Highly Available Transactions: Virtues and Limitations (Extended Version)

Peter Bailis, Aaron Davidson, Alan Fekete†, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica
UC Berkeley and †University of Sydney

ABSTRACT

To minimize network latency and remain online during server failures and network partitions, many modern distributed data storage systems eschew transactional functionality, which provides strong semantic guarantees for groups of multiple operations over multiple data items. In this work, we consider the problem of providing Highly Available Transactions (HATs): transactional guarantees that do not suffer unavailability during system partitions or incur high network latency. We introduce a taxonomy of highly available systems and analyze existing ACID isolation and distributed data consistency guarantees to identify which can and cannot be achieved in HAT systems. This unifies the literature on weak transactional isolation, replica consistency, and highly available systems. We analytically and experimentally quantify the availability and performance benefits of HATs—often two to three orders of magnitude over wide-area networks—and discuss their necessary semantic compromises.

1 Introduction

The last decade has seen a shift in the design of popular large-scale database systems, from the use of transactional RDBMSs [13, 37, 36] to the widespread adoption of loosely consistent distributed key-value stores [20, 22, 29]. Core to this shift was the 2000 introduction of Brewer's CAP Theorem, which stated that a highly available system cannot provide "strong" consistency guarantees in the presence of network partitions [15]. As formally proven [34], the CAP Theorem pertains to a data consistency model called linearizability, or the ability to read the most recent write to a data item that is replicated across servers [40]. However, despite its narrow scope, the CAP Theorem is often misconstrued as a broad result regarding the ability to provide ACID database properties with high availability [8, 15, 21]; this misunderstanding has led to substantial confusion regarding replica consistency, transactional isolation, and high availability. The recent resurgence of transactional systems suggests that programmers value transactional semantics, but most existing transactional data stores do not provide availability in the presence of partitions [18, 21, 41, 24, 47, 58, 60].

Indeed, serializable transactions—the gold standard of traditional ACID databases—are not achievable with high availability in the presence of network partitions [27]. However, database systems have a long tradition of providing weaker isolation and consistency guarantees [2, 11, 36, 37, 42]. Today's ACID and NewSQL databases often employ weak isolation models due to concurrency and performance benefits; weak isolation is overwhelmingly the default setting in these stores and is often the only option offered (Section 3). While weak isolation levels do not provide serializability for general-purpose transactions, they are apparently strong enough to deliver acceptable behavior to many application programmers and are substantially stronger than the semantics provided by current highly available systems. This raises a natural question: which semantics can be provided with high availability?

To date, the relationship between ACID semantics and high availability has not been well explored. We have a strong understanding of weak isolation in the single-server context from which it originated [2, 11, 37] and many papers offer techniques for providing distributed serializability [13, 24, 26, 41, 60] or snapshot isolation [42, 58]. Additionally, the distributed computing and parallel hardware literature contains many consistency models for single operations on replicated objects [22, 40, 47, 48, 59]. However, the literature lends few clues for providing semantic guarantees for multiple operations operating on multiple data items in a highly available distributed environment.

Our main contributions in this paper are as follows. We relate the many previously proposed database isolation and data consistency models to the goal of high availability, which guarantees a response from each non-failing server in the presence of arbitrary network partitions between them. We classify which among the wide array of models are achievable with high availability, denoting them as *Highly Available Transactions* (HATs). In doing so, we demonstrate that although many implementations of HAT semantics are not highly available, this is an artifact of the implementations rather than an inherent property of the semantics. Our investigation shows that, besides serializability, Snapshot Isolation and Repeatable Read isolation are not HAT-compliant, while most other isolation levels are achievable with high availability. We also demonstrate that many weak replica consistency models from distributed systems are both HAT-compliant and simultaneously achievable with several ACID properties.

Our investigation is based on both impossibility results and several constructive, proof-of-concept algorithms. For example, Snapshot Isolation and Repeatable Read isolation are not HAT-compliant because they require detecting conflicts between concurrent updates (as needed for preventing Lost Updates or Write Skew phenomena), which we show is unavailable. However, Read Committed isolation, transactional atomicity (Section 5.1.2), and many other consistency models from database and distributed systems are achievable via algorithms that rely on multi-versioning and limited client-side caching. For several guarantees, such as causal consistency with phantom prevention and ANSI Repeatable Read, we consider a modified form of high availability in which clients "stick to" (i.e., have affinity with) at least one server—a property which is often implicit in the distributed systems literature [40, 47, 48] but which requires explicit consideration in a client-server replicated database context. This sticky availability is widely employed [47, 63] but is a less restrictive model (and therefore more easily achievable) than traditional high availability.

At a high level, the virtues of HATs are guaranteed responses

 Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.
Proceedings of the VLDB Endowment, Vol. 7, No. 3

from any replica, low latency, and a range of semantic guarantees including several whose usefulness is widely accepted such as Read Committed. However, highly available systems are fundamentally unable to prevent concurrent updates to shared data items and cannot provide recency guarantees for reads. To understand when these virtues and limitations are relevant in practice, we survey both practitioner accounts and academic literature, perform experimental analysis on modern cloud infrastructure, and analyze representative applications for their semantic requirements. Our experiences with a HAT prototype running across multiple geo-replicated datacenters indicate that HATs offer a one to three order of magnitude latency decrease compared to traditional distributed serializability protocols, and they can provide acceptable semantics for a wide range of programs, especially those with monotonic logic and commutative updates [4, 57]. HAT systems can also enforce arbitrary foreign key constraints for multi-item updates and, in some cases, provide limited uniqueness guarantees. However, HATs can fall short for applications with concurrency-sensitive operations, requiring unavailable, synchronous coordination.

Finally, we recognize that the large variety of ACID isolation levels and distributed consistency models (and therefore those in our taxonomy) can be confusing; the subtle distinctions between models may appear to be of academic concern. Accordingly, we offer the following pragmatic takeaways:

1. The default (and sometimes strongest) configurations of most widely deployed database systems expose a range of anomalies that can compromise application-level consistency.
2. Many of these "weak isolation" models are achievable without sacrificing high availability if implemented correctly. However, none of the achievable models prevents concurrent modifications.
3. In addition to providing a guaranteed response and horizontal scale-out, these highly available HAT models allow one to three order of magnitude lower latencies on current infrastructure.
4. For correct behavior, applications may require a combination of HAT and (ideally sparing use of) non-HAT isolation levels; future database designers should plan accordingly.

2 Why High Availability?

Why does high availability matter? Peter Deutsch starts his classic list of "Fallacies of Distributed Computing" with two concerns fundamental to distributed database systems: "*1.)* The network is reliable. *2.)* Latency is zero" [31]. In a distributed setting, network failures may prevent database servers from communicating, and, in the absence of failures, communication is slowed by factors like physical distance, network congestion, and routing. As we will see (Section 4), highly available system designs mitigate the effects of network *partitions* and *latency*. In this section, we draw on a range of evidence that indicates that partitions occur with frequency in real-world deployments and latencies between datacenters are substantial, often on the order of several hundreds of milliseconds.

2.1 Network Partitions at Scale

According to James Hamilton, Vice President and Distinguished Engineer on the Amazon Web Services team, "network partitions should be rare but net gear continues to cause more issues than it should" [38]. Anecdotal evidence confirms Hamilton's assertion. In April 2011, a network misconfiguration led to a twelve-hour series of outages across the Amazon EC2 and RDS services [7]. Subsequent misconfigurations and partial failures such as another EC2 outage in October 2012 have led to full site disruptions for popular web services like Reddit, Foursquare, and Heroku [32]. At global scale, hardware failures—like the 2011 outages in Internet backbones in North America and Europe due a router bug [56]—and misconfigurations like the BGP faults in 2008 [50] and 2010 [51] can cause widespread partitioning behavior.

Many of our discussions with practitioners—especially those operating on public cloud infrastructure—as well as reports from large-scale operators like Google [28] confirm that partition management is an important consideration for service operators today. System designs that do not account for partition behavior may prove difficult to operate at scale: for example, less than one year after its announcement, Yahoo!'s PNUTS developers explicitly added support for weaker, highly available operation. The engineers explained that "strict adherence [to strong consistency] leads to difficult situations under network partitioning or server failures...in many circumstances, applications need a relaxed approach" [52].

