The Space Complexity of Transactional Interactive Reads *

Masoud Saeida Ardekani
UPMC-LIP6
masoud.saeida-ardekani@lip6.fr

Marek Zawirski Pierre Sutra
INRIA & UPMC-LIP6
firstname.lastname@lip6.fr

Marc Shapiro
INRIA & UPMC-LIP6
marc.shapiro@acm.org

Abstract

Transactional Web Applications need to perform fast interactive reads while ensuring reasonable isolation guarantees. This paper studies the problem of taking consistent snapshots for transactions with interactive reads. We introduce four levels of freshness, and solutions to guarantee them. We also explore trade-offs between the space complexity and the freshness levels.

Categories and Subject Descriptors C.2.4 [*Computer-Communication Networks*]: Distributed Systems—Distributed applications;Distributed databases; H.3.4 [*Information Storage and Retrieval*]: Systems and Software—Distributed systems

General Terms Algorithms, Performance

Keywords Transactional Processing, Consistent Snapshot, Cloud Computing, Data Freshness, Key-Value Stores

1. Introduction

The cloud is a new paradigm for the dynamic provisioning of computing services usually supported by state-of-the-art data-centers containing ensembles of networked virtual machines. Among classical cloud-based applications, Transactional Web Applications (TWA) are data intensive and require a high level of availability and scalability. Several recent works have studied the inherent limitations of current cloud infrastructures [3, 5, 11] to support TWA applications, and novel solutions have been proposed [6, 7, 10].

Typical TWA applications need to perform interactive reads [7], i.e., the read set of the transaction changes over time, spanning multiple machines while ensuring reasonable isolation. Freshness of versions read by transaction matters. Beside of potential value for applications, freshness helps to reduce the abort rate of transactions and simplifies load balancing. In order to scale well, interactive reads are executed optimistically without relying on an expensive protocol. The usual approach is to timestamp transactions and use snapshots: a transaction reads the snapshot consisting of all the transactions that committed prior to it. Timestamping however requires a central server which is subject to failure and may become a bottleneck. To deal with this problem, recent transactional infrastructures [6, 10] propose instead to use distributed protocols. We observe that these protocols offer different guarantees with various space complexity (memory and network usage). For instance, a conservative protocol offers possibly stale values at low cost, while a more permissive system returns fresher snapshots at higher cost.

This paper studies the problem of reading consistent snapshots in a distributed transactional system and the relation between space complexity and freshness. We introduce four levels of freshness, and we study the cost of different solutions to attain each of these levels. In more detail, our contributions are the following:

- We define four freshness levels. Under maximum freshness, read operations observe the most recent committed versions of objects. Base freshness is the level offered by the timestamp-based protocol. Positive freshness is intermediate between maximum and base freshness. Negative freshness does not guarantee progress of transactions.
- We present four algorithms, matching the freshness levels. We compare their space complexity, and argue informally that higher freshness increases space complexity.

The remainder of the paper is organized as follows. In Section 2, we describe our transactional system model. In Section 3, we give a specification of the interactive read problem, define freshness levels and present a generic algorithm for interactive reads. In Section 4, we study solutions to the problem of taking consistent snapshots, and assess the space complexity of each freshness level. We compare to related works in Section 5 and conclude in Section 6.

2. Model

In this section, we formally introduce the model used to represent histories, transactions, and the transactional system.

* This research is supported in part by ANR projects ConcoRDanT (ANR-10-BLAN 0208) and Prose (ANR-09-VERS-007-02), and by a Google European Doctoral Fellowship (2010–2013), awarded to Marek Zawirski.

HotCDP 2012 - 1st International Workshop on Hot Topics in Cloud Data Processing
April 10, 2012, Bern, Switzerland.

Objects & transactions Let $Objects$ be a set of objects, and $\mathcal{T}$ be a set of transaction identifiers. Given an object x, and a transaction identifier i, x_i denotes version i of x. Initially every object x has version x_0. A *transaction* $T_{i \in \mathcal{T}}$ is a finite sequence of read and write operations followed by a terminating operation, which can be either a commit (c_i) or an abort (a_i). In a history (defined shortly), $w_i(x_i)$ denotes that transaction T_i writes version i of object x, and $r_i(x_j)$ means that T_i reads version j of object x. In a transaction, every object is read or written at most once, and every write is preceded by a read of the same object. We consider the first operation of a transaction to be its starting point (denoted s_i).

