G_W(V, E_W)$, is a directed graph where V is the set of values written in H, and E_W contains an edge (v, v') if and only if $w(v) <_H w(v')$.*

Informally speaking, G_R and G_W both capture constraints with respect to k-atomicity on the order in which writes may appear to take effect. Figure 3.1 presents an example of these graph structures for a small input history. Operations in the history are illustrated as intervals of time following the style of diagrams used by Herlihy and Wing [20]. The horizontal dimension represents time and the vertical dimension is used only to separate the intervals visually. The history has two 3-atomic value orders, namely $T_1 = \langle 5, 2, 1, 3, 4\rangle$ and $T_2 = \langle 5, 2, 3, 1, 4\rangle$, but fails to satisfy 2-atomicity because $w(1)$ and $w(3)$ both separate $w(2)$ from $r(2)$. Consequently, the history is not 1-atomic either.

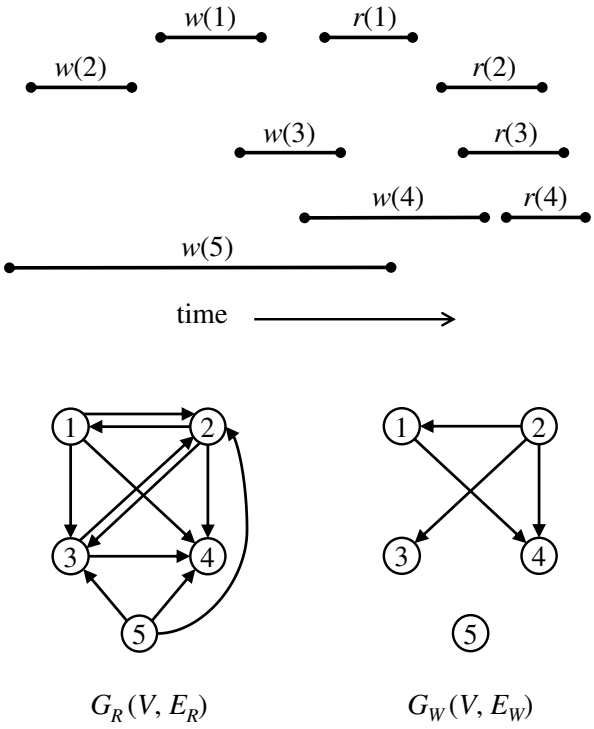

FIG. 3.1. *Example execution history (top) and corresponding graph representation (bottom).*

We use G_R and G_W to characterize k-atomicity in Lemma 3.8, which asserts that H is k-atomic if and only if there is an ordering of the writes that is consistent with the direction of the edges in G_W, and in which the edges of G_R never point

"backward" by more than $k-1$ positions. The intuition underlying backward edges is as follows: since an edge of G_R from v_i to v_j indicates that $w(v_i) <_H r(v_j)$ for some read $r(v_j)$, if v_j precedes v_i by k or more positions in some ordering of the writes, then that separates $r(v_j)$ from its dictating write $w(v_j)$ by k or more other writes including $w(v_i)$.

LEMMA 3.8. *A history H is k-atomic if and only if its corresponding write graph $G_W(V, E_W)$ has a topological ordering $T = \langle v_1, v_2, \ldots, v_{|V|} \rangle$, such that for any two vertices v_i and v_j in T, if the read value graph $G_R(V, E_R)$ for H contains an edge (v_i, v_j), then $j > i - k$.*

Proof. Recall from Definition 3.1 that H is k-atomic if and only if there exists a k-atomic total order for H. Intuitively, the proof strategy is to show that the topological ordering T defined in the statement of the lemma is the k-atomic value order corresponding to such a k-atomic total order, where the edges of G_R determine the staleness of individual reads with respect to the total order. For example, an edge from some value v_i to a value v_j that precedes v_i in T by $x > 0$ positions indicates that some $r(v_j)$ is separated from $w(v_j)$ in the k-atomic total order by at least x other writes including $w(v_i)$, as illustrated in Figure 3.2, and hence $r(v_j)$ returns the $(x+1)$st-latest or older value with respect to T.

$$w(v_j), \overbrace{\underbrace{\ldots, w(v_i)}_{x \text{ writes}}, \ldots}^{x \text{ or more writes}}, r(v_j)$$

FIG. 3.2. *Example of a total order of operations for a history H where $r(v_j)$ returns the $(x+1)$st-latest or older value due to the presence of an edge (v_i, v_j) in G_R.*

Suppose that H is k-atomic. Then there is a total ordering S of the operations in H such that $<_S \sim <_H$ and where each read returns the value assigned by one of the last k writes. Let $T = \langle v_1, v_2, \ldots, v_{|V|} \rangle$ be the sequence of values written in S and note that T is a topological ordering of G_W. To derive a contradiction, suppose that G_R contains an edge (v_i, v_j) such that $j \leq i - k$, which corresponds to the example in Figure 3.2 with $x \geq k$. Such an edge indicates that $w(v_i)$ happens before some read $r(v_j)$ in H; hence $w(v_i) <_S r(v_j)$. Furthermore, there are $i - j \geq k$ write operations, including $w(v_i)$, that separate $r(v_j)$ from its dictating write $w(v_j)$ in S. Since writes assign distinct values under Assumption 3.2, this implies that $r(v_j)$ does not return one of the last k written values in S, thus contradicting S being a k-atomic total order for H.

Conversely, suppose that $T = \langle v_1, v_2, \ldots, v_{|V|} \rangle$ exists and that $j > i - k$ holds for every edge (v_i, v_j) in G_R, which corresponds to the example in Figure 3.2 with $x < k$. We will exhibit a k-atomic ordering of the operations in H in three stages: (i) the operations will be placed in an order that is compatible with $<_H$; (ii) some reads will be shifted to ensure that a read never precedes its dictating write; (iii) we will show that in the final ordering each read returns one of the last k-values written.

Stage (i). Starting with an empty sequence, insert the write operations in the same order as their values in T. It follows easily that the ordering S constructed satisfies $<_S \sim <_H$ since T is a topological ordering of G_W. Next, form S' from S by processing the reads in increasing order of their start times, and placing each read as early as possible. That is, place each read immediately after the latest operation in S' that precedes it in $<_H$, or at the beginning if there is no such operation. We must

show that $<_{S'} \sim <_H$. To derive a contradiction, suppose that there are operations op_1 and op_2 such that $op_1 <_H op_2$ and $op_2 <_{S'} op_1$.

Case 1: op_1 and op_2 are both writes. This contradicts T being a topological ordering of G_W.

Case 2: op_1 and op_2 are both reads. Then $op_1 <_H op_2$ implies that op_1 starts before op_2, and so op_1 is processed before op_2. Therefore, when op_2 is processed it is placed in S' after op_1, which contradicts $op_2 <_{S'} op_1$.

Case 3: op_1 is a write and op_2 is a read. Then op_1 is once again processed before op_2, and we reach a contradiction as in Case 2.

Case 4: op_1 is a read and op_2 is a write. Without loss of generality suppose that op_1 is the first read processed that satisfies $op_1 <_H op_2$ and is placed in S' after op_2. Since each read is placed as early as possible, it follows that some other operation op_3 satisfying $op_3 <_H op_1$ was processed before op_1 and placed in S' after op_2. Thus, $op_2 <_{S'} op_3$. At the same time $op_3 <_H op_1$ and $op_1 <_H op_2$ imply $op_3 <_H op_2$. If op_3 is a write, then this contradicts Case 1. If op_3 is a read, then this contradicts the assumption that op_1 is the first read processed that precedes op_2 in $<_H$ and succeeds op_2 in $<_{S'}$.

Stage (ii). For every value, if one or more reads of the value precede their dictating write, then shift them to the right and place them immediately after the dictating write, while maintaining for each value the relative order of the reads of that value. Call the new sequence of operations S''. We will use induction on the number of reads shifted to show that $<_{S''} \sim <_H$. Let S''_i denote the sequence of operations after i shifts. In the base case $i = 0$ and $S''_i = S'$; hence $<_{S''_0} \sim <_H$ because stage (i) established $<_{S'} \sim <_H$. Now suppose for some $j \geq 0$ that $<_{S''_j} \sim <_H$ and consider S''_{j+1}. To derive a contradiction, suppose that H contains operations op_1, op_2 such that $op_1 <_H op_2$ but $op_2 <_{S''_{j+1}} op_1$. Since $<_{S''_j} \sim <_H$, it follows that op_1 is the $(j+1)$st shifted read. Let w_1 denote the dictating write of op_1. It follows from Assumption 3.4 that $f(w_1) < f(op_1)$, and so $op_1 <_H op_2$ implies $w_1 <_H op_2$. Thus, $op_2 \neq w_1$, and furthermore $op_2 <_{S''_j} w_1$ as otherwise op_1 would not be shifted past op_2 in the formation of S''_{j+1} from S''_j. Since $<_{S''_j} \sim <_H$, $op_2 <_{S''_j} w_1$ contradicts $w_1 <_H op_2$.

Stage (iii). We will show that S'' is a k-atomic ordering of the operations in H. By stage (i) S'' contains all the operations of H, and by stage (ii) no read precedes its dictating write. It remains to show that each read in S'' returns one of the last k-values written. To derive a contradiction, suppose that some read $r(v_j)$ does not do so; hence it is separated from its dictating write by k or more other writes, the last one of which is a write of some value v_i such that $i \geq j + k$. We will show that $w(v_i) <_H r(v_j)$. This implies that $G_R(V, E_R)$ contains an edge (v_i, v_j), which implies $j > i - k$ by our earlier assumption and contradicts $i \geq j + k$.

First, note that $r(v_j)$ was not shifted in stage (ii), as otherwise $i = j$ would hold, contradicting $i \geq j + k$. Next, to derive a contradiction suppose that $w(v_i) \not<_H r(v_j)$. Without loss of generality suppose that $r(v_j)$ is the first read processed in stage (i) that is placed in S' between $w(v_i)$ and the next write (if one exists), and for which $w(v_i) \not<_H r(v_j)$. Since $w(v_i) \not<_H r(v_j)$ and since $r(v_j)$ is placed after $w(v_i)$ in stage (i), which attempts to place reads as early as possible, it follows that some other read r is processed before $r(v_j)$ such that $w(v_i) <_{S'} r <_{S'} r(v_j)$ and $r <_H r(v_j)$. Then it follows from our assumption on $r(v_j)$ that $w(v_i) <_H r$. Since $w(v_i) <_H r$ and $r <_H r(v_j)$ it follows that $w(v_i) <_H r(v_j)$, which contradicts the earlier assumption that $w(v_i) \not<_H r(v_j)$. ∎

Applying Lemma 3.8 with $k = 3$ to the sample input illustrated in Figure 3.1, we see that 3-atomicity holds because $T_1 = \langle 5, 2, 1, 3, 4\rangle$ is a topological ordering of G_W such that the edges of G_R point backward by at most $k - 1 = 2$ positions (e.g., the edge from 3 to 2). On the other hand, applying the lemma with $k = 2$ we see that both 1 and 3 succeed 2 in any topological ordering T of G_W, but 1 and 3 both have edges to 2 in G_R, one of which must point backward by $2 > k-1$ positions with respect to T.

4. An efficient algorithm for a restricted class of histories. In this section, we present an efficient algorithm called GPO ("greedy plus obligations") that solves k-AV for arbitrary k in the special case when the input history satisfies Assumptions 3.2–3.4 as well as the following.

ASSUMPTION 4.1. *For any history H and for any value v written in H, the write $w(v)$ has some dictated read $r(v)$ such that $w(v) <_H r(v)$.*

Intuitively, this requirement constrains the k-AV problem by ensuring the presence of certain edges in the read value graph G_R for H. Specifically, for every pair of values v and v', Assumption 4.1 ensures that G_R contains the edge (v, v') if $f(w(v)) < f(w(v'))$, or the edge (v', v) if $f(w(v')) < f(w(v))$. In contrast, Assumptions 3.2–3.4 alone do not imply that v and v' are ordered by G_R in the case when $w(v)$ and $w(v')$ are concurrent.

4.1. Algorithm description. The main idea behind GPO is greedy plus obligations—hence the name. Like the LBT algorithm for 2-AV [16], GPO constructs a k-atomic value order in reverse or outputs NO if such an order does not exist. For each value placed in the k-atomic value order the algorithm evaluates any constraints this placement imposes on the choice of subsequent values and records these constraints using internal data structures.

At each iteration, GPO either places a value in the next position of the k-atomic value order or decides that the choice of values is overconstrained and outputs NO. In this context (and throughout this section), "next" refers to the position(s) immediately to the left of the suffix of the total order that has been constructed in prior iterations. Given multiple candidates for the next value to be placed, GPO greedily picks the one whose dictating write has the largest finish time. In some cases this choice is narrowed down to a subset of values that must be placed within the next $k - 1$ (or fewer) positions in the k-atomic value order due to constraints imposed on them by values that were placed earlier. Once GPO chooses the next value v, it identifies two types of constraints imposed on future choices by v with respect to the input history H: (i) if $w(v) <_H r(v')$ for some read of some value v' that has not yet been placed, then v' must be placed in one of the next $k - 1$ positions; (ii) if $w(v') <_H w(v'')$ for the value v' defined in the first constraint and some value v'' that has not yet been placed, then v'' must also be placed in the next $k - 1$ positions.