Several recent studies rigorously quantify partition behavior. A 2011 study of several Microsoft datacenters observed over 13,300 network failures with end-user impact, with an estimated median 59,000 packets lost per failure. The study found a mean of 40.8 network link failures per day (95th percentile: 136), with a median time to repair of around five minutes (and up to one week). Perhaps surprisingly, provisioning redundant networks only reduces impact of failures by up to 40%, meaning network providers cannot easily curtail partition behavior [35]. A 2010 study of over 200 wide-area routers found an average of 16.2–302.0 failures per link per year with an average annual downtime of 24–497 minutes per link per year (95th percentile at least 34 hours) [62]. In HP's managed enterprise networks, WAN, LAN, and connectivity problems account for 28.1% of all customer support tickets while 39% of tickets relate to network hardware. The median incident duration for highest priority tickets ranges from 114–188 minutes and up to a full day for all tickets [61]. Other studies confirm these results, showing median time between connectivity failures over a WAN network of approximately 3000 seconds with a median time to repair between 2 and 1000 seconds [49] as well as frequent path routing failures on the Internet [44]. A recent, informal report by Kingsbury and Bailis catalogs a host of additional practitioner reports [43]. Not surprisingly, isolating, quantifying, and accounting for these network failures is an area of active research in networking community [46].

These studies indicate that network partitions *do* occur within and across modern datacenters. We observe that these partitions must be met with either unavailability at some servers or, as we will discuss, relaxed semantic guarantees.

2.2 Latency and Planet Earth

Even with fault-free networks, distributed systems face the challenge of network communication latency, Deutsch's second "Fallacy." In this section, we quantify round-trip latencies, which are often large—hundreds of milliseconds in a geo-replicated, multi-datacenter context. Fundamentally, the speed at which two servers can communicate is (according to modern physics) bounded by the speed of light. In the best case, two servers on opposite sides of the Earth—communicating via a hypothetical link through the planet's core—require a minimum 85.1ms round-trip time (RTT; 133.7ms if sent at surface level). As services are replicated to multiple, geographically distinct sites, this cost of communication increases.

In real deployments, messages travel slower than the speed of light due to routing, congestion, and server-side overheads. To illustrate the difference between intra-datacenter, inter-datacenter, and inter-planetary networks, we performed a measurement study of network behavior on Amazon's EC2, a widely used public compute cloud. We measured one week of ping times (i.e., round-trip times, or RTTs) between all seven EC2 geographic "regions," across three "availability zones" (closely co-located datacenters),

	H2	H3
H1	0.55	0.56
H2		0.50

(a) Within us-east-b AZ

	C	D
B	1.08	3.12
C		3.57

(b) Across us-east AZs

	OR	VA	TO	IR	SY	SP	SI
CA	22.5	84.5	143.7	169.8	179.1	185.9	186.9
OR		82.9	135.1	170.6	200.6	207.8	234.4
VA			202.4	107.9	265.6	163.4	253.5
TO				278.3	144.2	301.4	90.6
IR					346.2	239.8	234.1
SY						333.6	243.1
SP							362.8

(c) Cross-region (CA: California, OR: Oregon, VA: Virginia, TO: Tokyo, IR: Ireland, SY: Sydney, SP: São Paulo, SI: Singapore)

Table 1: Mean RTT times on EC2 (min and max highlighted)

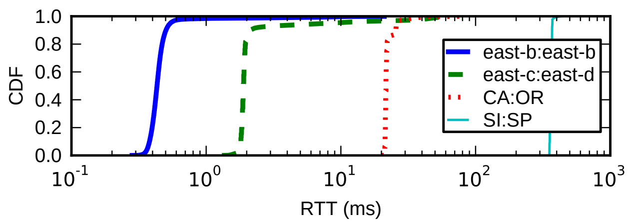

Figure 1: CDF of round-trip times for slowest inter- and intra-availability zone links compared to cross-region links.

and within a single "availability zone" (datacenter), at a granularity of 1s[1]. We summarize the results of our network measurement study in Table 1. On average, intra-datacenter communication (Table 1a) is between 1.82 and 6.38 times faster than across geographically co-located datacenters (Table 1b) and between 40 and 647 times faster than across geographically distributed datacenters (Table 1c). The cost of wide-area communication exceeds the speed of light: for example, while a speed-of-light RTT from São Paulo to Singapore RTT is 106.7ms, ping packets incur an average 362.8ms RTT (95th percentile: 649ms). As shown in Figure 1, the distribution of latencies varies between links, but the trend is clear: remote communication has a substantial cost. Quantifying and minimizing communication delays is also an active area of research in the networking community [65].

3 ACID in the Wild

The previous section demonstrated that distributed systems must address partitions and latency: what does this mean for distributed databases? Database researchers and designers have long realized that serializability is not achievable in a highly available system [27], meaning that, in environments like those in Section 2, database designs face a choice between availability and strong semantics. However, even in a single-node database, the coordination penalties associated with serializability can be severe and are manifested in the form of decreased concurrency (and, subsequently, performance degradation, scalability limitations, and, often, aborts due to deadlock or contention) [37]. Accordingly, to increase concurrency, database systems offer a range of ACID properties weaker than serializability: the host of so-called *weak isolation* models describe varying restrictions on the space of schedules that are allowable by the system [2, 5, 11]. None of these weak isolation models

[1]Data available at `http://bailis.org/blog/communication-costs-in-real-world-networks/`, `https://github.com/pbailis/aws-ping-traces`

Database	Default	Maximum
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	RR
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	S
MySQL 5.6	RR	S
MemSQL 1b	RC	RC
MS SQL Server 2012	RC	S
NuoDB	CR	CR
Oracle 11g	RC	SI
Oracle Berkeley DB	S	S
Oracle Berkeley DB JE	RR	S
Postgres 9.2.2	RC	S
SAP HANA	RC	SI
ScaleDB 1.02	RC	RC
VoltDB	S	S
RC: read committed, RR: repeatable read, SI: snapshot isolation, S: serializability, CS: cursor stability, CR: consistent read		

Table 2: Default and maximum isolation levels for ACID and NewSQL databases as of January 2013 (from [8]).

guarantees serializability, but, as we see below, their benefits are often considered to outweigh costs of possible consistency anomalies that might arise from their use.

To understand the prevalence of weak isolation, we recently surveyed the default and maximum isolation guarantees provided by 18 databases, often claiming to provide "ACID" or "NewSQL" functionality [8]. As shown in Table 2, only three out of 18 databases provided serializability by default, and eight did not provide serializability as an option at all. This is particularly surprising when we consider the widespread deployment of many of these non-serializable databases, like Oracle 11g, which are known to power major businesses and product functionality. Given that these weak transactional models are frequently used, our inability to provide serializability in arbitrary HATs appears non-fatal for practical applications. If application writers and database vendors have already decided that the benefits of weak isolation outweigh potential application inconsistencies, then, in a highly available environment that prohibits serializability, similar decisions may be tenable.

It has been unknown *which* of these guarantees can be provided with high availability, or are HAT-compliant. Existing algorithms for providing weak isolation are often designed for a single-node context and are, to the best of our knowledge, unavailable due to reliance on concurrency control mechanisms like locking that are not resilient to partial failure (Section 6.1). Moreover, we are not aware of any prior literature that provides guidance as to the relationship between weak isolation and high availability: prior work has examined the relationship between serializability and high availability [27] and weak isolation in general [2, 11, 37] but not weak isolation and high availability together. A primary goal in the remainder of this paper is to understand which models are HAT-compliant.

4 High Availability

To understand which guarantees can be provided with high availability, we must first define what high availability means. In this section, we will formulate a model that captures a range of availability models, including high availability, availability with stickiness, and transactional availability.

Informally, highly available algorithms ensure "always on" operation and, as a side effect, guarantee low latency. If users of a highly available system are able to contact a (set of) server(s) in a system, they are guaranteed a response; this means servers will not need to synchronously communicate with others. If servers are partitioned from one another, they do not need to stall in order to provide clients a "safe" response to operations. This lack of fast-path coordination also means that a highly available system also provides low latency [1]; in a wide-area setting, clients of a highly available system need not wait for cross-datacenter communication. To properly describe whether a *transactional* system is highly available, we need to describe what servers a client must contact as well as what kinds of responses a server can provide, especially given the possibility of aborts.