We denote by $WS(T_i)$ the write set of T_i, i.e., the set of objects written by transaction T_i. Similarly, $RS(T_i)$ denotes the read set of transaction T_i.

Histories A *history* h is a partially ordered set of operations such that (1) for every operation o_i appearing in h, transaction T_i terminates in h, (2) for every two operations o_i and o'_i appearing in h, if o_i precedes o'_i in T_i, then $o_i <_h o'_i$, (3) for every read $r_i(x_j)$ in h, there exists a write operation $w_j(x_j)$ such that $w_j(x_j) <_h r_i(x_j)$, and (4) any two write operations over the same objects are ordered by $<_h$. We note $\ll_h$ the version order induced by h between the different versions of an object, i.e., $\ll_h = \{(x_i, x_j) : \exists x, w_i(x_i) <_h w_j(x_j)\}$.

System A database $\mathcal{D}$ is a finite set of tuples (x, v, i) where x is an object (data item), v a value, and $i \in \mathcal{T}$ a version. We consider a message-passing distributed system of n processes $\Pi = \{p_1, \ldots, p_n\}$. Each process holds a copy of a subset of $\mathcal{D}$ as its local database. For some object x, $Replicas(x)$ denotes the set of processes (or replicas) that hold a copy of x. The coordinator of T_i, denoted by $coord(T_i)$, is in charge of executing T_i on behalf of some client (not modeled). The coordinator does not know the read set nor the write set of T_i in advance, hence the system supports *interactive reads*.

In our model, every update transaction that writes x must read it previously. This implies that every update transaction depends on a previous update transaction, or the initial transaction T_0. More formally:

Definition (Dependency)**.** *Consider a history h, and two transactions T_i and T_j in h. We note $T_j \lhd T_i$ when $r_i(x_j)$ is in h. By extension over transitive closure, transaction T_i* depends *on transaction T_j when $T_j \lhd^* T_i$ holds.*

3. Problem Specification

This paper studies the space complexity needed for taking a consistent snapshot in the presence of interactive reads, and under different freshness levels.

To illustrate the problem, we consider the execution depicted in Figure 1. In this execution, processes P_1, P_2 and P_3 replicate objects x, y and z respectively. Transactions T_1 and T_2 write new values to object x and z respectively. Transaction T_3 reads version z_2 (installed by T_2) and writes to object y. Transactions T_4 and T_5 modify objects y and z. We focus on the read-only transaction T_a that first reads version x_1 of object x, then tries to read objects y and z. Its read operations are delivered in real time according to the dotted line. Using this execution, we introduce consistent snapshots as well as our different freshness levels.

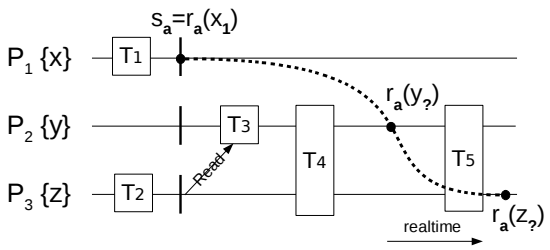

Figure 1: Sample execution. Dashed line: start of read-only transaction T_a. Dotted line: execution of transaction T_a in real-time.

3.1 Consistent Snapshot

Roughly speaking, a transaction that would miss some effect of another transaction upon which it depends, observes a non-consistent snapshot [2]. For example, consider transaction T_a in Figure 1. It observes a non-consistent snapshot if it reads versions $\{x_1, y_3, z_0\}$, $\{x_1, y_4, z_2\}$ or $\{x_1, y_4, z_5\}$.

Definition (Consistent snapshot)**.** *A transaction T_i in a history h observes a consistent snapshot when for every object x, if $r_i(x_j)$ and $w_k(x_k)$ both belong to h and T_i depends on T_k , then $x_k \ll x_j$ holds. By extension, history h is consistent if all the transactions in h observe a consistent snapshot.*

3.2 Freshness

We consider the following four levels of freshness:

Negative freshness is the weakest freshness level. A history has negative freshness if it is consistent and a read operation of a transaction can arbitrarily read an old version. Since solutions that ensure negative freshness are trivial (e.g., always returning the initial version) and undesirable, we do not consider this property in this paper.