The constraints are tracked using $k-1$ sets of values called *obligation sets*, denoted by $B[1]$ to $B[k-1]$, initially all empty. The purpose of $B[i]$ is to record the values that must be placed in the next i positions of the k-atomic value order, and so naturally GPO must maintain the invariant $B[1] \subseteq B[2] \subseteq \cdots \subseteq B[k-1]$. An obligation set $B[i]$ is called *overfull*, *full*, or *underfull* if $|B[i]| > i$, $|B[i]| = i$, or $|B[i]| < i$, respectively. As GPO decides the next value to be placed it faces three possibilities. If there exists an overfull obligation set, then GPO simply outputs NO because it is impossible to place more than i values in the next i positions. If no obligation sets are overfull but at least one of them is full, then GPO first identifies the smallest full set, and

then among the values in that set it greedily picks the one whose dictating write has the largest finish time. Otherwise, all obligation sets are underfull, and GPO greedily chooses a value that has not yet been placed and whose dictating write has the largest finish time.

Having chosen the next value v, GPO updates the obligation sets by removing (if necessary) the value that was just placed, and then copying the remaining values from $B[i+1]$ to $B[i]$ for all i. Intuitively, if a value was supposed to be placed in the next $i+1$ positions at the beginning of a given iteration and is not chosen in the same iteration, then it must be placed in the next i positions at the beginning of the next iteration. In addition, GPO updates $B[k-1]$ by adding values determined according to the two types of constraints imposed by v, as described earlier. As long as no overfull obligation set is encountered, the algorithm continues to choose values and update the obligation sets until it has constructed a k-atomic value order for the input history. At that point GPO returns YES. The full procedure is presented as Algorithm 2.

The algorithm maintains $k-1$ separate obligation sets, as opposed to their union as a single set, because constraints placed on the values in these sets tighten after each iteration. For example, if $B[i]$ and $B[j]$ are both full and $i < j$, then placing the values from $B[i]$ in the next i positions is more urgent than placing the values from $B[j]$ in the next j iterations. In particular, the $j-i$ values of $B[j] \setminus B[i]$ must not be placed in the next i positions as otherwise there is no way to discharge the obligations represented by $B[i]$. This is why the algorithm chooses the next value v specifically from the smallest full obligation set (line 6) after discovering that at least one obligation set is full (line 5), as opposed to choosing the value from the union of the obligation sets (if not empty) whose dictating write has the largest finish time.

The intuition underlying lines 7 and 9 of GPO, which choose a value v with maximum finish time over any other candidate value v', follows from two observations. First, $w(v)$ happens before a subset of the operations that $w(v')$ happens before, and hence the choice of v as the next value in the reverse k-atomic order is the least constraining with respect to other values that have not yet been chosen. Second, choosing v in an earlier iteration than v' positions v' to the left of v in the k-atomic value order, which tends to be consistent with the direction of edges in G_R because $f(w(v')) < f(w(v))$ implies that G_R contains the edge (v', v) under Assumption 4.1. On the other hand, if the algorithm chose v' in an earlier iteration and v later, then v would precede v' by some number x of positions in the chosen total order over values. This opens the possibility that $x > k-1$, in which case the presence of the edge (v', v) in G_R would preclude the partially constructed total order from extending to a k-atomic value order because the edge reaches too far backward (see Lemma 3.8).

4.2. Examples. With respect to the example history presented in Figure 3.1, GPO works correctly only if we remove $w(5)$, which lacks a dictated read and hence violates Assumption 4.1. Consider the case $k = 3$ with $w(5)$ removed. The algorithm chooses 4 at line 9 in the first iteration, which imposes no additional constraints. It then chooses 3 at line 9 in the second iteration, and adds the values $\{1, 2\}$ to $B[2]$. In the third iteration it chooses 1 from $B[2]$ at line 7, and then copies 2 to $B[1]$. In the fourth iteration it chooses 2 from $B[1]$ at line 7. Thus, GPO computes the topological ordering $T = \langle 2, 1, 3, 4\rangle$ and returns YES. This response is correct because T is a 3-atomic value order for the input history without $w(5)$.

Next, consider the case $k = 2$ with $w(5)$ removed. As before, the algorithm chooses 4 in the first iteration. In the second iteration it chooses 3 at line 9, and adds

the values $\{1, 2\}$ to $B[1]$, causing a return of NO at line 19 because $|B[1]| = 2 > 1$. This response is correct because the input history is not 2-atomic, even without $w(5)$.

Finally, consider the full history in Figure 3.1 with $w(5)$ included. The algorithm chooses 4 at line 9 in the first iteration, which imposes no additional constraints. It then chooses 5 at line 9 in the second iteration, and adds the values $\{1, 2, 3\}$ to $B[2]$, causing a return of NO at line 19 because $|B[2]| = 3 > 2$. This response is incorrect because the input history is 3-atomic. Thus, GPO breaks because the input history fails to satisfy Assumption 4.1.

```
   Input: history H and integer k, 2 ≤ k ≤ n.
   Output: YES if H is k-atomic, NO otherwise.
   // suffix of a k-atomic value order for H
 1 T := empty sequence
   // set of values that have not yet been added to T
 2 U := set of values written in H
 3 for i := 1 to k − 1 do B[i] := ∅
 4 while U ≠ ∅ do
       // apply greedy heuristic
 5     if ∃ i : |B[i]| = i then
 6         ℓ := smallest i such that |B[i]| = i
 7         v := value in B[ℓ] such that w(v) has the largest finish time
 8     else
 9         v := value in U such that w(v) has the largest finish time
10     end
11     remove v from U and prepend it to T
       // update obligation sets
12     W := {v' | v' ∈ U and some read r(v') satisfies w(v) <_H r(v')}
13     W' := {v'' | v'' ∈ U and some value v' ∈ W satisfies w(v') <_H w(v'')}
14     for j := 1 to k − 2 do
15         B[j] := B[j + 1] \ {v}
16     end
17     B[k − 1] := (B[k − 1] ∪ W ∪ W') \ {v}
18     if ∃ i : |B[i]| > i then
19         return NO
20     end
21 end
22 return YES
```

Algorithm 2: The GPO algorithm.

4.3. Analysis. References to $G_R(V, E_R)$ and $G_W(V, E_W)$ in our analysis pertain to the read value graph and the write graph for the input history H (Definitions 3.6 and 3.7). These structures are needed only for analysis and therefore do not appear in the pseudocode of Algorithm 2. The symbols T_x, U_x, and $B_x[i]$ denote the values of T, U, and $B[i]$ at the end of iteration x of the outer loop.

LEMMA 4.2. *The outer loop of the algorithm provides the following invariant:*

(a) *for every value v accessed in H, v is either in T_x or in U_x (but not in both); and*

(b) *for every* i, $1 \leq i \leq k-1$, *if there exists a value* $v \in B_x[i]$ *and a value* $v' \in U_x$ *such that* $w(v) <_H w(v')$, *then* $v' \in B_x[i]$.

Proof. We proceed by induction on x.

Basis: $x = 0$. Initially U_0 is the set of all values written in H, which equals the set of all values accessed in H by Assumption 3.2, and $T_0 = \langle\rangle$. This implies part (a). Part (b) holds trivially because there is no $w \in B_0[i]$ for any i.

Induction step. Suppose that parts (a)–(b) hold for some iteration number x. Assuming that the outer loop has another iteration, consider the state of affairs at the end of iteration $x+1$. Part (a) follows from line 11 where U_{x+1} is formed by removing the chosen value v from U_x and T_{x+1} is formed by prepending v to T_x. For part (b), consider any $z \in B_{x+1}[i]$, where $1 \leq i \leq k-1$, and any $z' \in U_{x+1}$ such that $w(z) <_H w(z')$. We must show that $z' \in B_{x+1}[i]$ as well. Since we assume $z' \in U_{x+1}$, note that z' cannot be the value v chosen at line 7 or 9 during iteration $x+1$, as otherwise z' is removed from U_{x+1} at line 11 during the same iteration. Also note that $z' \in U_x$ since $z' \in U_{x+1}$ and since elements are never added to U after line 2.

Case A: $i = k-1$. If $z \in B_x[i]$, then $z' \in B_x[i]$ by part (b) for iteration x because $z' \in U_x$ and $w(z) <_H w(z')$. Since $z' \neq v$, as argued earlier, it follows from line 17 that z' remains in $B_{x+1}[i]$, as needed. Otherwise, $z \notin B_x[i]$ and so z is one of the values added to $B_{x+1}[i]$ during iteration $x+1$. Thus, z is a value in one of the sets W and W' computed at lines 12 and 13. Next, we will show that z' is added to W' at line 13. If z is added to W, then z' is added to W' at line 13 because $z' \in U_{x+1}$ and $w(z) <_H w(z')$. Otherwise, z is added to W' and so W contains some value $z'' \in U_{x+1}$ such that $w(z'') <_H w(z)$, in which case $w(z) <_H w(z')$ implies $w(z'') <_H w(z')$ and so z' is added to W' at line 13 because $z'' \in W$ and $z' \in U_{x+1}$. Since $z' \neq v$, this implies that z' is added to $B_{x+1}[i]$ at line 17, as needed.

Case B: $i < k-1$. Since $z \in B_{x+1}[i]$, it follows from line 15 that $z \in B_x[i+1]$. Furthermore, since $z' \in U_x$, part (b) for iteration x implies that $z' \in B_x[i+1]$ as well. Since $z' \neq v$, as argued earlier, line 15 implies that $z' \in B_{x+1}[i]$, as needed. ∎

LEMMA 4.3. *Suppose that the algorithm outputs YES. Then the sequence of values* T *obtained when line* 22 *is reached is a topological ordering of* G_W.

Proof. Every value added to T is prepended at line 11, and the choice of each value is made at line 7 or 9. Suppose that the final sequence of values chosen is $T = \langle v_1, v_2, \ldots, v_{n_w} \rangle$. To derive a contradiction, suppose that T contains values v_i, v_j such that $j < i$ and yet $w(v_i) <_H w(v_j)$. Then after the iteration x when v_i is chosen, the value v_j remains in U_x. Since elements are never added to U after line 2, this implies that $v_j \in U_{x-1}$, and so during iteration x the value v_i is chosen at line 7 from $B[\ell]$ and not at line 9, as otherwise the algorithm would choose v_j or some other value at line 9 instead of v_i because $w(v_i) <_H w(v_j)$. Thus, $v_i \in B_{x-1}[\ell]$ and $v_j \in U_{x-1}$ hold. Since $w(v_i) <_H w(v_j)$, it follows from Lemma 4.2 (b) that $v_j \in B_{x-1}[\ell]$. This contradicts the algorithm choosing v_i over v_j from $B[\ell]$ at line 7 during iteration x. ∎

LEMMA 4.4. *If the algorithm outputs YES, then* H *is* k*-atomic.*

Proof. Intuitively, we want to show that a YES output indicates that the algorithm successfully constructed a k-atomic value order for H by repeatedly prepending values to T. Interpreting k-atomicity according to Lemma 3.8, the proof obligation is to show that T is a topological ordering of G_W, and that the edges of G_R do not reach too far backward with respect to T. We will establish the latter by arguing that any edge (a, b) of G_R that reaches backward would result in the addition of b to the obligation sets during the iteration that places a, and hence the placement of b in one

of the next $k-1$ iterations, thus contradicting the assumption that the edge reaches too far (i.e., k or more positions) backward.