Traditionally, a system provides **high availability** if every user that can contact a correct (non-failing) server eventually receives a response from that server, even in the presence of arbitrary, indefinitely long network partitions between servers [34].[2] As in a standard distributed database, designated servers might perform operations for different data items. A server that can handle an operation for a given data item is called a *replica* for that item.[3]

4.1 Sticky Availability

In addition to high availability, which allows operations on any replica, distributed algorithms often assume a model in which clients always contact the same logical replica(s) across subsequent operations, whereby each of the client's prior operations (but not necessarily other clients' operations) are reflected in the database state that they observe. As we will discuss in Section 5, clients can ensure continuity between operations (e.g., reading their prior updates to a data item) by maintaining affinity or "stickiness" with a server or set of servers [63]. In a fully replicated system, where all servers are replicas for all data items, stickiness is simple: a client can maintain stickiness by contacting the same server for each of its requests. However, to stay "sticky" in a partially-replicated system, where servers are replicas for subsets of the set of data items (which we consider in this paper), a client must maintain stickiness with a single *logical* copy of the database, which may consist of multiple physical servers. We say that a system provides **sticky availability** if, whenever a client's transactions is executed against a copy of database state that reflects all of the client's prior operations, it eventually receives a response, even in the presence of indefinitely long partitions (where "reflects" is dependent on semantics). A client may choose to become sticky available by acting as a server itself; for example, a client might cache its reads and writes [10, 59, 66]. Any guarantee achievable in a highly available system is achievable in a sticky high availability system but not vice-versa.

4.2 Transactional Availability

Until now, we have considered single-object, single-operation availability. This is standard in the distributed systems literature (e.g., distributed register models such as linearizability all concern single objects [40]), yet the database literature largely focuses on transactions: groups of multiple operations over multiple objects. Accordingly, by itself, traditional definitions of high availability are insufficient to describe availability guarantees for transactions. Additionally, given the choice of *commit* and *abort* responses—which signal transaction success or failure to a client—we must take care in defining transactional availability.

We say that a transaction has **replica availability** if it can contact at least one replica for every item it attempts to access; this may result in "lower availability" than a non-transactional availability requirement (e.g., single-item availability). Additionally, given the possibility of system-initiated aborts, we need to ensure useful forward progress: a system can trivially guarantee clients a response by always aborting all transactions. However, this is an unsatisfactory system because nothing good (transaction commit) ever happens; we should require a *liveness* property [53].

A system cannot guarantee that every transaction will commit—transactions may choose to abort themselves—but we need to make sure that the system will not indefinitely abort transactions on its own volition. We call a transaction abort due to a transaction's own choosing (e.g., as an operation of the transaction itself or due to a would-be violation of a declared integrity constraint) an *internal abort* and an abort due to system implementation or operation an *external abort*. We say that a system provides **transactional availability** if, given replica availability for every data item in a transaction, the transaction eventually commits (possibly after multiple client retries) or internally aborts [8]. A system provides **sticky transactional availability** if, given sticky availability, a transaction eventually commits or internally aborts.

5 Highly Available Transactions

HAT systems provide transactions with transactional availability or sticky transactional availability. They offer latency and availability benefits over traditional distributed databases, yet they cannot achieve all possible semantics. In this section, we describe ACID, distributed replica consistency, and session consistency levels which can be achieved with high availability (Read Committed isolation, variants of Repeatable Read, atomic reads, and many session guarantees), those with sticky availability (read your writes, PRAM and causal consistency). We also discuss properties that cannot be provided in a HAT system (those preventing Lost Update and Write Skew or guaranteeing recency). We present a full summary of these results in Section 5.3.

As Brewer states, "systems and database communities are separate but overlapping (with distinct vocabulary)" [15]. With this challenge in mind, we build on existing properties and definitions from the database and distributed systems literature, providing a brief, informal explanation and example for each guarantee. The database isolation guarantees require particular care, since different DBMSs often use the same terminology for different mechanisms and may provide additional guarantees in addition to our implementation-agnostic definitions. We draw largely on Adya's dissertation [2] and somewhat on its predecessor work: the ANSI SQL specification [5] and Berenson et al.'s subsequent critique [11].

For brevity, we provide an informal presentation of each guarantee here (accompanied by appropriate references) but give a full set of formal definitions in Appendix A. In our examples, we exclusively consider read and write operations, denoting a write of value v to data item d as $w_d(v)$ and a read from data item d returning v as $r_d(v)$. We assume that all data items have the null value, $\perp$, at database initialization, and, unless otherwise specified, all transactions in the examples commit.

5.1 Achievable HAT Semantics

To begin, we present well-known semantics that can be achieved in HAT systems. In this section, our primary goal is feasibility, not performance. As a result, we offer proof-of-concept highly available algorithms that are not necessarily optimal or even efficient: the challenge is to prove the existence of algorithms that provide high availability. However, we briefly study a subset of their performance implications in Section 6.

[2]Under this definition from the distributed systems literature, systems that require a majority of servers to be online are not available. Similarly, a system which guarantees that servers provide a response with high probability is not available. This admittedly stringent requirement matches the assumptions made in the CAP Theorem [34] and guarantees low latency [1].

[3]There is a further distinction between a *fully replicated* system, in which all servers are replicas for all data items and a *partially replicated* system, in which at least one server acts as a replica for a proper subset of all data items. For generality, and, given the prevalence of these "sharded" or "partitioned" systems [20, 22, 24, 29, 41], we consider partial replication here.

5.1.1 ACID Isolation Guarantees

To begin, Adya captures **Read Uncommitted** isolation as *PL-1*. In this model, writes to each object are totally ordered, corresponding to the order in which they are installed in the database. In a distributed database, different replicas may receive writes to their local copies of data at different times but should handle concurrent updates (i.e., overwrites) in accordance with the total order for each item. *PL-1* requires that writes to different objects be ordered consistently across transactions, prohibiting Adya's phenomenon $G0$ (also called "Dirty Writes" [11]). If we build a graph of transactions with edges from one transaction to another and, when the former overwrites the latter's write to the same object, then, under Read Uncommitted, the graph should not contain cycles [2]. Consider the following example:

$$T_1 : w_x(1)\ w_y(1)$$
$$T_2 : w_x(2)\ w_y(2)$$

In this example, under Read Uncommitted, it is impossible for the database to order T_1's $w_x(1)$ before T_2's $w_x(2)$ but order T_2's $w_y(2)$ before T_1's $w_y(1)$. Read Uncommitted is easily achieved by marking each of a transaction's writes with the same timestamp (unique across transactions; e.g., combining a client's ID with a sequence number) and applying a "last writer wins" conflict reconciliation policy at each replica. Later properties will strengthen Read Uncommitted.

Read Committed isolation is particularly important in practice as it is the default isolation level of many DBMSs (Section 3). Centralized implementations differ, with some based on long-duration exclusive locks and short-duration read locks [37] and others based on multiple versions. These implementations often provide recency and monotonicity properties beyond what is implied by the name "Read Committed" and what is captured by the implementation-agnostic definition: under Read Committed, transactions should not access uncommitted or intermediate versions of data items. This prohibits both "Dirty Writes", as above, and also "Dirty Reads" phenomena. This isolation is Adya's *PL-2* and is formalized by prohibiting Adya's *G1*$\{$*a-c*$\}$ (or ANSI's $P1$, or "broad" $P1$ [2.2] from Berenson et al.). For instance, in the example below, T_3 should never see a = 1, and, if T_2 aborts, T_3 should not read a = 3:

$$T_1 : w_x(1)\ w_x(2)$$
$$T_2 : w_x(3)$$
$$T_3 : r_x(a)$$

It is fairly easy for a HAT system to prevent "Dirty Reads": if each client never writes uncommitted data to shared copies of data, then transactions will never read each others' dirty data. As a simple solution, clients can buffer their writes until they commit, or, alternatively, can send them to servers, who will not deliver their value to other readers until notified that the writes have been committed. Unlike a lock-based implementation, this implementation does not provide recency or monotonicity guarantees but it satisfies the implementation-agnostic definition.