Base freshness guarantees that every transaction T_i observes the most recent and consistent versions of all objects committed before the first operation of the transaction. We recall that the first operation of transaction T_i is considered to be the start operation s_i. For instance, T_a has base freshness when it reads versions $\{x_1, y_0, z_2\}$ (dashed line).

Definition (Base Freshness)**.** *A consistent history h has base freshness iff for every read operation $r_i(x_j)$, c_j is before s_i and there is no version x_k such that $c_j <_h c_k <_h s_i$ holds.*

Despite the fact that most of the existing storage systems offer base freshness, as we mentioned in Section 1, some applications may favor higher level of freshness.

Maximum freshness is the highest freshness level a history can ensure. It guarantees that each read observes the

most recent consistent version of an object. For example, the history depicted in Figure 1 has maximum freshness when transaction T_a reads versions $\{x_1, y_4, z_4\}$ (notice that reading $\{x_1, y_4, z_5\}$ is not consistent).

Definition (Maximum Freshness)**.** *A consistent history h has maximum freshness iff for every read operation $r_i(x_j)$, if $w_j(x_j) <_h w_k(x_k) <_h r_i(x_j)$ holds for some write $w_k(x_k)$, then replacing $r_i(x_j)$ by $r_i(x_k)$ in h leads to a non-consistent snapshot.*

Although desirable, maximum freshness is very demanding, and as we will see in Section 4, few systems can ensure this property. Therefore, we introduce an intermediate level which is stronger than base freshness, yet cheaper than maximum freshness called *positive freshness*. In positive freshness, a transaction can read a version installed after the transaction has started. For instance, the history depicted in Figure 1 has positive freshness if transaction T_a reads versions $\{x_1, y_3, z_2\}$.

Definition (Positive Freshness)**.** *A consistent history h has positive freshness iff for every read operation $r_i(x_j)$, there is no a version x_k such that $c_j <_h c_k <_h s_i$ holds.*

3.3 Generic Algorithm

Algorithm 1 shows a generic certification-based algorithm $\mathcal{A}_{gen}$ independent of freshness levels. The algorithm is generic in the sense that by final modifications of the functions whose names are underlined, different freshness levels can be achieved. In Section 4, we will consider different implementations of these functions, and how they can affect freshness and space complexity.

Upon receiving a begin operation, $\mathcal{A}_{gen}$ calls the initialization function (line 5). Write operations are executed and stored locally (line 8). To read an object x, the transaction coordinator $coord(T_i)$ first checks to see if x has been previously updated. If so, it returns the corresponding value (line 11), otherwise, $coord(T_i)$ sends a READ_RESOLVE request to a process that replicates object x (line 13).

When a process receives the READ_RESOLVE from a coordinator, it calls a function *choose* in order to select a version of the object such that it reads a consistent snapshot of the system (line 19), and returns the result back to $coord(T_i)$.

We assume that, upon receiving a request for committing transaction T_i, a certification test is performed to guarantee that committing T_i ensures the consistency criterion of the system (line 23). The consistency criterion should at least ensure that no two concurrent write-conflicting transaction both commit (e.g. snapshot isolation). If the certification outcome is positive, then T_i is applied at all processes, as defined by concrete algorithms in the next section.

Algorithm 1 Generic Algorithm $\mathcal{A}_{gen}$

```
1:  Variables:
2:      db                                          // committed object versions
3:
4:  execute(BEGIN, T_i)
5:      act:    initialize(T_i)                                          // 1
6:
7:  execute(WRITE, x, v, T_i)
8:      act:    updates(T_i) ← updates(T_i) ∪ {x, v, i}
9:
10: execute(READ, x, T_i)
11:     act:    if ∃(x, v, i) ∈ updates(T_i) then return v
12:             else
13:               send ⟨READ_RESOLVE, T_i, x⟩ to Replicas(x)
14:               wait until received ⟨READ_RESOLVED, T_i, x, v⟩
15:               return v
16:
17: readResolve(x, T_i)
18:     pre:    received ⟨READ_RESOLVE, T_i, x⟩ from q
19:     act:    (x, v, l) ← choose ⟨T_i, x⟩                              // 2
20:             send ⟨READ_RESOLVED, T_i, x, v⟩ to q
21:
22: execute(COMMIT, T_i)
23:     pre:    certify(T_i)
24:     act:    apply(T_i)                                               // 3
```