Suppose that the algorithm outputs YES. Then by Lemma 4.3 the sequence of values $T = \langle v_1, v_2, \ldots, v_{n_w} \rangle$ when line 22 is reached is a topological ordering of G_W. It suffices to show that for every $1 \leq i,\ j \leq n_w$ if G_R contains an edge (v_i, v_j), then $j > i - k$, in which case Lemma 3.8 implies that H is k-atomic. To derive a contradiction, suppose otherwise. Then G_R contains an edge (v_i, v_j) such that $j \leq i - k < i$ (since $k > 1$), and so v_i is chosen in an earlier iteration than v_j. Consider iteration x of the outer loop, where v_i is chosen at line 7 or at line 9. Since $(v_i, v_j) \in E_R$, it follows that value v_j has a read r_j such that $w(v_i) <_H r_j$. Then v_j is added to W at line 12 in iteration x because v_j has not yet been chosen, and hence $v_j \in U_x$ by Lemma 4.2 (a). As a result, v_j is added to $B[k-1]$ at line 17 during iteration x, and so $v_j \in B_x[k-1]$. Now consider iteration $x+y$ for $1 \leq y < k-1$, which exists provided that $k > 2$. Value v_j is not added to T in any of these iterations, the last of which is iteration $x+k-2$, since we assume that $j \leq i-k$ and so at the earliest v_j can be added during iteration $x+k$. Thus, $v_j \in U_{x+k-2}$ holds by Lemma 4.2 (a) and since we are considering $k > 2$. Since $v_j \in B_x[k-1]$, this implies $v_j \in B_{x+k-2}[k-1]$ because a value is not removed from $B[k-1]$ until the iteration in which it is removed from U and added to T. Furthermore, line 15 implies that $v_j \in B_{x+y}[k-1-y]$ for $1 \leq y < k-1$; in particular $v_j \in B_{x+k-2}[1]$. Since we assume that the algorithm outputs YES, it follows from line 18 during iteration $x+k-1$ that $|B_{x+k-2}[1]| \leq 1$ and so $v_j \in B_{x+k-2}[1]$ implies $B_{x+k-2}[1] = \{v_j\}$. As a result, $B[1] = \{v_j\}$ at the beginning of iteration $x+k-1$, which implies that v_j is chosen at line 7 during the same iteration. This contradicts the earlier observation that v_j is not chosen until iteration $x+k$ or later. ∎

LEMMA 4.5. *If H is k-atomic, then the algorithm outputs YES.*

Proof. Intuitively, we want to show that if H is k-atomic, then the algorithm determines a possible k-atomic value order for H by repeatedly prepending values to T and successfully avoids an overfull obligation set that would lead to a NO response at line 19. Interpreting k-atomicity according to Lemma 3.8, the proof obligation is to show that the constructed sequence T is indeed a topological ordering of G_W, and that the edges of G_R do not reach too far backward with respect to T as a result of each obligation set remaining either underfull or full after each iteration.

The proof proceeds by induction, showing that after each iteration the partially constructed sequence T is a suffix of some k-atomic value order T'. The main technical challenge in the proof is that in general the history H may have multiple k-atomic value orders. As a result, if iteration x constructs a sequence T_x that can be extended to a k-atomic value order T'_x, there is no guarantee that the sequence T_{x+1} obtained by prepending a value to T_x in the next iteration can also be extended to the *same* T'_x. The proof therefore describes how T'_x can be modified carefully, if needed, to obtain another k-atomic value order for H that extends T_{x+1}.

Suppose that H is k-atomic. We will show that the algorithm outputs YES by proving the following predicate $P(x)$ for all $0 \leq x \leq n_w$:

(a) the algorithm executes line 11 at least x times and computes a sequence $T_x = \langle v_1^x, v_2^x, \ldots, v_x^x \rangle$ that is the suffix of some k-atomic value order for H;
(b) for any i such that $1 \leq i \leq k-1$ and for any value v, if $v \in B_x[i]$, then v is one of the i immediate predecessors of v_1^x in every k-atomic value order for which T_x is a suffix, and hence $|B_x[i]| \leq i$; and
(c) for any value v, if G_R contains an edge (a, b) such that a is $i \geq 0$ positions to

the right of v_1^x in T_x and b is not in T_x, then $b \in B_x[k-1-j]$ for all j such that $0 \leq j \leq i$.

Part (b) of the predicate ensures that the algorithm never exits at line 19, and therefore outputs YES at line 22, as needed.

We proceed by induction on x.

Basis: $x = 0$. Parts (a) and (c) follow trivially since $T_x = \langle\rangle$. Part (b) follows trivially since $B_x[i] = \emptyset$ for all i.

Induction step. Choose an arbitrary x, $0 \leq x < n_w$, and suppose that $P(x)$ holds. First, we will show that the algorithm reaches line 11 and computes T_{x+1} in iteration $x+1$. To derive a contradiction, suppose otherwise, meaning that line 11 is executed fewer than $x+1$ times. It follows from $P(x)$-(a) that iteration x occurs and extracts a value from U at line 11. Furthermore, since U initially contains n_w elements, and since the algorithm removes one value from U per iteration, it follows that iteration x leaves $n_w - x > 0$ elements in U. Therefore, the fact that line 11 is not executed $x+1$ times is due to termination at line 19 in iteration x, and not due to the loop guard at line 4. This implies $|B_x[i]| > i$ for some i, which contradicts $P(x)$-(b).

Proof of $P(x+1)$-(a). We must show that there exists a k-atomic value order T'_{x+1} for H such that T_{x+1} is a suffix of T'_{x+1}. Suppose otherwise. By $P(x)$-(a) there exists a k-atomic value order T'_x for H such that T_x is a suffix of T'_x, and T_{x+1} is not a suffix of T'_x, as otherwise $P(x+1)$-(a) would hold with $T'_{x+1} = T'_x$. Since T_x is a suffix of both T_{x+1} and T'_x this implies that T_{x+1} and T'_x disagree on the element in position $x+1$ counting from the end. Let v be the element in this position in T'_x and suppose that T_{x+1} (i.e., the algorithm) instead chooses $v' \neq v$ in this position. Furthermore, suppose without loss of generality that T'_x is chosen so that v' appears in the latest possible (i.e., rightmost) position preceding v. Let S denote the set of values comprising v, v', as well as any other value that appears between v and v' in T'_x. The scenario under consideration is illustrated in Figure 4.1.

$$
\begin{array}{lcccccccccc}
T_x & = & & & & & \langle v_1^x, & v_2^x, & \ldots, & v_x^x \rangle \\
 & = & & & & & \langle v_2^{x+1}, & v_3^{x+1}, & \ldots, & v_{x+1}^{x+1} \rangle \\
T_{x+1} & = & & & & \langle v_1^{x+1}, & v_2^{x+1}, & v_3^{x+1}, & \ldots, & v_{x+1}^{x+1} \rangle \\
 & = & & & & \langle v', & v_2^{x+1}, & v_3^{x+1}, & \ldots, & v_{x+1}^{x+1} \rangle \\
T'_x & = & \langle \ldots, & v' & \ldots, & \underbrace{v,}_{\text{position } x+1 \text{ from the end}} & v_2^{x+1}, & v_3^{x+1}, & \ldots, & v_{x+1}^{x+1} \rangle \\
 & & & & & \underbrace{\qquad\qquad}_{S} & & & &
\end{array}
$$

FIG. 4.1. *Relationship between T_x, T_{x+1}, T'_x, and S.*

In the remainder of the proof we will derive a contradiction, but first we establish some salient properties of v and v' in Claims 4.6–4.9.

CLAIM 4.6. *Value v' is one of the $k-1$ immediate predecessors of v in T'_x. Furthermore, if v' is chosen at line 7 from $B_x[\ell]$, then v' is one of the $\ell - 1$ immediate predecessors of v in T'_x.*

Proof. If v' is chosen at line 7 from $B_x[\ell]$ for some ℓ, then by $P(x)$-(b) value v' is one of the ℓ immediate predecessors in T'_x of the first element of T_x (i.e., the xth element from the end in T'_x). This implies that v' is one of the $\ell - 1$ immediate predecessors of v in T'_x, as needed. Otherwise, v' is chosen at line 9 as the element of U_x with maximum finish time, and so $f(w(v')) > f(w(v))$. By Assumption 4.1 value

v' has a read r in H such that $w(v') <_H r$; hence $w(v) <_H r$ and so G_R contains the edge (v, v'). Since T'_x is a k-atomic value order for H and since v' precedes v in T'_x, it follows from Lemma 3.8 that v' is one of the $k-1$ immediate predecessors of v in T'_x, as needed. ∎

CLAIM 4.7. *$S \subseteq U_x$, $|S| \leq k$, and if v' is chosen at line 7 from $B_x[\ell]$, then $|S| \leq \ell$. Furthermore, v' has the maximum finish time of any value in S.*

Proof. First observe that by definition v' is the earliest in T'_x of all the elements in S, and v is the latest. Since v precedes in T'_x all the elements of T_x, we see that none of the elements in S are in T_x, and hence all of them are in U_x by Lemma 4.2 (a). Thus, $S \subseteq U_x$. Furthermore, Claim 4.6 implies that $|S| \leq k$, and that if v' is chosen at line 7 from $B_x[\ell]$, then $|S| \leq \ell$. Next, we focus on the finish time of v'. If v' is chosen at line 7 from $B_x[\ell]$, then $|B_x[\ell]| = \ell$ by line 5. Moreover, $P(x)$-(b) implies that the ℓ distinct values in $B_x[\ell]$ include v and its $\ell - 1$ immediate predecessors in T'_x. Since we showed that $|S| \leq \ell$ in this case, v' must be one of these predecessors, and so $S \subseteq B_x[\ell]$. By the choice of v' at line 7 we see that v' has maximum finish time among all elements in S. Otherwise, v' is chosen at line 9 from U_x. Since we showed that $S \subseteq U_x$, it follows by the choice of v' at line 9 that v' has maximum finish time among all elements in S. ∎

CLAIM 4.8. *For any value $v'' \in S \setminus \{v'\}$, if the sequence T'_{x+1} obtained from T'_x by swapping v' with v'' is a topological ordering of G_W, then $v'' \in B_x[|S| - 1]$.*

Proof. Fix any value $v'' \in S \setminus \{v'\}$, and let T'_{x+1} be the sequence obtained from T'_x by swapping v' with v''. Suppose that T'_{x+1} is a topological ordering of G_w. Recall that T'_x is a k-atomic value order for H such that T_x is a suffix of T'_x, and that we assume v' appears in T'_x at the latest possible position preceding v. Then T'_{x+1} is not a k-atomic value order for H as otherwise that would contradict the definition of v' because T_x is also a suffix of T'_{x+1} and v' appears later in T'_{x+1} than in T'_x. Since we assume that T'_{x+1} is a topological ordering of G_W and yet it is not a k-atomic value order for H, it follows from Lemma 3.8 that G_R contains an edge (a, b) such that b precedes a in T'_{x+1} by k or more positions. Furthermore, since T'_x is a k-atomic value order for H, it follows from Lemma 3.8 that a and b cannot both occupy the same positions in T'_x as they do in T'_{x+1}; hence at least one of these two values shifts as a result of swapping v' with v'' in T'_x (i.e., $\{a, b\} \cap \{v', v''\} \neq \emptyset$). In particular, b must shift to the left or a must shift to the right in the transformation from T'_x to T'_{x+1}. We will prove that a does not shift to the right and hence b shifts to the left. To derive a contradiction, suppose that this is not so, and consider the following two cases.

Case A: a shifts to the right and b shifts to the left. Then $a = v'$ and $b = v''$, and so $a, b \in S$. Since we assume that b precedes a and b is not one of the $k-1$ immediate predecessors of a in T'_{x+1}, there are at least $k - 1$ elements between b and a in T'_{x+1}. Since the transformation from T'_x to T'_{x+1} preserves the property that the elements of S remain contiguous, this implies that $|S| > k$. This contradicts Claim 4.7, which asserts that $|S| \leq k$.

Case B: a shifts to the right and b does not shift. Then $a = v'$. If $b \in S$, then $\{a, b\} \subseteq S$, and since the elements of S remain contiguous in T'_{x+1}, the fact that b precedes a in T'_{x+1} by k or more positions contradicts Claim 4.7, which asserts that $|S| \leq k$. Thus, $b \notin S$ and so b precedes a in both T'_{x+1} and T'_x. Since v' is the earliest element of S in T'_x and v is the latest, it follows that b also precedes v in T'_x, and that swapping v' with v'' places v' in T'_{x+1} at a position no later than that of v in T'_x. This implies that the number of elements between b and v in T'_x is greater than or equal to

the number of elements between b and $a = v'$ in T'_{x+1} because b is in the same position in T'_x as in T'_{x+1}. Since we assume that b precedes a by more than $k-1$ positions in T'_{x+1}, this implies that b also precedes v by more than $k-1$ positions in T'_x. However, since G_R contains the edge (v', b) and since $f(w(v)) < f(w(v'))$ holds by Claim 4.7, it follows that G_R also contains the edge (v, b), which contradicts Lemma 3.8 with respect to T'_x.