Several different properties have been labeled **Repeatable Read** isolation. As we will show in Section 5.2.1, some of these are not achievable in a HAT system. However, the ANSI standardized implementation-agnostic definition [5] *is* achievable and directly captures the spirit of the term: if a transaction reads the same data more than once, it sees the same value each time (preventing "Fuzzy Read," or $P2$). In this paper, to disambiguate between other definitions of "Repeatable Read," we will call this property "cut isolation," since each transaction reads from a non-changing cut, or snapshot, over the data items. If this property holds over reads from discrete data items, we call it **Item Cut Isolation**, and, if we also expect a cut over predicate-based reads (e.g., `SELECT WHERE`; preventing Phantoms [37], or Berenson et al.'s $P3/A3$), we have the stronger property of **Predicate Cut-Isolation**. In the example below, under both levels of cut isolation, T_3 must read a = 1:

$$T_1 : w_x(1)$$
$$T_2 : w_x(2)$$
$$T_3 : r_x(1)\ r_x(a)$$

It is possible to satisfy Item Cut Isolation with high availability by having transactions store a copy of any read data at the client such that the values that they read for each item never changes unless they overwrite it themselves. These stored values can be discarded at the end of each transaction and can alternatively be accomplished on (sticky) replicas via multi-versioning. Predicate Cut Isolation is also achievable in HAT systems via similar caching middleware or multi-versioning that track entire logical ranges of predicates in addition to item based reads.

5.1.2 ACID Atomicity Guarantees

Atomicity, informally guaranteeing that either all or none of transactions' effects should succeed, is core to ACID guarantees. Although, at least by the ACID acronym, atomicity is not an "isolation" property, atomicity properties also restrict the updates visible to other transactions. Accordingly, here, we consider the *isolation* effects of atomicity, which we call **Monotonic Atomic View (MAV)** isolation. Under MAV, once some of the effects of a transaction T_i are observed by another transaction T_j, thereafter, all effects of T_i are observed by T_j. That is, if a transaction T_j reads a version of an object that transaction T_i wrote, then a later read by T_j cannot return a value whose later version is installed by T_i. Together with item cut isolation, MAV prevents Read Skew anomalies (Berenson et al.'s A5A) and is useful in several contexts such as maintaining foreign key constraints, consistent global secondary indexing, and maintenance of derived data. In the example below, under MAV, because T_2 has read T_1's write to y, T_2 must observe b = c = 1 (or later versions for each key):

$$T_1 : w_x(1)\ w_y(1)\ w_z(1)$$
$$T_2 : r_x(a)\ r_y(1)\ r_x(b)\ r_z(c)$$

T_2 can also observe $a = \perp$, $a = 1$, or a later version of x. In the hierarchy of existing isolation properties, we place MAV below Adya's *PL-2L* (as it does not necessarily enforce transitive read-write dependencies) but above Read Committed ($PL-2$). Notably, MAV requires disallows reading intermediate writes (Adya's $G1b$): observing all effects of a transaction implicitly requires observing the final (committed) effects of the transaction as well.

Perplexingly, discussions of MAV are absent from existing treatments of weak isolation. This is perhaps again due to the single-node context in which prior work was developed: on a single server (or a fully replicated database), MAV is achievable via lightweight locking and/or local concurrency control over data items [25, 42]. In contrast, in a distributed environment, MAV over arbitrary groups of non-co-located items is considerably more difficult to achieve with high availability.

As a straw man, replicas can store all versions ever written to each data item. Replicas can gossip information about versions they have observed and construct a lower bound on the versions that can be found on every replica (which can be represented by either a list of versions, or, more realistically, a vector clock). At the start of each transaction, clients can choose a *read timestamp* that is lower than or equal to the this global lower bound, and, during transaction execution, replicas return the latest version of each item that is not greater than the client's chosen timestamp. If this lower bound is advanced along transactional boundaries, clients will observe MAV. This algorithm has several variants in the literature [19, 66], and older versions can be asynchronously garbage collected.

We have developed a more efficient MAV algorithm, which we sketch here and provide greater detail in Appendix B. We begin with our Read Committed algorithm, but replicas wait to reveal new writes to readers until all of the replicas for the final writes in the transaction have received their respective writes (are *pending stable*). Clients include additional metadata with each write: a single timestamp for all writes in the transaction (e.g., as in Read Uncommitted) and a list of items written to in the transaction. When a client reads, the return value's timestamp and list of items form a lower bound on the versions that the client should read for the other items. When a client reads, it attaches a timestamp to its request representing the current lower bound for that item. Replicas use this timestamp to respond with either a write matching the timestamp or a pending stable write with a higher timestamp. Servers keep two sets of writes for each data item: the write with the highest timestamp that is pending stable and a set of writes that are not yet pending stable. This is entirely master-less and operations never block due to replica coordination.

5.1.3 Session Guarantees

A useful class of safety guarantees refer to real-time or client-centric ordering within a *session*, "an abstraction for the sequence of...operations performed during the execution of an application" [59]. These "session guarantees" have been explored in the distributed systems literature [59, 63] and sometimes in the database literature [26]. For us, a session describes a context that should persist between transactions: for example, on a social networking site, all of a user's transactions submitted between "log in" and "log out" operations might form a session.

Several session guarantees can be made with high availability:

Monotonic reads requires that, within a session, subsequent reads to a given object "never return any previous values"; reads from each item progress according to a total order (e.g., the order from Read Uncommitted).

Monotonic writes requires that each session's writes become visible in the order they were submitted. Any order on transactions (as in Read Uncommitted isolation) should also be consistent with any precedence that a global observer would see.

Writes Follow Reads requires that, if a session observes an effect of transaction T_1 and subsequently commits transaction T_2, then another session can only observe effects of T_2 if it can also observe T_1's effects (or later values that supersede T_1's); this corresponds to Lamport's "happens-before" relation [45]. Any order on transactions should respect this transitive order.

The above guarantees can be achieved by forcing servers to wait to reveal new writes (say, by buffering them in separate local storage) until each write's respective dependencies are visible on all replicas. This mechanism effectively ensures that all clients read from a globally agreed upon lower bound on the versions written. This is highly available because a client will never block due to inability to find a server with a sufficiently up-to-date version of a data item. However, it does not imply that transactions will read their own writes or, in the presence of partitions, make forward progress through the version history. The problem is that under non-sticky availability, a system must handle the possibility that, under a partition, an unfortunate client will be forced to issue its next requests against a partitioned, out-of-date server.

A solution to this conundrum is to forgo high availability and settle for sticky availability. Sticky availability permits three additional guarantees, which we first define and then prove are unachievable in a generic highly available system:

Read your writes requires that whenever a client reads a given data item after updating it, the read returns the updated value (or a value that overwrote the previously written value).

PRAM (Pipelined Random Access Memory) provides the illusion of serializing each of the operations (both reads and writes) within each session and is the combination of monotonic reads, monotonic writes, and read your writes [40].

Causal consistency [3] is the combination of all of the session guarantees [16] (alternatively, PRAM with writes-follow-reads) and is also referred to by Adya as *PL-2L* isolation [2]).

Read your writes is not achievable in a highly available system. Consider a client that executes the following two transactions:

$$T_1 : w_x(1)$$
$$T_2 : r_x(a)$$

If the client executes T_1 against a server that is partitioned from the rest of the other servers, then, for transactional availability, the server must allow T_1 to commit. If the same client subsequently executes T_2 against the same (partitioned) server in the same session, then it will be able to read its writes. However, if the network topology changes and the client can only execute T_2 on a different replica that is partitioned from the replica that executed T_1, then the system will have to either stall indefinitely to allow the client to read her writes (violating transactional availability) or will have to sacrifice read your writes guarantees. However, if the client remains sticky with the server that executed T_1, then we can disallow this scenario. Accordingly, read your writes, and, by proxy, causal consistency and PRAM require stickiness. Read your writes is provided by default in a sticky system. Causality and PRAM guarantees can be accomplished with well-known variants [3, 10, 47, 59, 66] of the prior session guarantee algorithms we presented earlier: only reveal new writes to clients when their (respective, model-specific) dependencies have been revealed.

5.1.4 Additional HAT Guarantees

In this section, we briefly discuss two additional kinds of guarantees that are achievable in HAT systems.

Consistency A HAT system can make limited application-level consistency guarantees. It can often execute commutative and logically monotonic [4] operations without the risk of invalidating application-level integrity constraints and can maintain limited criteria like foreign key constraints (via MAV). We do not describe the entire space of application-level consistency properties that are achievable (see Section 7) but we specifically evaluate TPC-C transaction semantics with HAT guarantees in Section 6.

Convergence Under arbitrary (but not infinite delays), HAT systems can ensure convergence, or *eventual consistency*: in the absence of new mutations to a data item, all servers should eventually agree on the value for each item [48, 63]. This is typically accomplished by any number of anti-entropy protocols, which periodically update neighboring servers with the latest value for each data item [30]. Establishing a final convergent value is related to determining a total order on transaction updates to each item, as in Read Uncommitted.