4. The Spectrum of Solutions

In this section we present implementations of the generic algorithm $\mathcal{A}_{gen}$ using different object versioning mechanisms and read algorithms. All of these algorithms produce consitent histories. We also classify the algorithms by offered level of freshness, and study their space complexity.

4.1 Maximum Freshness Algorithm

In order to guarantee maximum freshness, an algorithm must allow reading the latest version that is consistent with the set of reads previously executed. Since in some cases this version might not exist at the time the transaction starts, this property is expensive to verify locally. It requires each process to locally compute dependency relation $\lhd^*$ between pairs of committed transactions, and to determine the set of objects modified between dependent transactions. Saeida Ardekani et al. [9] developed the *dependence vector* (DV) which represents this information compactly.

A dependence vector is a vector with the size of the number of objects in the system. Dependence vector is assigned to every version of an object and all objects written by the same transaction have the same dependence vector. Thus for simplicity, we present them as vectors assigned to transactions, i.e. $DV(T_i)$ denotes a dependence vector assigned to new versions of objects modified by T_i. For version x_i written by T_i, the object's own entry $DV(T_i)[x]$ represents its version number, i.e. number of committed transactions that modified object x prior to T_i, inclusive. Other entries of a dependence vector, $DV(T_i)[y]$ for $y \notin WS(T_i)$ identify the versions of objects that T_i depends upon.

Algorithm 2 presents maximum freshness implementation of $\mathcal{A}_{gen}$ denoted by $\mathcal{A}_{max}$. The algorithm keeps track of versions read by a transaction and maintains transaction's

Algorithm 2 Maximum Freshness Algorithm $\mathcal{A}_{max}$

1: *initialize*(T_i) // no initialization necessary
2:
3: *choose* $\langle T_i, x\rangle$
4: **pre:** $\exists(x,v,l) \in db : \forall y_k \in RS(T_i) :$
5: $DV(T_l)[x] \geq DV(T_k)[x] \wedge$
6: $DV(T_l)[y] \leq DV(T_k)[y]$
7: **act:** **return** $max\ (x,v,l)$ // the latest according to $\ll_h$
8:
9: *apply*(T_i)
10: **act:** $DV(T_i) \leftarrow max\ \{DV(T_l) : x_l \in RS(T_i)\}$
11: $+ \sum_{y_i \in WS(T_i)} \hat{y}$ // $\hat{y}$ is a unit vector
12: *send* $\langle \text{APPLY}, T_i\rangle$ *to* Π
13:
14: *performApply*(T_i)
15: **pre:** $(received\ \langle \text{APPLY}, T_i\rangle\ from\ q)$
16: **act:** $db \leftarrow db \cup \{(x,v,l) \in updates(T_i) :$
17: $p_j \in Replicas(x)\}$

dependence vectors as previously described. The read rules ensure that selected object version does not miss any dependency induced by previous reads (line 5) and does not invalidate any version previously read (line 6).

Upon committing T_i, its DV is computed by taking the maximum of DVs of objects read by T_i (line 10) and incrementing the entries of modified objects (line 11). The transaction is then applied at all processes.

Theorem 1. *Every history with maximum freshness is admissible by algorithm $\mathcal{A}_{max}$.*

Algorithm $\mathcal{A}_{max}$ guarantees maximum freshness, and its space complexity is $O(m)$ where m is the number of objects in the system. We conjecture that this is optimal:

Conjecture 1. *Any implementation of algorithm $\mathcal{A}_{gen}$ that admits all the histories with maximum freshness has a space complexity of $\Omega(m)$.*

4.2 Positive Freshness Algorithms

Since keeping track of the dependence relations and of modified objects accurately is expensive, it is reasonable to consider their safe approximations and require only positive freshness.