Thus a does not shift and b shifts to the left. This implies $b = v''$; hence $b \neq v'$. Since we assume that b precedes a by more than $k-1$ positions in T'_{x+1}, it follows that a is not an element of S. This is because the elements of S are contiguous in T'_{x+1} and there are at most k of them by Claim 4.7; hence a must follow all these elements in both T'_x and T'_{x+1} given its distance from b. At the same time, since T'_x is a k-atomic value order for H, Lemma 3.8 implies that b is one of the $k-1$ immediate predecessors of a in T'_x. Now suppose that a follows the immediate successor of v in T'_x; namely $v_2^{x+1} = v_1^x$ (i.e., the first element of T_x and the second element of T_{x+1}, as shown in Figure 4.1) by exactly $i \geq 0$ positions. Then b is a value at most $k-1-i$ positions to the left of v_2^{x+1} in T'_x. Since $b \neq v'$ and v' is the earliest in T'_x of all the values in S, we see that $k-1-i < |S|$; hence $i \geq k - |S|$. Since all elements of S appear in T'_x before v_2^{x+1}, it follows that $b = v''$ is not in T_x; hence applying $P(x)$-(c) we see that $b \in B_x[k-1-j]$ for all j such that $0 \leq j \leq i$. Letting $j = k - |S|$ this implies $v'' \in B_x[|S|-1]$, as needed. ∎

CLAIM 4.9. *For any value $v'' \in S \setminus \{v'\}$, if the sequence T'_{x+1} obtained from T'_x by swapping v' with v'' is not a topological ordering of G_W, then $v'' \in B_x[|S|-1]$.*

Proof. Since T'_x is a k-atomic value order for H, Lemma 3.8 implies that T'_x is a topological ordering of G_W. Since T'_{x+1} is not a topological ordering of G_W, it follows that swapping v' with v'' positions two values in T'_{x+1} in reverse order with respect to the edge set of G_W. One of these values is v' or v'', and since the values in S are contiguous in T'_x we see that the other value, say v''', is also an element of S (possibly $v''' \in \{v', v''\}$). Furthermore, since v' has the largest finish time of all values in S by Claim 4.7, it must be the case that v'' and v''' are out of order in T'_{x+1} with respect to G_W—not v' and v'''. Since swapping v' with v'' moves v'' to an earlier position, this implies that v'' precedes v''' in T'_{x+1} and yet $w(v''') <_H w(v'')$. Now let $z \neq v'$ be a value in S that minimizes $f(w(z))$. Since v' and z have the largest and smallest finish times, respectively, of all values in S, swapping v' with z in T'_x must yield another topological ordering of G_W, and hence $z \in B_x[|S|-1]$ holds by Claim 4.8. If $z = v''$, the claim follows. Assume now that $z \neq v''$. Then $w(z) <_H w(v'')$ because $f(w(z)) \leq f(w(v'''))$ by the definition of z and because $w(v''') <_H w(v'')$. Furthermore, since v'' is chosen from S, it follows from Claim 4.7 that $v'' \in U_x$. Thus, applying Lemma 4.2 (b) with $z \in B_x[|S|-1]$ and $v'' \in U_x$ we obtain $v'' \in B_x[|S|-1]$, as needed. ∎

We now return to the proof of $P(x+1)$-(a). Claims 4.8 and 4.9 together imply that for any value $v'' \in S \setminus \{v'\}$, $v'' \in B_x[|S|-1]$ holds. As a result, $S \setminus \{v'\} \subseteq B_x[|S|-1]$ and hence $|B_x[|S|-1]| \geq |S|-1$. This implies $|B_x[|S|-1]| = |S|-1$ because $|B_x[i]| \leq i$ holds for all i by $P(x)$-(b). Since $|B_x[|S|-1]| = |S|-1$, the condition at line 5 is true during iteration $x+1$, and hence v' is chosen at line 7 rather than at line 9. In that case, $|B_x[|S|-1]| = |S|-1$ implies $|S|-1 \geq \ell$ by the minimality of ℓ, which is ensured by line 6. This contradicts Claim 4.7, which asserts that $|S| \leq \ell$ and implies that $|S|-1 < \ell$. This completes the proof of $P(x+1)$-(a).

Proof of $P(x+1)$-(b). Consider $B_{x+1}[i]$ for any i, and any value $z \in B_{x+1}[i]$. We must show that z is one of the i immediate predecessors of v_1^{x+1} in any k-atomic value

order T' for H such that T_{x+1} is a suffix of T'. If $i < k-1$, then $B_{x+1}[i] \subseteq B_x[i+1]$ by line 15, and so $z \in B_x[i+1]$. By $P(x)$-(b) it follows that z is one of the $i+1$ immediate predecessors of v_1^x in any k-atomic value order for H for which T_x is a suffix. Since T_x is a suffix of T_{x+1} and since v_1^{x+1} is the immediate predecessor of v_1^x in T_{x+1}, this implies that z is one of the i immediate predecessors of v_1^{x+1} in any k-atomic value order for H for which T_{x+1} is a suffix, including T' itself. Otherwise, $i = k-1$ and so $B_{x+1}[i]$ comprises the union of $B_x[i]$ and the sets W and W' computed at lines 12 and 13, excluding v_1^{x+1} itself. If $z \in B_x[i]$, then by $P(x)$-(b), the value z is one of the i immediate predecessors of v_1^x in any k-atomic value order for H for which T_x is a suffix, and hence one of the $i-1 < i$ immediate predecessors of v_1^{x+1} in any k-atomic value order for H for which T_{x+1} is a suffix, including T' itself. Otherwise, z is added to $B[i]$ during iteration $x+1$ through the union with W and W' at line 17. If $z \in W$, then z has a read r such that $w(v_1^{x+1}) <_H r$, and so G_R contains an edge $(v_1^{x+1}, z) \in E_R$. Then, by Lemma 3.8, the value z is one of the $k-1 = i$ values immediately preceding v_1^{x+1} in T', as needed. Otherwise, $z \in W'$ and so W contains some z' such that $w(z') <_H w(z)$. Then z' is one of the $k-1$ values immediately preceding v_1^{x+1} in T' by the same argument as applied for z in the previous case where $z \in W$. Since $w(z') <_H w(z)$, this implies that z is either one of the $k-1$ values immediately preceding v_1^{x+1} in T', or some value that appears in a later position. Since $W' \subset U_x$ by line 13, we see that z is not in T_x by Lemma 4.2 (a), and furthermore that $z \neq v_1^{x+1}$, as otherwise it would be removed at line 17. Thus, z is exactly one of the $k-1$ values immediately preceding v_1^{x+1} in T', as needed.

Proof of $P(x+1)$-(c). Suppose that G_R contains an edge (a, b) such that a is $i \geq 0$ positions to the right of v_1^{x+1} in T_{x+1} and b is not in T_{x+1}. We must show that $b \in B_{x+1}[k-1-j]$ for all j such that $0 \leq j \leq i$. If $i = 0$, then $a = v_1^{x+1}$ and we must show that $b \in B_{x+1}[k-1]$. Since G_R contains the edge (a, b), we see that b has a read $r(b)$ such that $w(a) <_H r(b)$. Furthermore, since b is not in T_{x+1} we see that $b \in U_{x+1}$ by Lemma 4.2 (a). Since $a = v_1^{x+1}$, $w(a) <_H r(b)$, and $b \in U_{x+1}$, the algorithm adds b to the set W at line 12 during iteration $x+1$. Since b is not in T_{x+1}, it follows that $b \neq v_1^{x+1}$, and so b is then added to $B_{x+1}[k-1]$ at line 17 during iteration $x+1$. Thus, $P(x+1)$-(c) holds when $i = 0$. Otherwise, $i > 0$ and $a \neq v_1^{x+1}$; hence a is $i-1$ positions to the right of v_1^x in T_x, which is a suffix of T_{x+1}. In that case $P(x)$-(c) implies that $b \in B_x[k-1-j]$ for all j such that $0 \leq j \leq i-1$. Since b is not in T_{x+1}, it follows that $b \neq v_1^{x+1}$; hence b is not removed from the obligation sets at lines 15 and 17 during iteration $x+1$. As a result, b is copied to $B_{x+1}[k-1-j]$ for all j such that $1 \leq j \leq i$ at line 15, and b remains in $B_{x+1}[k-1]$ by line 17. Thus, $b \in B_{x+1}[k-1-j]$ for all j such that $0 \leq j \leq i$, and $P(x+1)$-(c) holds when $i > 0$.

This completes the proof of Lemma 4.5. ∎

THEOREM 4.10. *The algorithm outputs YES if and only if H is k-atomic. Furthermore, the algorithm has time complexity $O(n^2 + n \cdot k \log k)$, where n is the number of operations in the input history H.*

Proof. If the algorithm outputs YES, then H is k-atomic by Lemma 4.4. If H is k-atomic, then the algorithm outputs YES by Lemma 4.5. Thus, the algorithm outputs YES if and only if H is k-atomic.

For time complexity, suppose that all the set variables (i.e., U and $B[1]$ to $B[k-1]$) are represented using a balanced tree structure sorted by the finish time of the corresponding dictating write. Since these variables store distinct values from 1 to $n_w \leq n$, it follows that insertions, deletions, and maximum element lookups take $O(\log n)$ time, and the number of elements can be queried in $O(1)$ time. Furthermore,

we can assume without loss of generality that $|B[i]| \leq i$, as otherwise we can stop the algorithm and return NO, as per line 18. Thus, $B[i]$ supports fundamental operations in $O(\log i)$ time. Let the sequence T be represented using a linked list with an $O(1)$-time prepend operation. Initially T is empty and U is populated with all the values in H, and so lines 1–2 require $O(n \log n)$ time. The outer loop has at most $n_w \leq n$ iterations and so we consider the cost of each iteration. Line 5 requires $O(1)$ time per obligation set and hence $O(k)$ time in total. Lines 6–9 require $O(k + \log n)$ time. Line 11 requires $O(\log n)$ time to remove v from U and $O(1)$ time to prepend v to T. Line 12 requires a scan of U in $O(n)$ time, and $O(n)$ time to record W (e.g., using a linked list). Line 13 requires computing the smallest finish time of any dictated write for the values in W, which takes $O(n)$ time, and another scan of U in $O(n)$ time to compute W'. The inner loop at lines 14–16 requires shuffling $k-1$ obligation sets, which requires $O(k)$ time if pointers or references to the structures are copied rather than the structures themselves. Furthermore, v must be deleted from each set, which requires $O(\log i)$ time for $B[i]$ and $O(k \log k)$ time in total. Line 17 entails $O(k)$ insertions into $B[k-1]$ because, as explained earlier, the algorithm can be stopped as soon as $|B[k-1]| > k-1$. The total cost of these insertions is $O(k \log k)$. Finally, the test at line 18 requires $O(1)$ steps per obligation set, or $O(k)$ time in total. Thus, each iteration runs in $O(n + k \log k)$ time. The overall time complexity of the algorithm is $O(n^2 + n \cdot k \log k)$, as needed. ∎

5. A general algorithm. In this section we present a k-AV algorithm called CGS ("configuration graph search") for any history satisfying Assumptions 3.2–3.4, whose worst-case time complexity is polynomial only in the special case when both k and our specific measure of write concurrency are bounded by constants.

5.1. Algorithm description. The algorithm uses the graph-theoretic representations of the input history H described in section 3 as Definitions 3.6 and 3.7. Its complexity depends on both the size of these graph structures and on a formal notion of concurrency among write operations captured in Definition 5.1.

DEFINITION 5.1. *The* write concurrency *of a history H is the maximum number of writes that any one write overlaps with (not necessarily at a common point), including itself.*

The algorithm works by analyzing possible topological orderings of G_W, which is why its time complexity is sensitive to the write concurrency, denoted in this section by m. For example, if $m = 1$, then the write operations in H are totally ordered by $<_H$ and k-atomicity is decided easily to placing reads carefully around writes, as explained in [16]. On the other hand, for $m > 1$, the number of possible topological orderings in the worst case grows exponentially with the number of writes n_w, which makes it impractical to enumerate such orderings. We therefore adopt an approach that computes pieces of the possible topological orderings called *configurations*, which are formalized in Definition 5.2, and then searches for ways to combine the pieces to form a full topological ordering. This approach is inspired by Saxe's algorithm for deciding whether the bandwidth of a graph is less than or equal to a given parameter k [31]. Saxe's dynamic programming technique attempts to construct a linear graph layout that exhibits a bandwidth of k or less by stringing together structures that represent subsets of at most k vertices and the edges incident on them. Our configurations are similar conceptually to these structures, but constrained by the edges of two graphs (G_R and G_W) representing the input history H, rather than a single input graph.

Each configuration computed by the CGS algorithm is a contiguous subsequence

of a k-atomic value order for H, and must satisfy internally any constraints related to edges of G_R and G_W in accordance with Lemma 3.8 (see Definition 5.2). Configurations are then composed carefully, when possible, to form longer contiguous subsequences of a k-atomic value order for H. Specifically, the algorithm attempts to string pairs of overlapping configurations together and reduces k-AV to the problem of finding a path of a certain length in a graph whose vertices represent the configurations and whose edges indicate that two configurations can be composed safely (see Lemma 5.7). For example, a configuration $C = \langle v_a, v_b, v_c \rangle$ might be considered for composition with a configuration $C' = \langle v_b, v_c, v_d \rangle$ because C' begins with a suffix of C. If C and C' meet certain additional requirements discussed shortly (see Definition 5.3), then we say that C' *extends* C, and that their composition *induces* the sequence of values $\langle v_a, v_b, v_c, v_d \rangle$ (see Definition 5.4). The graph of configurations (denoted G_C) would then contain a directed edge (C, C').