5.2 Unachievable HAT Semantics

While there are infinitely many HAT models (Section 7), at this point, we have largely exhausted the range of achievable, previously defined (and useful) semantics that are available to HAT systems. Before summarizing our possibility results, we will present impossibility results for HATs, also defined in terms of previously identified isolation and consistency anomalies. Most notably, it is impossible to prevent Lost Update or Write Skew in a HAT system.

5.2.1 *Unachievable ACID Isolation*

In this section, we demonstrate that preventing Lost Update and Write Skew—and therefore providing Snapshot Isolation, Repeatable Read, and one-copy serializability—inherently requires foregoing high availability guarantees.

Berenson et al. define *Lost Update* as when one transaction $T1$ reads a given data item, a second transaction $T2$ updates the same data item, then $T1$ modifies the data item based on its original read of the data item, "missing" or "losing" $T2$'s newer update. Consider a database containing only the following transactions:

$$T_1 : r_x(a)\ w_x(a+2)$$
$$T_2 : w_x(2)$$

If T_1 reads $a = 1$ but T_2's write to x precedes T_1's write operation, then the database will end up with $a = 3$, a state that could not have resulted in a serial execution due to T_2's "Lost Update."

It is impossible to prevent Lost Update in a highly available environment. Consider two clients who submit the following T_1 and T_2 on opposite sides of a network partition:

$$T_1 : r_x(100)\ w_x(100+20=120)$$
$$T_2 : r_x(100)\ w_x(100+30=130)$$

Regardless of whether $x = 120$ or $x = 130$ is chosen by a replica, the database state could not have arisen from a serial execution of T_1 and T_2.[4] To prevent this, either T_1 or T_2 should not have committed. Each client's respective server might try to detect that another write occurred, but this requires knowing the version of the latest write to x. In our example, this reduces to a requirement for linearizability, which is, via Gilbert and Lynch's proof of the CAP Theorem, provably at odds with high availability [34].

Write Skew is a generalization of Lost Update to multiple keys. It occurs when one transaction $T1$ reads a given data item x, a second transaction $T2$ reads a different data item y, then $T1$ writes to y and commits and $T2$ writes to x and commits. As an example of Write Skew, consider the following two transactions:

$$T_1 : r_y(0)\ w_x(1)$$
$$T_2 : r_x(0)\ w_y(1)$$

As Berenson et al. describe, if there was an integrity constraint between x and y such that only one of x or y should have value 1 at any given time, then this write skew would violate the constraint (which is preserved in serializable executions). Write skew is a somewhat esoteric anomaly—for example, it does not appear in TPC-C [33]—but, as a generalization of Lost Update, it is also unavailable to HAT systems.

Consistent Read, Snapshot Isolation (including Parallel Snapshot Isolation [58]), and Cursor Stability guarantees are all unavailable because they require preventing Lost Update phenomena. Repeatable Read (defined by Gray [37], Berenson et al. [11], and Adya [2]) and One-Copy Serializability [6] need to prevent both Lost Update and Write Skew. Their prevention requirements mean that these guarantees are inherently unachievable in a HAT system.

[4] In this example, we assume that, as is standard in modern databases, replicas accept values as they are written (i.e., register semantics). This particular example could be made serializable via the use of commutative updates (Section 6) but the problem persists in the general case.

5.2.2 *Unachievable Recency Guarantees*

Distributed data storage systems often make various recency guarantees on reads of data items. Unfortunately, an indefinitely long partition can force an available system to violate any recency bound, so recency bounds are not enforceable by HAT systems [34]. One of the most famous of these guarantees is linearizability [40], which states that reads will return the last completed write to a data item, and there are several other (weaker) variants such as safe and regular register semantics. When applied to transactional semantics, the combination of one-copy serializability and linearizability is called *strong (or strict) one-copy serializability* [2] (e.g., Spanner [24]). It is also common, particularly in systems that allow reading from masters and slaves, to provide a guarantee such as "read a version that is no more than five seconds out of date" or similar. None of these guarantees are HAT-compliant.

5.2.3 *Durability*

A client requiring that its transactions' effects survive F server faults requires that the client be able to contact at least $F+1$ non-failing replicas before committing. This affects availability and, according to the Gilbert and Lynch definition we have adopted, $F >$ 1 fault tolerance is not achievable with high availability.

5.3 Summary

As we summarize in Table 3, a wide range of isolation levels are achievable in HAT systems. With sticky availability, a system can achieve read your writes guarantees and PRAM and causal consistency. However, many other prominent semantics, such as Snapshot Isolation, One-Copy Serializability, and Strong Serializability cannot be achieved due to the inability to prevent Lost Update and Write Skew phenomena.

We illustrate the hierarchy of available, sticky available, and unavailable consistency models we have discussed in Figure 2. Many models are simultaneously achievable, but we find several particularly compelling. If we combine all HAT and sticky guarantees, we have transactional, causally consistent snapshot reads (i.e., Causal Transactional Predicate Cut Isolation). If we combine MAV and P-CI, we have transactional snapshot reads. We can achieve RC, MR, and RYW by simply sticking clients to servers. We can also combine unavailable models—for example, an unavailable system might provide PRAM and One-Copy Serializability [26].

To the best of our knowledge, this is the first unification of transactional isolation, distributed consistency, and session guarantee models. Interestingly, strong one-copy serializability entails all other models, while considering the (large) power set of all compatible models (e.g., the diagram depicts 144 possible HAT combinations) hints at the vast expanse of consistency models found in the literature. This taxonomy is not exhaustive (Section 8), but we believe it lends substantial clarity to the relationships between a large subset of the prominent ACID and distributed consistency models. Additional read/write transaction semantics that we have omitted should be classifiable based on the available primitives and HAT-incompatible anomaly prevention we have already discussed.

In light the of current practice of deploying weak isolation levels (Section 3), it is perhaps surprising that so many weak isolation levels are achievable as HATs. Indeed, isolation levels such as Read Committed expose and are defined in terms of end-user anomalies that could not arise during serializable execution. However, the prevalence of these models suggests that, in many cases, applications can tolerate these their associated anomalies. Given our HAT-compliance results, this in turn hints that–despite idiosyncrasies relating to concurrent updates and data recency–highly available database systems can provide sufficiently strong semantics for many applications. Indeed, HAT databases may expose more anomalies

HA	Read Uncommitted (RU), Read Committed (RC), Monotonic Atomic View (MAV), Item Cut Isolation (I-CI), Predicate Cut Isolation (P-CI), Writes Follow Reads (WFR), Monotonic Reads (MR), Monotonic Writes (MW)
Sticky	Read Your Writes (RYW), PRAM, Causal
Unavailable	Cursor Stability (CS)†, Snapshot Isolation (SI)†, Repeatable Read (RR)†‡, One-Copy Serializability (1SR)†‡, Recency$^{\oplus}$, Safe$^{\oplus}$, Regular$^{\oplus}$, Linearizability$^{\oplus}$, Strong 1SR$^{\dagger\ddagger\oplus}$

Table 3: Summary of highly available, sticky available, and unavailable models considered in this paper. Unavailable models are labeled by cause of unavailability: preventing lost update†, preventing write skew‡, and requiring recency guarantees$^{\oplus}$.

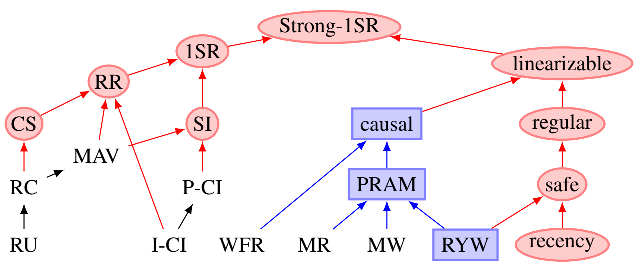

Figure 2: Partial ordering of HAT, sticky available (in boxes, blue), and unavailable models (circled, red) from Table 3. Directed edges represent ordering by model strength. Incomparable models can be simultaneously achieved, and the availability of a combination of models has the availability of the least available individual model.

than a single-site database operating under weak isolation (particularly during network partitions). However, for a fixed isolation level (which, in practice, can vary across databases and may differ from implementation-agnostic definitions in the literature), users of single-site database are subject to the same (worst-case) application-level anomalies as a HAT implementation. The necessary (indefinite) visibility penalties (i.e., the right side of Figure 2) and lack of support for preventing concurrent updates (via the upper left half of Figure 2) mean HATs are *not* well-suited for all applications (see Section 6): these limitations are fundamental. However, common practices such as ad-hoc, user-level compensation and per-statement isolation "upgrades" (e.g., `SELECT FOR UPDATE` under weak isolation)—commonly used to augment weak isolation—are also applicable in HAT systems (although they may in turn compromise availability).