One way to achieve this is by partitioning objects and serializing updates within every partition to represent versions more compactly. Consider algorithm $\mathcal{A}_{++}$, a variant of $\mathcal{A}_{max}$ using coarser grained vectors with entry only per each partition. Formally, let equivalence relation $\sim$ divide objects into classes (partitions) according to the set of processes where they are replicated, i.e. $x \sim y \triangleq (Replicas(x) = Replicas(y))$. We denote by $[x]$ the equivalence class (partition) of object x. Each partition of objects can then use its own entry in DV, i.e. commit of transaction T_i increases only entries of modified partitions $\{[x_i] : x_i \in WS(T_i)\}$.

The resulting vector, with the size number of partitions, represents a strengthening of the order induced by the original DV. Thus, it approximates the dependence relation and represents the set of objects possibly modified between the two transactions. The optimization, however, requires extra input from the certification to enforce uniqueness of DVs.

Algorithm 3 Positive Freshness Algorithm $\mathcal{A}_+$ at process p_j

1: **Variables:**
2: $currVV$ // vector of processes, initially $[0,\ldots,0]$
3:
4: *initialize*(T_i)
5: **act:** $initVV(T_i) \leftarrow currVV$
6:
7: *choose* $\langle T_i, x\rangle$
8: **pre:** $currVV \geq initVV(T_i)$
9: $\forall y_k \in RS(T_i) : currVV \geq VV(T_k)$
10: **act:** $V_1 \leftarrow \{(x,v,l) \in db : VV(T_l) \leq initVV(T_i)\}$
11: $V_2 \leftarrow \{(x,v,l) \in db : VV(T_l) \not\geq initVV(T_i) \wedge$
12: $\forall y_k \in RS(T_i) : VV(T_l) \not\geq VV(T_k)\}$
13: **return** $max\ V_1 \cup V_2$ // the latest according to $\ll_h$
14:
15: *apply*(T_i)
16: **pre:** $currVV \geq initVV(T_i)$
17: $\forall y_k \in RS(T_i) : currVV \geq VV(T_k)$
18: **act:** $VV(T_i) \leftarrow currVV + \hat{j}$
19: *performApply*(T_i)
20: *send* $\langle \text{APPLY}, T_i\rangle$ *to* $\Pi \setminus p_j$
21:
22: *performApply*(T_i)
23: **pre:** $(p_j = coord(T_i) \vee received\ \langle \text{APPLY}, T_i\rangle\ from\ q)$
24: $\exists k : currVV + \hat{k} \geq VV(T_i)$
25: **act:** $db \leftarrow db \cup \{(x,v,l) \in updates(T_i) :$
26: $p_j \in Replicas(x)\}$
27: $currVV \leftarrow \max(currVV,\ VV(T_i))$

Another natural choice to approximate dependence is to use the happened-before order. Our new Algorithm 3, noted $\mathcal{A}_+$, extends the algorithm given by Sovran et al. [10], using version vectors to capture the happened-before order between transactions. A *version vector* (VV) is a vector with one entry per process, defined for each committed transaction. The algorithm maintains the following invariant used during reads: $T_i \lhd^* T_j \Rightarrow VV(T_i) < VV(T_j)$.

Every process in $\mathcal{A}_+$ maintains the vector $currVV$ identifying locally committed transactions. At the beginning of transaction T_i, this vector becomes the snapshot point of the transaction ($initVV(T_i)$). As in the Sovran's algorithm, the transaction may read versions that were visible at the beginning of the transaction (line 10). Considering this rule alone, it offers only base freshness.

To allow extending snapshots after the transaction has begun (positive freshness), the algorithm also reads from transactions that were committed concurrently to the transactions included in the current snapshot (lines 11 to 12).

Upon a successful certification of transaction T_i, the coordinator assigns the transaction a vector $VV(T_i)$ that dominates both all transactions in the snapshot of T_i and the coordinator's local $currVV$ (line 18). This construction ensures

that the partial order over version vectors is compatible with the transaction dependence order.

We now state two results regarding freshness of histories admissible by variants of $\mathcal{A}_+$ algorithm.

Conjecture 2. *Every history admissible by algorithm $\mathcal{A}_+$ has positive freshness.*

Let algorithm $\mathcal{A}_0$ be similar to the algorithm $\mathcal{A}_+$ without reading concurrent transactions ($V_2 = \emptyset$ in line 11) [10].