The main technical challenge in stringing together configurations is to ensure that the constraints related to edges of G_R and G_W compose correctly, meaning that if they hold at the level of individual vertices or edges in G_C then they hold automatically for each path in G_C. This enables polynomial running time if m and k are bounded from above by constants, as otherwise the number of paths that must be checked grows exponentially with n_w in the worst case. To begin with, if configurations C and C' individually are subsequences of a topological ordering for G_W, and C' extends C, then their composition should automatically be a longer subsequence of a topological ordering for G_W. For example, if $C = \langle v_e, v_f \rangle$ and $C' = \langle v_g, v_h \rangle$ are configurations, then it should never be the case that C' extends C (i.e., $v_g = v_f$) and C simultaneously extends C' (i.e., $v_e = v_h$), as otherwise their repeated composition would induce a sequence of nonunique values. Similarly, if C' extends C, then it should never be the case that $v_h <_H v_e$, as otherwise their composition would order v_e before v_h, counter to any topological ordering of G_W. We avoid such undesirable scenarios in two ways. First, we require that if C begins with v_e and C' ends with v_h, then C' extends C only if $v_e \neq v_h$ and $v_h \not<_H v_e$ (see Definition 5.3). Second, we require that each configuration C contains at least m values (see Definition 5.2), which ensures that any value v not present in C is ordered by $<_H$ (i.e., by the edge set of G_W) with respect to at least one value in C. As we show later on in Lemma 5.5 and Corollary 5.6, these properties ensure that all paths through the graph G_C of configurations are finite and, if long enough, induce topological orderings of G_W.

Constraints related to edges of G_R are dealt with by ensuring that each configuration C preserves them internally (see Definition 5.2 (b)), and also by "pinning down" any backward edges of G_R with respect to the last value v of C. More precisely, we require that if G_R contains an edge (v, v') for some $v' \neq v$, then v' is either one of the $k-1$ immediate predecessors of v in C, or a value outside of C that is ordered after some value in C by $<_H$ (see Definition 5.2 (c)). This ensures that in any topological ordering of G_W obtained by composing C with other configurations, (v, v') either reaches backward by at most $k-1$ positions or reaches forward, as required by Lemma 3.8. The configuration size required for this technique to work as stated is at least k. Thus, an overall configuration size of $\max(m, k)$ is sufficient to deal with constraints imposed by edges of both G_W and G_R using the techniques described.

DEFINITION 5.2. *An* (m, k)-configuration *with respect to a read value graph* $G_R(V, E_R)$ *and a write graph* $G_W(V, E_W)$ *is any sequence* $S = \langle v_1, v_2, \ldots, v_\ell \rangle$ *of* $\ell = \max(m, k)$ *values that satisfies the following properties:*

(a) *S is a contiguous subsequence of some topological ordering of G_W;*

(b) *if G_R contains an edge (v, v') such that both v and v' are values in S, say $v = v_i$ and $v' = v_j$, then $j > i - k$; and*
(c) *if G_R contains an edge (v, v') such that v is the last value in S and v' is not in S, then S contains a value v'' such that G_W contains the edge (v'', v').*

DEFINITION 5.3. *For any (m, k)-configurations C and C' with respect to a read value graph $G_R(V, E_R)$ and write graph $G_W(V, E_W)$, we say that C'* extends *C if and only if C' is obtained from C by removing the first value v and appending a value $v' \neq v$ such that $(v', v) \notin E_W$.*

The high-level pseudocode of CGS is presented as Algorithm 3. First, the graphs G_R and G_W representing the structure of the input history H are computed at lines 1–2. Next, the algorithm checks whether $k \geq n_w$ at line 3, which means that H is automatically k-atomic under Assumptions 3.2–3.4. In this case CGS terminates with a YES output at line 4. Then, the algorithm looks for a value v such that k or more edges of G_R originate at v and terminate at values that precede v in $<_H$. If such a value v exists, then for any topological ordering of G_W, there will be at least one edge from v that reaches back by k or more positions, which implies that H is not k-atomic (see Lemma 3.8). In this case CGS terminates with a NO output at line 6. Bounding the number of such backward edges at this stage reduces the complexity of the next phase of the algorithm (see analysis of line 9 in the proof of Theorem 5.11), which constructs the graph G_C of configurations at lines 9–11 after computing the write concurrency of H earlier at line 8. The cost of executing lines 9–11 dominates the time complexity of CGS as the number of possible configurations grows exponentially with n_w in the worst case. Finally, lines 12–15 decide k-atomicity based on the length of the longest path in G_C. As we show later on in Lemma 5.7, H is k-atomic if and only if G_C is nonempty and contains a path with $|V| - \max(m, k)$ edges.

```
   Input: history H and integer k, 2 ≤ k ≤ n.
   Output: YES if H is k-atomic, NO otherwise.
 1 G_R(V, E_R) := read value graph for H
 2 G_W(V, E_W) := write graph for H
 3 if |V| ≤ k then
 4 |   return YES
 5 else if ∃v ∈ V s.t. |{v' ∈ V | (v', v) ∈ E_W ∧ (v, v') ∈ E_R}| ≥ k then
 6 |   return NO
 7 else
 8 |   m := write concurrency of H
 9 |   C := set of (m, k)-configurations with respect to G_R and G_W
10 |   E_C := {(c, c') ⊆ C × C | configuration c' extends configuration c}
11 |   G_C := configuration graph (C, E_C)
12 |   if G_C is nonempty and contains a path with |V| − max(m, k) edges then
13 |   |   return YES
14 |   else
15 |   |   return NO
16 |   end
17 end
```

Algorithm 3: The CGS algorithm.

5.2. Examples. With respect to the example history presented in Figure 3.1, CGS works correctly for all $k \geq 1$. The write concurrency for this particular input is $m = 5$ because $w(5)$ overlaps the four other writes as well as itself. Recall from section 3 that the input history is 3-atomic but not 1-atomic or 2-atomic. For $k = 1$, the condition at line 5 holds with $v = 1$ and $v' = 2$, and so the algorithm correctly returns NO at line 6. For $k = 2$, the condition at line 5 is false and so the algorithm proceeds to build the set $\mathcal{C}$ of $(5, 2)$-configurations at line 9. According to Definition 5.2 there are no possible $(5, 2)$-configurations because it is impossible to satisfy clause (b). As a result, the configuration graph G_C computed at line 11 is empty. Thus, the condition at line 12 is false and CGS correctly returns NO at line 15. For $k = 3$, the condition at line 5 is false and so the algorithm proceeds to build the set $\mathcal{C}$ of $(5, 3)$-configurations at line 9. According to Definition 5.2 the only possible $(5, 3)$-configurations are $\langle 5, 2, 1, 3, 4\rangle$ and $\langle 5, 2, 3, 1, 4\rangle$, which are the two possible 3-atomic value orders for H. The configuration graph G_C computed at line 11 is therefore nonempty and contains a path with $|V| - \max(m, k) = 5 - 5 = 0$ edges. Thus, the condition at line 12 holds and CGS correctly returns YES at line 13.

Finally, consider the example history presented in Figure 3.1 with $w(5)$ removed. In that case the write concurrency becomes $m = 3$, and the input history remains 3-atomic but not 2-atomic, as explained in section 4.2. The corresponding write graph G_W has three topological orderings, as shown in Table 5.1. The possible configurations for $k = 2$ and $k = 3$ are then enumerated in Table 5.2, and the corresponding configuration graphs are presented in Figure 5.1.

TABLE 5.1
Exhaustive list of topological orderings for G_W corresponding to the history from Figure 3.1 *with $w(5)$ removed.*

Topological ordering	Is 2-atomic	Is 3-atomic
$\langle 2, 1, 3, 4\rangle$	no because $w(3) <_H r(2)$	yes
$\langle 2, 1, 4, 3\rangle$	no because $w(3) <_H r(2)$	no because $w(3) <_H r(2)$
$\langle 2, 3, 1, 4\rangle$	no because $w(1) <_H r(2)$	yes

TABLE 5.2
Exhaustive list of configurations for $k = 2$ and $k = 3$ corresponding to the history from Figure 3.1 *with $w(5)$ removed.*

Subsequence of topological ordering	Is a $(3, 2)$-configuration	Is a $(3, 3)$-configuration
$\langle 1, 3, 4\rangle$	yes	yes
$\langle 2, 1, 4\rangle$	yes	yes
$\langle 3, 1, 4\rangle$	yes	yes
$\langle 2, 1, 3\rangle$	no due to Definition 5.2 (b) with $v_i = 3$ and $v_j = 2$	yes
$\langle 2, 3, 1\rangle$	no due to Definition 5.2 (b) with $v_i = 1$ and $v_j = 2$	yes
$\langle 1, 4, 3\rangle$	no due to Definition 5.2 (c) with $v = 3$ and $v' = 2$	no due to Definition 5.2 (c) with $v = 3$ and $v' = 2$

For $k = 2$, the condition at line 5 is false and so the algorithm proceeds to build the set $\mathcal{C}$ of $(3, 2)$-configurations at line 9. As shown in Table 5.2, the $(3, 2)$-configurations in this case are $\langle 1, 3, 4\rangle$, $\langle 2, 1, 4\rangle$, and $\langle 3, 1, 4\rangle$. Since none of these $(3, 2)$-configurations extends any other, the configuration graph G_C computed at line 11 has three vertices but no edges and hence does not contain a path with $|V| - \max(3, 2) = 4 - 3 = 2$ edges.

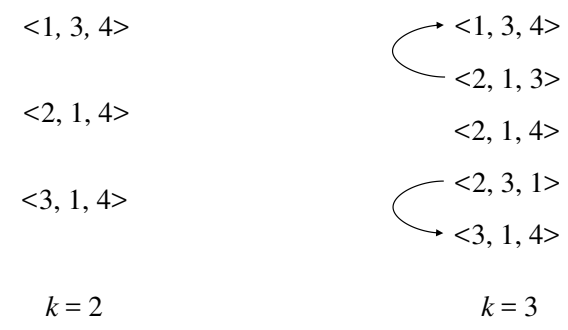

FIG. 5.1. *Configuration graphs for $k = 2$ (left) and $k = 3$ (right) corresponding to the history from Figure* 3.1 *with $w(5)$ removed.*

Thus, the condition at line 12 is false and CGS correctly returns NO at line 15. For $k = 3$, the condition at line 5 is again false and the possible $(3, 3)$-configurations are $\langle 1, 3, 4\rangle$, $\langle 2, 1, 3\rangle$, $\langle 2, 1, 4\rangle$, $\langle 2, 3, 1\rangle$, and $\langle 3, 1, 4\rangle$. Since $\langle 1, 3, 4\rangle$ extends $\langle 2, 1, 3\rangle$, and $\langle 3, 1, 4\rangle$ extends $\langle 2, 3, 1\rangle$, the graph G_C computed at line 11 is nonempty and contains two paths with $|V| - \max(3, 3) = 4 - 3 = 1$ edges. These two paths induce the two possible k-atomic value orders for H: $\langle 2, 1, 3, 4\rangle$ and $\langle 2, 3, 1, 4\rangle$. Thus, the condition at line 12 holds and CGS correctly returns YES at line 13.

5.3. Analysis. In this section we suppose that Assumptions 3.2–3.4 hold and we show that the algorithm returns YES if and only if the input history H is k-atomic. In our analysis any references to the *configuration graph* G_C pertain to the graph structure computed at line 11 of the algorithm.

DEFINITION 5.4. *Let $P = \langle C_1, C_2, C_3, \ldots, C_\ell\rangle$ be a finite path in the configuration graph G_C. The* sequence S of values induced by P *is defined recursively as follows: if $\ell = 1$, then $S = C_1$, as otherwise S is the sequence of values induced by $\langle C_1, C_2, C_3, \ldots, C_{\ell-1}\rangle$ followed by the last value of C_ℓ.*

LEMMA 5.5. *For any finite path $P = \langle C_1, C_2, C_3, \ldots, C_\ell\rangle$ in the configuration graph G_C, the sequence S of values induced by P is a subsequence of some topological ordering of G_W.*

Proof. To show that S is a subsequence of some topological ordering of G_W, we must prove two properties: (i) S is a sequence of distinct values; and (ii) if v_i precedes v_j in S, then $v_j \not<_H v_i$. The proof proceeds by induction on ℓ. The base case $\ell = 1$ follows directly from Definition 5.2 (a) since each vertex in G_C is an (m, k)-configuration. Now suppose that the lemma holds for some $\ell \geq 1$ and consider a path $P_{\ell+1}$ of length $\ell + 1$. Let P_ℓ denote the prefix of $P_{\ell+1}$ of length ℓ. Let S_ℓ and $S_{\ell+1}$ denote the sequences of values induced by P_ℓ and $P_{\ell+1}$, respectively. Let C_ℓ and $C_{\ell+1}$ denote the last (m, k)-configurations in P_ℓ and $P_{\ell+1}$, respectively. Let v_i denote the ith element of $S_{\ell+1}$ and note that v_ℓ is the first element of C_ℓ. A counterexample to the lemma comprises a pair of values v_i and v_j such that $i < j$, v_j is the last value of both $S_{\ell+1}$ and $C_{\ell+1}$, and $v_i = v_j$ if property (i) is violated, or $v_j <_H v_i$ if property (ii) is violated. To derive a contradiction, suppose that such a pair v_i, v_j exists in $S_{\ell+1}$. It follows from the base case (applied to a path comprising $C_{\ell+1}$ only) that the values v_i and v_j cannot both occur in $C_{\ell+1}$ with v_i preceding v_j. Furthermore, it follows from the induction hypothesis that the values v_i and v_j cannot both occur in S_ℓ with

v_i preceding v_j. Thus, value v_i occurs in $S_{\ell+1}$ only before the positions corresponding to the initial $\max(m,k)-1$ elements of $C_{\ell+1}$, which constitute a suffix of S_ℓ, whereas v_j occurs in $C_{\ell+1}$ in the last position. Since $C_{\ell+1}$ extends C_ℓ, it also follows from Definition 5.3 that $v_j \neq v_\ell$ and that $v_j \not<_H v_\ell$, which implies $v_i \neq v_\ell$ regardless of whether property (i) or property (ii) is violated. In other words, v_i occurs in $S_{\ell+1}$ only before the positions corresponding to all $\max(m,k)$ elements of C_ℓ.