6 HAT Implications

With an understanding of which semantics are HAT-compliant, in this section, we analyze the implications of these results for existing systems and briefly study HAT systems on public cloud infrastructure. Specifically:

1. We revisit traditional database concurrency control with a focus on coordination costs and on high availability.
2. We examine the properties required by an OLTP application based on the TPC-C benchmark.
3. We perform a brief experimental evaluation of HAT versus non-HAT properties on public cloud infrastructure.

6.1 HA and Existing Algorithms

While we have shown that many database isolation levels are achievable as HATs, many traditional concurrency control mechanisms do not provide high availability—even for HAT-compliant isolation levels. Existing mechanisms often presume (or are adapted from) single-server non-partitioned deployments or otherwise focus on serializability as a primary use case. In this section, we briefly discuss design decisions and algorithmic details that preclude high availability.

Serializability To establish a serial order on transactions, algorithms for achieving serializability of general-purpose read-write transactions in a distributed setting [13, 27] require at least one RTT before committing. As an example, traditional two-phase locking for a transaction of length T may require T `lock` operations and will require at least one `lock` and one `unlock` operation. In a distributed environment, each of these lock operations requires coordination, either with other database servers or with a lock service. If this coordination mechanism is unavailable, transactions cannot safely commit. Similarly, optimistic concurrency control requires coordinating via a validation step, while deterministic transaction scheduling [55] requires contacting a scheduler. Serializability under multi-version concurrency control requires checking for update conflicts. All told, the reliance on a globally agreed total order necessitates a minimum of one round-trip to a designated master or coordination service for each of these classic algorithms. As we saw in Section 2, is will be determined by the deployment environment; we will further demonstrate this in Section 6.3.

Non-serializability Most existing distributed implementations of weak isolation are not highly available. Lock-based mechanisms such as those in Gray's original proposal [37] do not degrade gracefully in the presence of partial failures. (Note, however, that lock-based protocols *do* offer the benefit of recency guarantees.) While multi-versioned storage systems allow for a variety of transactional guarantees, few offer traditional weak isolation (e.g., non-"tentative update" schemes) in this context. Chan and Gray's read-only transactions have item-cut isolation with causal consistency and MAV (session *PL-2L* [2]) but are unavailable in the presence of coordinator failure and assume serializable update transactions [19]; this is similar to read-only and write-only transactions more recently proposed by Eiger [47]. Brantner's S3 database [14] and Bayou [59] can all provide variants of session *PL-2L* with high availability, but none provide this HAT functionality without substantial modification. Accordingly, it is possible to implement many guarantees weaker than serializability—including HAT semantics—and still not achieve high availability. We view high availability as a core design consideration in future concurrency control designs.

6.2 Application Requirements

Thus far, we have largely ignored the question of when HAT semantics are useful (or otherwise are too weak). As we showed in Section 5, the main cost of high availability and low latency comes in the inability to prevent Lost Update, Write Skew, and provide recency bounds. To better understand the impact of HAT-compliance in an application context, we consider a concrete application: the TPC-C benchmark. In brief, we find that four of five transactions can be executed via HATs, while the fifth requires unavailability.

TPC-C consists of five transactions, capturing the operation of a wholesale warehouse, including sales, payments, and deliveries. Two transactions—*Order-Status* and *Stock-Level*—are read-only and can be executed safely with HATs. Clients may read stale data, but this does not violate TPC-C requirements and clients will read their writes if they are sticky-available. Another transaction type, *Payment*, updates running balances for warehouses, districts, and customer records and provides an audit trail. The transaction is monotonic—increment- and append-only—so all balance increase operations commute, and MAV allows the maintenance of foreign-key integrity constraints (e.g., via `UPDATE/DELETE CASCADE`).

New-Order and Delivery. While three out of five transactions are easily achievable with HATs, the remaining two transactions are not as simple. The New-Order transaction places an order for a variable quantity of data items, updating warehouse stock as needed. It selects a sales district, assigns the order an ID number, adjusts the remaining warehouse stock, and writes a placeholder entry for the pending order. The Delivery transaction represents the fulfillment of a New-Order: it deletes the order from the pending list, updates the customer's balance, updates the order's carrier ID and delivery time, and updates the customer balance.

IDs and decrements. The New-Order transaction presents two challenges: ID assignment and stock maintenance. First, each New-Order transaction requires a unique ID number for the order. We can create a unique number by, say, concatenating the client ID and a timestamp. However, the TPC-C specification requires order numbers to be *sequentially* assigned within a district, which requires preventing Lost Update. Accordingly, HATs cannot provide compliant TPC-C execution but can maintain uniqueness constraints. Second, the New-Order transaction decrements inventory counts: what if the count becomes negative? Fortunately, New-Order restocks each item's inventory count (increments by 91) if it would become negative as the result of placing an order. This means that, even in the presence of concurrent New-Orders, an item's stock will never fall below zero. This is TPC-C compliant, but a HAT system might end up with more stock than in an unavailable implementation with synchronous coordination.

TPC-C Non-monotonicity. The Delivery transaction is challenging due to non-monotonicity. Each Delivery deletes a pending order from the New-Order table and should be idempotent in order to avoid billing a customer twice; this implies a need to prevent Lost Update. This issue can be avoided by moving the non-monotonicity to the real world—the carrier that picks up the package for an order can ensure that no other carrier will do so—but cannot provide a correct execution with HATs alone. However, according to distributed transaction architects [39], these compensatory actions are relatively common in real-world business processes.

Integrity Constraints. Throughout execution, TPC-C also requires the maintenance of several integrity constraints. For example, Consistency Condition 1 (3.3.2.1) requires that each warehouse's sales count must reflect the sum of its subordinate sales districts. This integrity constraint spans two tables but, given the ability to update rows in both tables atomically via MAV, can be easily maintained. Consistency Conditions 4 through 12 (3.3.2.4-12) can similarly be satisfied by applying updates atomically across tables. Consistency Conditions 2 and 3 (3.3.2.2-3) concern order ID assignment and are problematic. Finally, while TPC-C is not subject to multi-key anomalies, we note that many TPC-E isolation tests (i.e., simultaneously modifying a product description and its thumbnail) are also achievable using HATs.

Summary. Many—but not all—TPC-C transactions are well served by HATs. The two problematic transactions, New-Order and Payment, rely on non-monotonic state update. The former can be modified to ensure ID uniqueness but not sequential ID ordering, while the latter is inherently non-monotonic, requiring external compensation or stronger consistency protocols. Based on these experiences and discussions with practitioners, we believe that HAT guarantees can provide useful semantics for a large class of application functionality, while a (possibly small) subset of operations will require stronger, unavailable properties.

6.3 Experimental Costs

To demonstrate the performance implications of HAT guarantees in a real-world environment, we implemented a HAT database prototype. We verify that, as Section 2.2's measurements suggested, "strongly consistent" algorithms incur substantial latency penalties (over WAN, 10 to 100 times higher than their HAT counterparts) compared to HAT-compliant algorithms, which scale linearly. Our goal is *not* a complete performance analysis of HAT semantics but instead a proof-of-concept demonstration of a small subset of the HAT space on real-world infrastructure.

Implementation. Our prototype database is a partially replicated (hash-based partitioned) key-value backed by LevelDB and implemented in Java using Apache Thrift. It currently supports eventual consistency (hereafter, `eventual`; last-writer-wins RU with standard all-to-all anti-entropy between replicas) and the efficient HAT MAV algorithm as sketched in Section 5.1.2. (hereafter, `MAV`). We support non-HAT operation whereby all operations for a given key are routed to a (randomly) designated `master` replica for each key (guaranteeing single-key linearizability, as in Gilbert and Lynch's CAP Theorem proof [34] and in PNUTS [22]'s "read latest" operation; hereafter, `master`) as well as distributed two-phase locking. Servers are durable: they synchronously write to LevelDB before responding to client requests, while new writes in MAV are synchronously flushed to a disk-resident write-ahead log.