Theorem 2. *Every history admissible by algorithm $\mathcal{A}_0$ has base freshness.*

The space complexity of Algorithm $\mathcal{A}_+$ is $O(n)$, where n is the number of processes in the system. It is interesting to consider why this algorithm does not always offer maximum freshness: $\mathcal{A}_+$ extends the initial snapshot to transactions that are not ordered by happened-before and independent with the transactions included in the initial snapshot. However, (1) it does not admit snapshots including independent transactions ordered by VV, and (2) it does not allow to extend snapshot with transactions depending on the initial snapshot (ordered by VV) that did not modify the objects previously read by transaction T_i. These two cases require more accurate dependence tracking. This is an inherent trade-off between the freshness of interactive reads and the space-complexity of the algorithm.

5. Related Work

Several papers study the freshness problem in the context of key-value stores. Wada et al. [11] investigate the probability of observing stale data in different cloud systems. More recently, Bailis et al. [1] have developed a probabilistic model to quantify staleness in quorum-replicated data stores. These works study the relation between the freshness of reads and the protocol used by key value stores, such as a read-write quorums. However, they do not consider transactions and consistency across different keys.

Golab et al. [4] provide an online algorithm for atomicity verification of individual keys in a key-value store. They define the staleness of observed read values as the number of missed writes (k-atomicity) or the amount of time (Δ-atomicity). These definitions prove to deliver measurable metrics in practice. We plan to extend these metrics into the transactional context in order to measure more fine grained freshness levels.

$\mathcal{A}_0$ was introduced by Sovran et al. [10] and guarantees only base freshness. Raynal et al. [8] used a similar pessimistic algorithm which makes it less practical. Saeida Ardekani et al. [9] proposed $\mathcal{A}_{max}$ using DVs.

6. Conclusions and Future Work

In this paper, we define negative, base, positive and maximum freshness levels. We present an optimal, but costly, maximum freshness algorithm, and a cheap base freshness algorithm. We explore the trade-off between space complexity and freshness, and introduced two novel algorithms ($\mathcal{A}_+$ and $\mathcal{A}_{++}$). They ensure positive freshness for transactional interactive reads at lower cost than $\mathcal{A}_{max}$. Table 1 summarizes our results by relating freshness to space complexity.

Algorithm	Freshness	Space Complexity
$\mathcal{A}_{max}$ [9]	maximum	$O(m)$
$\mathcal{A}_{++}$	positive	$O(p)$
$\mathcal{A}_+$	positive	$O(n)$
$\mathcal{A}_0$ [10]	base	$O(n)$

Table 1: Space complexity w.r.t. freshness level for each of the presented algorithms - variables n, m and p denote respectively the number of processes, objects and partitions in the system.

For the future work, we plan to compare the four algorithms experimentally, as well as studying in more details the relation between space complexity and freshness.

References

[1] P. Bailis, S. Venkataraman, J. M. Hellerstein, M. Franklin, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. Technical Report UCB/EECS-2012-4, Berkeley, Jan 2012.

[2] A. Chan and R. Gray. Implementing Distributed Read-Only Transactions. *IEEE Transactions on Software Engineering*, SE-11(2):205–212, Feb. 1985.

[3] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *SIGMOD*, pages 117–128. ACM, 2000.

[4] W. M. Golab, X. Li, and M. A. Shah. Analyzing consistency properties for fun and profit. In *PODC*, pages 197–206, 2011.

[5] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *SIGMOD*, pages 579–590. ACM, 2010.

[6] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *SOSP*, pages 401—-416. ACM, 2011.

[7] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. *Middleware*, pages 155–174, 2004.

[8] M. Raynal, G. Thia-Kime, and M. Ahamad. From serializable to causal transactions for collaborative applications. In *EUROMICRO*, pages 314–321. IEEE Comput. Soc, 1997.

[9] M. Saeida Ardekani, P. Sutra, N. Preguiça, and M. Shapiro. Non-Monotonic Snapshot Isolation. Technical Report RR-7805, INRIA, Nov. 2011.

[10] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, pages 385–400. ACM, 2011.

[11] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR*, pages 134–143, 2011.