Next, we focus on C_ℓ, which begins with v_ℓ. Since C_ℓ is an (m,k)-configuration, it contains at least m values. Furthermore, since we showed that value v_i occurs only before the elements of C_ℓ in $S_{\ell+1}$, v_i is different from all the values in C_ℓ, and so it follows from Definition 5.1 that one of these m or more values, say v_a, is adjacent to v_i in G_W. Since v_i appears in S_ℓ before the elements of C_ℓ, hence before v_a, it follows from the induction hypothesis that $v_a \not<_H v_i$ as otherwise S_ℓ would not be a subsequence of some topological ordering. Thus, v_a being adjacent to v_i in G_W specifically implies $v_i <_H v_a$.

To complete the proof, we will show that $v_j <_H v_a$ and that v_a precedes v_j in $C_{\ell+1}$, and then derive a final contradiction. First, we focus on the relative order of v_j, which is the last element of $C_{\ell+1}$, and v_a in $<_H$. If v_i and v_j violate property (i), then $v_i = v_j$, and so $v_i <_H v_a$ implies $v_j <_H v_a$, as needed. If v_i and v_j violate property (ii), then $v_j <_H v_i$, and so $v_i <_H v_a$ once again implies $v_j <_H v_a$. Next, consider the position of v_a in $S_{\ell+1}$ relative to the elements of $C_{\ell+1}$. Since $C_{\ell+1}$ extends C_ℓ and v_ℓ is defined as the first element of C_ℓ, it follows from Definition 5.3 that $v_j \not<_H v_\ell$ because v_j is the last element of $C_{\ell+1}$, and so $v_j <_H v_a$ implies $v_a \neq v_\ell$. Since v_a is also one of the values in C_ℓ, $v_a \neq v_\ell$ implies that v_a is not the first element but one of the last $\max(m,k)-1$ elements of C_ℓ, and hence one of the first $\max(m,k)-1$ elements of $C_{\ell+1}$, which extends C_ℓ. Thus, $v_j <_H v_a$ and v_a precedes v_j in $C_{\ell+1}$, which contradicts Definition 5.2 (a) since $C_{\ell+1}$ is an (m,k)-configuration. ∎

COROLLARY 5.6. *Suppose that H contains $n_w \geq k$ writes. Then G_C is acyclic and any path P through the configuration graph G_C has length (number of edges) at most $n_w - \max(m,k)$.*

Proof. By Lemma 5.5, any finite path through G_C induces a sequence S of values that is a subsequence of some topological ordering of G_W. As a result, S has at most n_w values, all of which are distinct. The first (m,k)-configuration in P contributes $\max(m,k)$ of these, and subsequent configurations contribute one value each. Therefore, if P has L edges, then S has $L + \max(m,k)$ values, and $L + \max(m,k) \leq n_w$ implies $L \leq n_w - \max(m,k)$. Moreover, G_C is acyclic because every path in the graph is finite. ∎

LEMMA 5.7. *Suppose that H contains $n_w \geq k$ writes. Then H is k-atomic if and only if the configuration graph G_C is nonempty and has a path with exactly $n_w - \max(m,k)$ edges.*

Proof. Suppose that H is k-atomic, and note that $n_w - \max(m,k) \geq 0$ since we assume $n_w \geq k$ and since $n_w \geq m$ holds in general. We will show that G_C contains a path P with $n_w - \max(m,k)$ edges and $n_w - \max(m,k)+1$ vertices, which induces a sequence of values of length n_w. Since H is k-atomic, we can apply Lemma 3.8 to obtain a topological ordering $T = \langle v_1, v_2, \ldots, v_{n_w} \rangle$ of G_W such that edges of G_R reach backward by at most $k-1$ positions. It suffices to show that for every ℓ such that $1 \leq \ell \leq n_w - \max(m,k)+1$, the tuple $C_\ell = \langle v_\ell, v_{\ell+1}, \ldots, v_{\ell+\max(m,k)-1} \rangle$ is an (m,k)-configuration, in which case the desired path P is the one that traverses $C_1, C_2, \ldots, C_{n_w-\max(m,k)+1}$. Consider an arbitrary ℓ, $1 \leq \ell \leq n_w - \max(m,k)+1$, and

for simplicity denote the values in C_ℓ by $\langle w_1, w_2, \ldots, w_{\max(m,k)} \rangle$. For property (a) of Definition 5.2, C_ℓ is a contiguous subsequence of T, which is a topological ordering of G_W. It further follows from Lemma 3.8 that if G_R contains an edge $(w_i, w_j) = (v_{\ell+i-1}, v_{\ell+j-1})$, then $\ell + j - 1 > \ell + i - 1 - k$ and so $j > i - k$. Thus, C_ℓ satisfies property (b) of Definition 5.2. For property (c) suppose that $(v_i, v_j) \in E_R$ where $v_i = w_{\max(m,k)}$ is the last element of C_ℓ and v_j is not in C_ℓ at all. Then from the properties of T obtained by Lemma 3.8 we observe that $j > i - k$. Since v_j is not in C_ℓ, this implies that $j > i$, and so v_j succeeds v_i in T. Since C_ℓ contains at least m values, v_j must be adjacent to at least one of the values of C_ℓ, say w_a, in G_W. Thus, either $(w_a, v_j) \in E_W$ or $(v_j, w_a) \in E_W$. Since T is a topological ordering of G_W where v_j appears after all values of C_ℓ, it follows that $(w_a, v_j) \in E_W$, as needed.

Conversely suppose that G_C has a path P with $n_w - \max(m, k)$ edges. Then P induces a sequence $S = \langle v_1, v_2, \ldots, v_{n_w} \rangle$ of values, which are distinct because G_C is acyclic by Corollary 5.6. Thus, S has exactly n_w values, and so Lemma 5.5 implies that S is a topological ordering of G_W. Now suppose that G_R contains an edge (v_i, v_j). It suffices to show that $j > i - k$, in which case H is k-atomic by Lemma 3.8. If P visits an (m, k)-configuration C that contains both v_i and v_j, then $j > i - k$ by Definition 5.2 (b), as needed. Otherwise let C be the (m, k)-configuration visited by P in which v_i appears in the latest possible position (i.e., the first configuration visited by P that contains v_i).

Case 1: v_i is the last value in C. Since v_j is not in C, it follows from Definition 5.2 (c) that C contains a value v_c such that G_W contains the edge (v_c, v_j). Since C is a contiguous subsequence of a topological ordering of G_W and since C does not contain v_j, this implies that $j > i$; hence $j > i - k$ as needed.

Case 2: v_i is not the last value in C. Then C is the first (m, k)-configuration visited by P, and so $i < \max(m, k)$. Since v_j is not in C, it follows that v_j appears in S after the values of C, and so $j > \max(m, k)$. Thus, $j > i > i - k$, as needed. ∎

THEOREM 5.8. *The algorithm returns YES if H is k-atomic and NO otherwise.*

Proof. The return statement at line 4 is reached when $|V| \leq k$. In this case, H contains at most k writes, and so H is k-atomic under Assumption 3.3, and the algorithm correctly returns YES. The return statement at line 6 is reached when the write of some value v is followed in $<_H$ by the reads of at least k other values whose dictating writes happen before $w(v)$. In this case G_R contains edges from v to at least k other vertices that must precede v in any topological ordering of G_W, and so Lemma 3.8 implies that H is not k-atomic. Thus, the algorithm correctly returns NO at line 6. Otherwise, $|V| > k$ and the algorithm returns at line 13 or line 15. Since H contains at least k writes in this case, Lemma 5.7 states that H is k-atomic if and only if G_C is nonempty and has a path with $n_w - \max(m, k)$ edges. Indeed, the algorithm returns YES at line 13 if this condition holds and NO at line 15 otherwise. ∎

In the remainder of this section, we analyze the running time of the algorithm with the help of two lemmas that bound the size of the graph of configurations.

LEMMA 5.9. *Let $C = \langle v_1, v_2, \ldots, v_\ell \rangle$ be an (m, k)-configuration for a history H. If v_i is fixed for some $1 \leq i < \ell$, then there are at most $2m - 1$ possible choices for v_{i+1}: at most $m - 1$ where $w(v_i)$ is concurrent with $w(v_{i+1})$ and at most m where $w(v_i) <_H w(v_{i+1})$.*

Proof. Fix v_i and let W be the subset of possible values for v_{i+1}. Since C must be a contiguous subsequence of some topological ordering of G_W, we can define W as the disjoint union of two subsets: W_1 is the set of values not adjacent with v_i in

G_W (i.e., values v_{i+1} such that $w(v_i)$ is concurrent with $w(v_{i+1})$), and W_2 is a subset of out-neighbors of v_i in G_W (i.e., values v_{i+1} such that $w(v_i) <_H w(v_{i+1})$). Since m is the write concurrency of H, it follows from Definition 5.1 that $|W_1| \leq m-1$ because $w(v_i)$ may overlap with at most $m-1$ writes other than itself. Similarly, $|W_2| \leq m$ as otherwise W_2 contains two values x, y such that G_W contains edges (v_i, x) and (x, y) (i.e., two values whose dictating writes are not concurrent), in which case y cannot be an immediate successor of v_i in a topological ordering of G_W. Thus $|W| = |W_1| + |W_2| \leq 2m-1$, and so there are at most $2m-1$ choices for v_{i+1}. ∎

LEMMA 5.10. *The graph G_C of (m,k)-configurations for a history H with n_w writes has at most $n_w \cdot (2m-1)^{\max(m,k)-1}$ vertices and at most $n_w \cdot (2m-1)^{\max(m,k)}$ edges.*

Proof. First consider the number of vertices. Let $C = \langle v_1, v_2, \ldots, v_{\max(m,k)} \rangle$ denote an arbitrary (m,k)-configuration in G_C. There are at most n_w choices for the first element, v_1. Now suppose that $v_1, \ldots, v_i$ have been decided for some $i < \max(m,k)$, and consider v_{i+1}. By Lemma 5.9 there are at most $2m-1$ possible choices for v_{i+1}. Since there are $\max(m,k)-1$ values in C after v_1, the total number of possible (m,k)-configurations is at most $n_w \cdot (2m-1)^{\max(m,k)-1}$, as needed.

Next consider the edges. Suppose that (m,k)-configurations C and C' are adjacent in G_C, and C' extends C. Then C' contains the last $\max(m,k)-1$ elements of C and one additional element, say v. As explained earlier there are at most $2m-1$ choices for v and so given C there are at most $2m-1$ choices for C'. Thus, the number of edges in G_C is at most a factor of $2m-1$ larger than the number of vertices. ∎

THEOREM 5.11. *The time complexity of the algorithm is $O(n^2 + n \cdot \max(m,k)^2 \cdot (2m-1)^{\max(m,k)-1})$, where n denotes the number of operations in the input history H, and m is the write concurrency of H.*

Proof. The time complexity is contingent on a careful implementation of the algorithm, which we describe in this proof. The analysis is broken down by line number.

Lines 1 and 2: $O(n^2)$ steps. The graphs G_R and G_W are represented using adjacency matrices. The values read and written by operations in H are first remapped to the domain of consecutive integers from 1 to n_w by fixing an order on the values, and then mapping each value to its position in the chosen order. For example, to fix the order, the values can be inserted (without duplicates) into a balanced binary tree sorted in ascending order in $O(n \log n)$ time. An in-order tree traversal then assigns to each distinct value its position in the sorted order, from 1 to n_w, in $O(n)$ time. Finally, the values in the history are replaced with the assigned integers, using an $O(\log n)$ lookup per operation and $O(n \log n)$ steps in total. The remapped values are used to index the rows and columns of the adjacency matrices, which can be populated in $O(n^2)$ steps by scanning H using a pair of nested for loops. In the remainder of the proof we will use the notation $G_W[i,j]$ and $G_R[i,j]$ to denote the matrix elements corresponding to the edge (i,j). The wildcard symbol $\star$ will be used to denote a matrix row or column, for example $G_W[\star, j]$ indicating all values whose dictating writes happen before $w(j)$. Thus, the construction of G_R and G_W requires $O(n^2)$ steps in total.

Line 3: $O(1)$ steps. The condition $|V| \leq k$ at line 3 can be evaluated in $O(1)$ steps since $|V| = n_w$ is known from the construction of the adjacency matrices.