Configuration. We deploy the database in *clusters*—disjoint sets of database servers that each contain a single, fully replicated copy of the data—typically across datacenters and stick all clients within a datacenter to their respective cluster (trivially providing read-your-writes and monotonic reads guarantees). By default, we deploy 5 Amazon EC2 `m1.xlarge` instances as servers in each cluster. For our workload, we link our client library to the YCSB benchmark [23], which is well suited to LevelDB's key-value schema, grouping every eight YCSB operations from the default workload (50% reads, 50% writes) to form a transaction. We increase the number of keys in the workload from the default 1,000 to 100,000 with uniform random key access, keeping the default value size of $1KB$, and running YCSB for 180 seconds per configuration.

Geo-replication. We first deploy the database prototype across an increasing number of datacenters. Figure 3A shows that, when operating two clusters within a single datacenter, mastering each data item results in approximately half the throughput and double the latency of `eventual`. This is because HAT models are able to utilize replicas in both clusters instead of always contacting the (single) master. `RC`—essentially `eventual` with buffering—is almost identical to `eventual`, while `MAV`—which incurs two writes for every client-side write (i.e., new writes are sent to the WAL then subsequently moved into LevelDB once stable)—achieves 75% of the throughput. Latency increases linearly with the number of YCSB clients due to contention within LevelDB.

In contrast, when the two clusters are deployed across the continental United States (Figure 3B), the average latency of `master` increases to 300ms (a 278–4257% latency increase; average 37ms latency per operation). For the same number of YCSB client threads, `master` has substantially lower throughput than the HAT configurations. Increasing the number of YCSB clients *does* increase the throughput of `master`, but our Thrift-based server-side connection processing did not gracefully handle more than several thousand concurrent connections. In contrast, across two datacenters, the performance of eventual, RC, and MAV are near identical to a single-datacenter deployment.

When five clusters (as opposed to two, as before) are deployed across the five EC2 datacenters with lowest communication cost

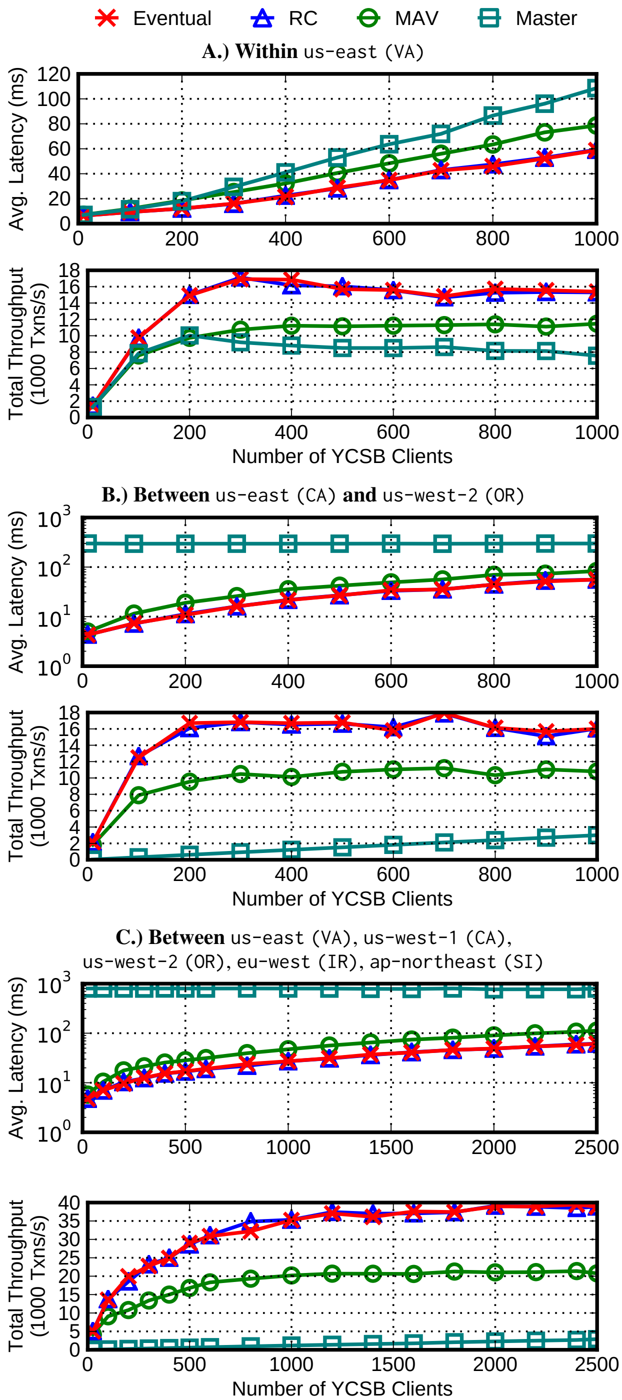

Figure 3: YCSB performance for two clusters of five servers each deployed within a single datacenter and cross-datacenters.

(Figure 3C), the trend continues: `master` latency increases to nearly 800ms per transaction. As an attempt at reducing this overhead, we implemented and benchmarked a variant of quorum-based replication as in Dynamo [29], where clients sent requests to all replicas, which completed as soon as a majority of servers responded (guaranteeing regular semantics [40]); this strategy (not pictured) did not substantially improve performance due to the network topology and because worst-case server load was unaffected. With five clusters, `MAV`'s relative throughput decreased: every YCSB `put` operation resulted in four `put` operations on remote replicas and, accordingly, the cost of anti-entropy increased (e.g., each server processed four replicas' anti-entropy as opposed to one before, reducing the opportunity for batching and decreasing available resources for incoming client requests). This in turn increased garbage collection activity and, more importantly, IOPS when compared to `eventual` and `RC`, causing `MAV` throughput to peak at around half of `eventual`. With in-memory persistence (i.e., no LevelDB or WAL), `MAV` throughput was within 20% of `eventual`.

We have intentionally omitted performance data for two-phase locking. `master` performed *far* better than our textbook implementation, which, in addition to requiring a WAN round-trip per operation, also incurred substantial overheads due to mutual exclusion via locking. We expect that, while techniques like those recently proposed in Calvin [60] can reduce the overhead of serializable transactions by avoiding locking, our mastered implementation and the data from Section 2.2 are reasonable lower bounds on latency.

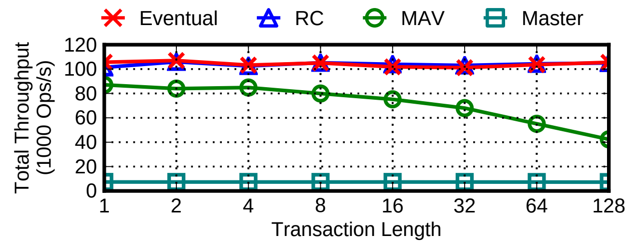

Figure 4: Transaction length versus throughput.

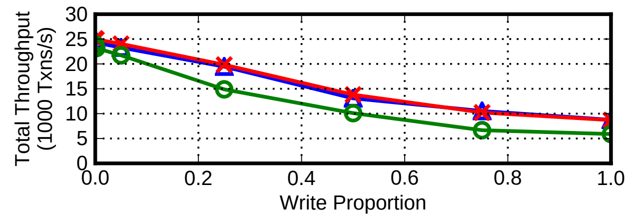

Figure 5: Proportion of reads and writes versus throughput.

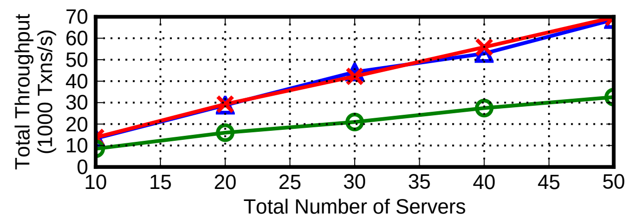

Figure 6: Scale-out of MAV, Eventual, and RC.

Transaction length. As shown in Figure 4 (clusters in Virginia and Oregon), throughput of `eventual`, `RC`, and `master` operation are unaffected by transaction length. In contrast, `MAV` throughput decreases linearly with increased transaction length: with 1 operation per transaction, `MAV` throughput is within 18% of `eventual` (34 bytes overhead), and with 128 operations per transaction, `MAV` throughput is within 60% (1898 bytes overhead). This reflects our `MAV` algorithm's metadata requirements, which are proportional to transaction length and consume IOPS and network bandwidth. We are currently investigating alternative HAT algorithms that do not incur this overhead.