Line 5: $O(n_w^2)$ steps. For each value v, the condition at line 5 can be tested in $O(n_w)$ steps as follows: determine $\{v' \in V \mid (v', v) \in E_W\}$ using $G_W[\star, v]$, determine

$\{v' \in V \mid (v, v') \in E_R\}$ using $G_R[v, \star]$, and take the intersection. Since column v of G_W and row v of G_R are n_w-element binary vectors, the intersection can be computed in $O(n_w)$ steps using a dot product. The sum of the elements in this dot product is the size of the set referred to by line 5.

Line 8: $O(n_w^2)$ steps. For each value v, the number of writes overlapping with $w(v)$ can be determined in $O(n_w)$ steps by using $G_W[\star, v]$ and $G_W[v, \star]$ to first identify nonoverlapping writes.

Line 9: $O(n_w^2 + n_w \cdot \max(m,k)^2 \cdot (2m-1)^{\max(m,k)-1})$ steps. The set of all (m,k)-configurations can be determined by enumerating candidate permutations of values and testing each permutation against Definition 5.2. By Lemma 5.10 the total number of such permutations is $O(n_w \cdot (2m-1)^{\max(m,k)-1})$ as there are at most n_w choices for the first element, and at most $2m-1$ choices for each subsequent element. To identify the candidate permutations efficiently we first compute for each value v the set W of at most $2m-1$ values that may follow v in a topological ordering of G_W, which is denoted $Succ(v)$ in the remainder of the proof. It follows from Lemma 5.9 that $Succ(v)$ is the union of the set W_1 of values whose writes are concurrent with $w(v)$, and a subset W_2 of values whose writes happen after $w(v)$. To compute W_1 it suffices to compare v against every other value. Set W_2 is computed similarly, except that we must also determine the minimum finish time t of the writes that happen after $w(v)$ and then prune any writes that start after time t. Precomputing W_1 and W_2 for each vertex takes $O(n_w^2)$ steps in total, and the additional cost of enumerating candidate permutations of values is $O(n_w \cdot (2m-1)^{\max(m,k)-1})$.

As the algorithm enumerates candidate permutations of $\max(m,k)$ values, each one is tested against the three properties of Definition 5.2 to determine whether it constitutes an (m,k)-configuration. Property (a) can be tested in $O(\max(m,k)^2)$ steps by considering all possible pairs of values v, v' in a permutation and checking their relative order against the adjacency matrix for G_W. Similarly, property (b) can be tested in $O(\max(m,k)^2)$ steps by considering pairs of values v, v' that are separated by k or more positions and checking their relative order against G_R. For property (c) we must consider edges $(v, v') \in E_R$ such that v is the last value in the permutation and v' is not in the permutation at all. Such values v' comprise three disjoint categories: (i) $(v, v') \in E_W$, (ii) $w(v)$ is concurrent with $w(v')$, and (iii) $(v', v) \in E_W$. A value in category (i) satisfies Definition 5.2 (c) automatically, and so only values in categories (ii) and (iii) need to be tested explicitly against property (c). The subsets of values in categories (ii) and (iii) can be precomputed in $O(n_w)$ steps for a given v using the adjacency matrix rows $G_R[v, \star]$ and $G_W[v, \star]$. The sizes of these subsets are bounded by m and k as follows. By definition of the write concurrency parameter m (see Definition 5.1), there can be at most $m-1$ choices for v' in category (ii). By the negation of the condition evaluated at line 5, which holds as otherwise line 9 would not be reached, there can be at most $k-1$ choices for v' in category (iii) since $(v, v') \in E_R$. Thus, given v there are at most $m+k-2$ choices for v' in total outside of category (i), and for each choice of v' we must decide whether an edge (v'', v') exists in E_W such that v'' is a value in the candidate permutation. Such an edge exists if and only if E_W contains an edge (v''', v'), where v''' is the value in the candidate permutation whose dictating write has the minimum finish time (possibly $v''' = v''$), and so it suffices to test for (v''', v') specifically. The value v''' can be computed once for each candidate permutation in $O(\max(m,k))$ steps. The cost of checking property (c) is therefore $O(\max(m,k))$ steps per candidate permutation with a one-time precomputation cost of $O(n_w^2)$ to determine categories (i)–(iii) for each value v. Thus, the total cost of identifying all the (m,k)-configurations is in

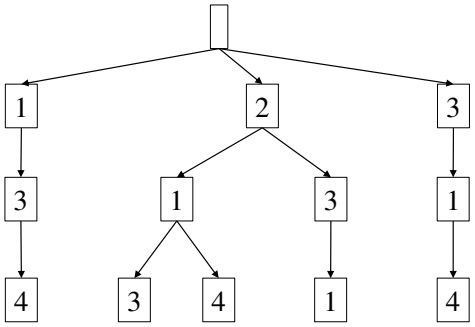

FIG. 5.2. *Trie structure corresponding to the set of* $(3,3)$*-configurations shown in Figure* 5.1.

$O(n_w^2 + n_w \cdot \max(m,k)^2 \cdot (2m-1)^{\max(m,k)-1})$.

The candidate permutations can be stored efficiently using a trie data structure [12]. An example of such a structure for the set of $(3,3)$-configurations presented in Figure 5.1 is illustrated in Figure 5.2. The root node has a branching factor of n_w and its child pointers are stored in an array indexed directly by the value of the first element in the permutation. It follows from Lemma 5.9 that the branching factor is at most $2m-1$ at lower levels of the trie, which can be much less than n_w, and so we use a sorted array instead to store child pointers in each node. The array contains pairs comprising the value of the next element in the permutation and the corresponding child pointer, and these pairs are sorted by the value so that following a child pointer takes $O(\log m)$ steps. Fundamental trie operations (i.e., insertions and lookups) take $O(\max(m,k) \cdot \log m)$ steps but the total cost of building the trie is linear in the number of (m,k)-configurations if they are inserted in ascending lexicographical order because the amortized cost of insertion in that case is $O(1)$. This is accomplished by enumerating the (m,k)-configurations in ascending order, which does not increase the time complexity asymptotically. Similarly, the cost of an in-order traversal of the trie is also linear in the number of elements.

Line 10*:* $O(n_w \cdot \max(m,k) \cdot \log m \cdot (2m-1)^{\max(m,k)-1})$ *steps.* The edges of the configuration graph G_C can be computed by traversing the trie to visit each (m,k)-configuration C, and finding every (m,k)-configuration C' that extends C according to Definition 5.3. A linear-time in-order traversal of the trie takes $O(n_w \cdot (2m-1)^{\max(m,k)-1})$ steps since that is the number of (m,k)-configurations by Lemma 5.10. It follows from Lemma 5.9 that for each (m,k)-configuration C there are at most $2m-1$ choices for the last value in C', which can be identified in $O(m)$ steps using the precomputed structure *Succ* described in the analysis of line 9. The adjacency matrix for G_W must also be tested to rule out the presence of an edge from the last value in C' to the first value in C. Finally, for each C' the algorithm must perform a lookup in the trie structure to determine if C' is indeed an (m,k)-configuration. Ordinarily each lookup takes $O(\max(m,k) \cdot \log m)$ steps, but in this case an optimization is possible since all configurations C' that extend the same C share the same values in all but the last position. After the first lookup for a given C, the remaining lookups can start at the level immediately above the leaf nodes instead of starting at the root, which takes $O(m)$ additional steps since there are at most $2m-1$ possibilities for C'. Thus, the cost of the trie accesses is $O(\max(m,k) \cdot \log m)$ for each C, and in total $O(n_w \cdot \max(m,k) \cdot \log m \cdot (2m-1)^{\max(m,k)-1})$ steps for all the (m,k)-configurations.

Each edge (C, C') of the configuration graph can be represented efficiently as a pair of pointers to the corresponding leaf nodes in the trie structure, which is sufficient for our purposes since we are interested in the length of paths through G_C. Such an adjacency list can be constructed in $O(1)$ steps per edge.

Line 12: $O(n_w \cdot (2m-1)^{\max(m,k)})$ steps. Since G_C is acyclic by Corollary 5.6, the longest path computation can be carried out efficiently [32]. First, depth-first search is used to compute a topological ordering of G_C. The length of the longest path ending at vertex v, denoted $L(v)$, is then determined by processing vertices in the computed order. Letting $I(v)$ denote the set of in-neighbors of v, $L(v) = 0$ if $I(v) = \emptyset$ and $L(v) = 1 + \max_{w \in I(v)} L(w)$ otherwise. Finally, the longest path length is determined by taking the maximum over the computed values of $L(v)$. Assuming an adjacency list representation, the total cost is linear in the size (number of edges) of the configuration graph, which is bounded in Lemma 5.10. The adjacency list can be computed at line 10, which enumerates the edges of G_C, without increasing asymptotic time complexity.

Total. The time complexity of the entire algorithm, implemented as described in this proof, is

$$O\left(n^2 + n_w^2 + n_w \cdot \max(m,k)^2 \cdot (2m-1)^{\max(m,k)-1} + n_w \cdot (2m-1)^{\max(m,k)}\right),$$

which can be rewritten as $O(n^2 + n_w^2 + n_w \cdot \max(m,k)^2 \cdot (2m-1)^{\max(m,k)-1})$. This implies the theorem since $n_w \leq n$. ∎

The time complexity of algorithm CGS, as stated in Theorem 5.11, is of the form $O(n^2 + f(m,k)n)$, where k is a constant independent of n, and $m \leq n$ is a parameter that depends on the input history but not on n directly. Therefore, Theorem 5.11 establishes that k-AV is fixed-parameter tractable under Assumptions 3.2–3.4 with respect to the combination of m and k.

6. Practical applicability. In this section we investigate the time complexity of algorithm CGS (section 5) in practice by computing the parameters n (number of operations) and m (write concurrency; see Definition 5.1) from experimental data. The data sets are borrowed from two recent experimental studies [19, 26].

Experimental environment. The suite of experiments presented in [19] was conducted using Cassandra 1.2.4 [22] as the storage system and the Yahoo Cloud Serving Benchmark (YCSB) 0.1.4 [6] as the workload generator. The hardware infrastructure comprised ten 64-bit 2.2 GHz dual-core AMD Opteron servers equipped with 8 GB DRAM, 7200 RPM SATA disks, and Gigabit Ethernet. Five machines were used to run the storage system and another five for the benchmark, all in the same private data center. The software environment included CentOS 5.5 Linux and OpenJDK 1.7.0_19. Clock skew across servers was estimated using the `ntpq` command as less than 10ms.

The more recent experiments in [26] were conducted using a newer version of Cassandra (2.0.10) and the same YCSB release. The hardware infrastructure was provisioned in Amazon's Elastic Compute Cloud (EC2). Three virtual machines were used, each equipped with a 64-bit 2.5 GHz quad-core Intel Xeon processor, 16 GB DRAM, and 2×40 GB SSD local storage. The software environment included Ubuntu 14.04 Linux and Oracle Java 1.7.0_72. In one set of experiments, all three virtual machines were deployed in the us-west-2 data center, and clocks were synchronized to within 0.05ms. In the second set of experiments, the three virtual machines were distributed across multiple data centers: us-west-1, us-east-1, and sa-east-1. Clocks

were synchronized to within 10ms.

Translation of execution histories into k-AV inputs. The data set comprises execution histories from 11 experiments, where each experiment includes multiple (3-9) runs with different combinations of storage system settings and workload parameters. In each run, the benchmark (YCSB) applied operations to the storage system (Cassandra) for at least 60 seconds, usually at a throughput level of approximately 1,000 operations per second per server. The start and finish times of operations were measured using a microsecond precision clock, leading to the possibility that the start and finish times of operations are not distinct, in contrast to the assumption stated in section 3. We adopt the convention that if an operation A ends at the same time as some other operation B begins, then A happens before B for the purpose of defining the graphs G_W and G_R representing the history. This convention is justified by the observation that the start and finish times of operations are measured at clients, whereas the actual execution of the operation at servers begins slightly later and finishes slightly earlier due to communication delays between the client and storage system back end.

Although the history generated by each of the experimental runs could be used directly as an instance of k-AV, each object accessed in the history can be analyzed separately thanks to the locality property of k-atomicity, as explained in section 2. Therefore, we first project the history obtained from a given run into a set of subhistories corresponding to distinct objects. Furthermore, each of these subhistories can often be decomposed into smaller fragments that can be analyzed independently—a type of divide-and-conquer strategy introduced in connection with the FZF algorithm for solving 2-AV [16]. To a first approximation, the decomposition can be achieved as follows: if the operations in a history H can be partitioned into sets S_1, S_2 such that the set of values accessed by operations in S_1 is disjoint from the set of values accessed by operations in S_2, then the projections H_1, H_2 of H corresponding to S_1, S_2 can be analyzed independently provided that $\max_{op_1 \in S_1} f(op_1) < \min_{op_2 \in S_2} s(op_2)$. In that case H is k-atomic if and only if both H_1 and H_2 are k-atomic. The projections H_1 and H_2 are decomposed recursively, if possible, to further reduce the size of the problem instances. In some cases, H_1 and H_2 can be analyzed independently even when $\max_{op_1 \in S_1} f(op_1) \geq \min_{op_2 \in S_2} s(op_2)$, leading to a more fine-grained decomposition. Before describing the details, we first review some terminology introduced by Gibbons and Korach [14].

A collection of operations that access the same value v with respect to the same key κ is called a *cluster*. The invocation and response times of the operations in a cluster for a particular v and κ define, roughly speaking, an interval of time during which v appears to be the current value of κ. More formally, a cluster has an associated *zone*, which is defined as the time interval from the minimum finish time of any operation in the cluster to the maximum start time of any operation in the cluster. A *forward zone* occurs when the minimum finish time is less than the maximum start time, meaning that at least one dictated read starts after some other operation in the zone finishes. A *backward zone* occurs when the minimum finish time is greater than or equal to the maximum start time, meaning that all operations in the zone overlap at a common point in time. For completeness we include in Figure 6.1 a copy of Figure 3 from [16], which illustrates a possible combination of zones in one history. In this figure *FZx* and *BZx* indicate forward and backward zones, respectively, and time increases from left to right.

The fine-grained decomposition procedure, which was introduced in [16], groups zones into maximal subsets called *chunks* such that (i) two forward zones are in the

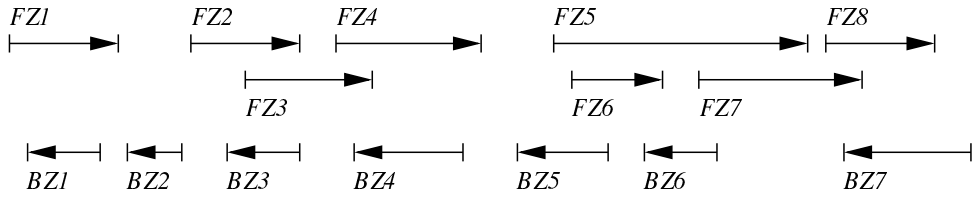

FIG. 6.1. *Example of zones in a history.* ©[2013] *IEEE. Reprinted, with permission, from* [16].

same chunk if they intersect, and (ii) a backward zone belongs to a chunk if its time interval is contained entirely in the union of the time intervals of the forward zones in that chunk. As an example, in Figure 6.1 there are three chunks: $\{FZ_1, BZ_1\}$, $\{FZ_2, FZ_3, FZ_4, BZ_3, BZ_4\}$, and $\{FZ_5, FZ_6, FZ_7, FZ_8, BZ_6\}$. Intuitively, the remaining zones BZ_2, BZ_5, and BZ_7 can be ignored in the context of k-AV because their operations can be linearized easily in between chunks. It is straightforward to show that under Assumptions 3.2–3.3 an execution history is k-atomic if and only if each chunk of that history is k-atomic.

Experimental results. We applied the above translation procedure to the 89 execution histories in the data sets under consideration, which generated more than eight million chunks. For each history we computed the total number of chunks, total number of zones, total number of operations, maximum number of operations per chunk, maximum write concurrency, number of chunks with $m \leq 5$, number of chunks that satisfy Assumption 4.1, and number of chunks where $m > 5$ and Assumption 4.1 is not satisfied. The chunks mostly comprised 1–200 operations and 1–10 zones. Thus, the problem instances were fairly small but not small enough to permit solving k-AV using a brute force approach. The number of chunks satisfying Assumption 4.1, hence computable efficiently using the GPO algorithm from section 4, was approximately 96% in the experiments conducted using multiple data centers, and more than 99% in the single data center experiments. Over 99% of the chunks exhibited $m \leq 5$, making them good candidates for the CGS algorithm from section 5 for small values of k. The remaining chunks, where $m > 5$ and Assumption 4.1 does not hold, accounted for fewer than 0.1% of the inputs, and may be too complex for our CGS algorithm.

In addition to analyzing the structure of the chunks, we implemented the GPO and CGS algorithms in Java and executed them on the inputs under consideration. CGS was invoked only on chunks that failed to satisfy Assumption 4.1 and was limited to one second of computation on a laptop equipped with a 2.39 GHz Intel i7-4500 dual-core processor and 8 GB DRAM. More than 99.98% of the inputs were solved successfully, revealing that k was generally less than 100. More than 99% of the inputs exhibited $k < 10$, and more than 90% were 1-atomic. Inputs with $k > 2$, which could not be analyzed using prior 1-AV and 2-AV algorithms [14, 16], were found mostly in the histories obtained from [26]. In those experiments, $k > 2$ occurred in more than 15% of the nonatomic (i.e., $k > 1$) inputs.

7. Conclusion and discussion. In this paper, we have presented two algorithms that solve the k-AV problem for arbitrary $k \geq 2$ in special cases. Our algorithms assume that each read has exactly one dictating write (Assumption 3.2), which circumvents NP-completeness when $k \leq 2$. The first algorithm (GPO) places an additional restriction on the structure of the execution history but always runs in polynomial time. The second algorithm (CGS) does not place any additional restrictions on the input but its running time is polynomial only if both k and our measure

of write concurrency (Definition 5.1) are bounded by constants. The time complexity of k-AV under Assumption 3.2, with no additional restrictions, remains an open question. Our contributions also leave open the possibility of reducing k-AV to the graph bandwidth (GBW) problem, which would settle the time complexity of k-AV, as well as the possibility of reducing GBW to k-AV.

Our algorithms enable precise measurement of version-based staleness, which in turn has several applications: (i) analyzing and understanding the behavior of eventually consistent storage systems (e.g., [8, 22, 33]); (ii) validating mathematical models of consistency (e.g., [3]); as well as (iii) verifying that a storage system or consistency tuning framework (e.g., [2, 34, 39]) delivers a promised level of consistency, and quantifying the severity of any consistency violations observed.

Measurement of version-based staleness entails computing the k-value of a history H, denoted k_H, which is the smallest integer $k \geq 1$ for which H is k-atomic. This value can be computed by a brute force method where k-AV is solved for consecutive values of k starting at $k = 1$, or by executing a binary search over $1 \leq k \leq n_w$ where n_w is the number of writes in H. When the time complexity of the k-AV solver grows exponentially with k, such as in our CGS algorithm, the overall running time is dominated by the highest k tested, and so the brute force approach is preferred as it only considers $k \leq k_H$. In other cases the computation may benefit from binary search with the overall time complexity bounded by $O(\log n_w)$ times the cost of solving k-AV for $k = n_w$.

Given that our k-AV algorithms do not guarantee polynomial running time without additional restrictions, their practical value depends on how often these restrictions hold in real data sets. To shed light on this point, we analyzed the execution histories from [19, 26] to determine representative values of n (number of operations) and m (write concurrency; Definition 5.1). The results suggest that in practice the inputs are typically small, comprising at most a few hundred operations, and the vast majority of k-AV instances satisfy Assumption 4.1, which makes GPO applicable. In most of the remaining inputs, m is small ($\leq$5), and CGS is useful for small k (e.g., $k \leq m$). A minute fraction ($<$0.1% in our analysis) of inputs fail to satisfy both Assumption 4.1 and $m \leq 5$ and may be too complex for CGS.

Acknowledgments. We are grateful to the anonymous referees for their careful proofreading of this work and their insightful suggestions. Xiaozhou (Steve) Li conducted part of this research at Hewlett Packard Labs.

REFERENCES

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

[2] M. S. Ardekani and D. B. Terry, *A self-configurable geo-replicated cloud storage system*, in Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Broomfield, CO, 2014, pp. 367–381.

[3] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, *Probabilistically bounded staleness for practical partial quorums*, PVLDB, 5 (2012), pp. 776–787.

[4] D. Bermbach and S. Tai, *Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior*, in Proceedings of the 6th Workshop on Middleware for Service Oriented Computing (MW4SOC), Lisbon, Portugal, 2011, 1.

[5] E. A. Brewer, *Towards robust distributed systems (abstract)*, in Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC), ACM, New York, 2000, p. 7.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, *Benchmarking cloud serving systems with YCSB*, in Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC), ACM, New York, 2010, pp. 143–154.

[7] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh, *Parameterized Algorithms*, Springer, New York, 2015.

[8] 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), ACM, New York, 2007, pp. 205–220.

[9] R. G. Downey and M. R. Fellows, *Parameterized Complexity*, Springer-Verlag, New York, 1997.

[10] S. Even, A. Itai, and A. Shamir, *On the complexity of timetable and multicommodity flow problems*, SIAM J. Comput., 5 (1976), pp. 691–703, https://doi.org/10.1137/0205048.

[11] J. Flum and M. Grohe, *Parameterized Complexity Theory*, Springer-Verlag, Berlin, 2006.

[12] E. Fredkin, *Trie memory*, Comm. ACM, 3 (1960), pp. 490–499.

[13] M. R. Garey, R. L. Graham, D. S. Johnson, and D. E. Knuth, *Complexity results for bandwidth minimization*, SIAM J. Appl. Math., 34 (1978), pp. 477–495, https://doi.org/10.1137/0134037.

[14] P. B. Gibbons and E. Korach, *Testing shared memories*, SIAM J. Comput., 26 (1997), pp. 1208–1244, https://doi.org/10.1137/S0097539794279614.

[15] S. Gilbert and N. A. Lynch, *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*, ACM SIGACT News, 33 (2002), pp. 51–59.

[16] W. Golab, J. Hurwitz, and X. Li, *On the k-atomicity-verification problem*, in Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS), IEEE, Washington, DC, 2013, pp. 591–600.

[17] W. Golab, X. Li, and M. A. Shah, *Analyzing consistency properties for fun and profit*, in Proceedings of the 30th ACM Symposium on Principles of Distributed Computing (PODC), ACM, New York, 2011, pp. 197–206.

[18] W. Golab, X. Li, A. López-Ortiz, and N. Nishimura, *Computing weak consistency in polynomial time (extended abstract)*, in Proceedings of the 34th ACM Symposium on Principles of Distributed Computing (PODC), ACM, New York, 2015, pp. 395–404.

[19] W. Golab, M. R. Rahman, A. AuYoung, K. Keeton, and I. Gupta, *Client-centric benchmarking of eventual consistency for cloud storage systems*, in Proceedings of the 34th International Conference on Distributed Computing Systems (ICDCS), IEEE, Washington, DC, 2014, pp. 493–502.

[20] M. Herlihy and J. M. Wing, *Linearizability: A correctness condition for concurrent objects*, ACM Trans. Program. Lang. Syst., 12 (1990), pp. 463–492.

[21] D. J. Kleitman and R. V. Vohra, *Computing the bandwidth of interval graphs*, SIAM J. Discrete Math., 3 (1990), pp. 373–375, https://doi.org/10.1137/0403033.

[22] A. Lakshman and P. Malik, *Cassandra: A decentralized structured storage system*, SIGOPS Oper. Syst. Rev., 44 (2010), pp. 35–40.

[23] L. Lamport, *On interprocess communication, Part* I: *Basic formalism and Part* II: *Algorithms*, Distributed Comput., 1 (1986), pp. 77–101.

[24] H. Lee and J. L. Welch, *Randomized registers and iterative algorithms*, Distributed Comput., 17 (2005), pp. 209–221.

[25] D. Malkhi, M. K. Reiter, and R. N. Wright, *Probabilistic quorum systems*, in Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC), ACM, New York, 1997, pp. 267–273.

[26] M. McKenzie, H. Fan, and W. Golab, *Fine-tuning the consistency-latency trade-off in quorum-replicated distributed storage systems*, in Proceedings of the IEEE International Conference on Big Data, Scalable Cloud Data Management (SCDM) Workshop, IEEE, Washington, DC, 2015, pp. 1708–1717.

[27] J. Misra, *Axioms for memory access in asynchronous hardware systems*, ACM Trans. Program. Lang. Syst., 8 (1986), pp. 142–153.

[28] R. Niedermeier, *Invitation to Fixed-Parameter Algorithms*, Oxford University Press, Oxford, UK, 2006.

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

[30] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, *YCSB++: Benchmarking and performance debugging advanced features in scalable table stores*, in Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC), ACM, New York, 2011, 9.

[31] J. B. Saxe, *Dynamic-programming algorithms for recognizing small-bandwidth graphs in polynomial time*, SIAM J. Algebraic Discrete Methods, 1 (1980), pp. 363–369, https://doi.org/10.1137/0601042.

[32] R. Sedgewick and K. D. Wayne, *Algorithms*, 4th ed., Addison–Wesley Professional, Boston, MA, 2011, pp. 661–666.

[33] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, *Serving large-scale batch computed data with Project Voldemort*, in Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST) (San Jose, CA), USENIX Association, Berkeley, CA, 2012, p. 18.

[34] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, *Consistency-based service level agreements for cloud storage*, in Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP), ACM, New York, 2013, pp. 309–324.

[35] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, *Managing update conflicts in Bayou, a weakly connected replicated storage system*, in Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), ACM, New York, 1995, pp. 172–182.

[36] 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), ACM, New York, 1999, pp. 163–172.

[37] W. Vogels, *Eventually consistent*, Queue, 6 (2008), pp. 14–19.

[38] 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 5th Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, 2011.

[39] H. Yu and A. Vahdat, *Design and evaluation of a conit-based continuous consistency model for replicated services*, ACM Trans. Comput. Syst., 20 (2002), pp. 239–282.