Read proportion. Our default (equal) proportion of reads and writes is fairly pessimistic: for example, Facebook reports 99.8% reads for their workload [47]. As shown in Figure 5 (clusters in Virginia and Oregon), with all reads, `MAV` is within 4.8% of `eventual`; with all writes, `MAV` is within 33%, and the throughput of `eventual` decreases by 288.8% compared to all reads. At 99.8% reads, MAV incurs a 7% overhead (5.8% for in-memory storage).

Scale-out. One of the key benefits of our HAT algorithms is that they are shared-nothing, meaning they should not compromise scalability. Figure 6 shows that varying the number of servers across two clusters in Virginia and Oregon (with 15 YCSB clients per server) results in linear scale-out for eventual, RC, and MAV. RC and eventual scale linearly: increasing the number of servers per cluster from 5 to 25 yields an approximately 5x throughput increase. For the same configurations, MAV scales by 3.8x, achieving over $260,000$ operations per second. MAV suffers from contention in LevelDB—with a memory-backed database, MAV scales by 4.25x (not shown)—and MAV-related performance heterogeneity across servers (Calvin's authors report similar heterogeneity on EC2 [60]). Initial experiments with a newer prototype including more efficient (non-Thrift) RPC and connection pooling suggest that this scalability can be substantially improved.

Summary. Our experimental prototype confirms our earlier analytical intuitions. HAT systems can provide useful semantics without substantial performance penalties. In particular, our MAV algorithm can achieve throughput competitive with eventual consistency at the expense of increased disk and network utilization. Perhaps more importantly, all HAT algorithms circumvent high WAN latencies inevitable with non-HAT implementations. Our results highlight Deutsch's observation that ignoring factors such as latency can "cause big trouble and painful learning experiences" [31]—in a single-site context, paying the cost of coordination may be tenable, but, especially as services are geo-replicated, costs increase.

7 Related Work

We have discussed traditional mechanisms for distributed coordination and several related systems in Section 6.1, but, in this section, we further discuss related work. In particular, we discuss related work on highly available semantics, mechanisms for concurrency control, and techniques for scalable distributed operations.

Weak consistency and high availability have been well studied. Serializability has long been known to be unachievable [27] and Brewer's CAP Theorem has attracted considerable attention [34]. Recent work on PACELC expands CAP by considering connections between "weak consistency" and low latency [1], while several studies examine weak isolation guarantees [2, 11]. There are a wide range of coordination-avoiding "optimistic replication" strategies [54] and several recent efforts at further understanding these strategies in light of current practice and the proliferation of "eventually consistent" stores [9, 12]. Notably, Bernstein and Das [12] specifically highlight the importance of stickiness [59, 63]—which we formalize in Section 4.1. Aside from our earlier workshop paper discussing transactional availability, real-world ACID, and HAT RC and I-CI [8]—which this work expands with additional semantics, algorithms, and analysis—we believe this paper is the first to explore the connections between transactional semantics, data consistency, and (Gilbert and Lynch [34]) availability.

There has been a recent resurgence of interest in distributed multi-object semantics, both in academia [14, 25, 47, 58, 60, 66] and industry [18, 24]. As discussed in Section 3, classic ACID databases provide strong semantics but their lock-based and traditional multi-versioned implementations are unavailable in the presence of partitions [13, 37]. Notably, Google's Spanner provides strong one-copy serializable transactions. While Spanner is highly specialized for Google's read-heavy workload, it relies on two-phase commit and two-phase locking for read/write transactions [24]. As we have discussed, the penalties associated with this design are fundamental to serializability. For users willing to tolerate unavailability and increased latency, Spanner, or similar "strongly consistent" systems [64]—including Calvin [60], G-Store [25], HBase, HStore [41], Orleans [18], Postgres-R [42], Walter [58], and a range of snapshot isolation techniques [26]—reasonable choices.

With HATs, we seek an alternative set of transactional semantics that are still useful but do not violate requirements for high availability or low latency. Recent systems proposals such as Swift [66], Eiger [47], and Bolt-on Causal Consistency [10] provide transactional causal consistency guarantees with varying availability and represent a new class of sticky HAT systems. There are infinitely many HAT models (i.e., always reading value 1 is incomparable with always returning value 2), but a recent report from UT Austin shows that no model stronger than causal consistency is achievable in a sticky highly available, *one-way convergent* system [48]. This result is promising and complementary to our results for general-purpose convergent data stores. Finally, Burkhardt et al. have concurrently developed an axiomatic specification for eventual consistency; their work-in-progress report contains alternate formalism for several HAT guarantees [17].

8 Conclusions and Future Work

The current state of database software offers uncomfortable and unnecessary choices between availability and transactional semantics. Through our analysis and experiments, we have demonstrated how goals of high availability will remain a critical aspect of many future data storage systems. We expose a broad design space of Highly Available Transactions (HATs), which can offer the key benefits of highly available distributed systems—"always on" operation during partitions and low-latency operations (often orders of magnitude lower than non-compliant)—while also providing a family of transactional isolation levels and replica semantics that have been adopted in practice. We also identify many semantic guarantees that are unachievable with high availability, including Lost Update and Write Skew anomaly prevention, concurrent update prevention, and bounds on data recency. Despite these limitations, and somewhat surprisingly, many of the default (and sometimes strongest) semantics provided by today's traditional database systems are achievable as HATs, hinting that distributed databases need not compromise availability, low latency, or scalability in order to serve many existing applications.

In this paper, we have largely focused on previously defined isolation and data consistency models from the consistency from the database and distributed systems communities. Their previous definitions and, in many cases, widespread adoption hints at their utility to end-users. However, we believe there is considerable work to be done to improve the programmability of highly available systems. Isolation and data consistency are means by which *application-level* consistency is achieved but are typically not end goals for end-user applications. Our results hint that a mix of HAT and non-HAT semantics (the latter used sparingly) is required for practical applications, but the decision to employ each and the system architectures for a hybrid approach remain open problems. While we have studied the analytical and experiment behavior of several HAT models, there is substantial work in further understanding the performance and design of systems within the large set of HAT models. Weakened failure assumptions as in escrow or in the form of bounded network asynchrony could enable richer HAT semantics at the cost of general-purpose availability. Alternatively, there is a range of possible solutions providing strong semantics during partition-free periods and weakened semantics during partitions. Based on our understanding of what is desired in the field and new-found knowledge of what is possible to achieve, we believe HATs represent a large and useful design space for exploration.

Acknowledgments We would like to thank Peter Alvaro, Neil Conway, Evan Jones, Adam Oliner, Aurojit Panda, Shivaram Venkataraman, and the HotOS and VLDB reviewers for their helpful feedback on this work. This research was supported in part by the Air Force Office of Scientic Research (grant FA95500810352), DARPA XData Award FA8750-12-2-0331, the National Science Foundation (grants CNS-0722077, IIS-0713661, and IIS-0803690), NSF CISE Expeditions award CCF-1139158, the National Science Foundation Graduate Research Fellowship (grant DGE-1106400), and by gifts from Amazon Web Services, Google, SAP, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, FitWave, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Samsung, Splunk, VMware, WANdisco, and Yahoo!.

9 References

[1] D. J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. IEEE Computer, 45(2), 2012.

[2] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, MIT, 1999.

[3] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation and programming. *Dist. Comp.*, 9(1), 1995.

[4] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR 2011*.

[5] ISO/IEC 9075-2:2011 *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*.

[6] R. Attar, P. A. Bernstein, and N. Goodman. Site initialization, recovery, and backup in a distributed database system. *IEEE Trans. Softw. Eng.*, 10(6):645–650, Nov. 1984.

[7] AWS. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. `http://tinyurl.com/6ab6el6`, April 2011.

[8] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. HAT, not CAP: Introducing Highly Available Transactions. In *HotOS 2013*.

[9] P. Bailis and A. Ghodsi. Eventual Consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3), March 2013.

[10] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *SIGMOD 2013*.

[11] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD 1995*.

[12] P. Bernstein and S. Das. Rethinking eventual consistency. In *SIGMOD*, 2013.

[13] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[14] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD 2008*.

[15] E. Brewer. Towards robust distributed systems. Keynote at PODC 2000.

[16] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *PDP 2004*.

[17] S. Burckhardt, A. Gotsman, and H. Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39. `http://tinyurl.com/bqty9yz`.

[18] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *SOCC 2011*.

[19] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Transactions on Software Engineering*, (2):205–212, 1985.

[20] F. Chang, J. Dean, S. Ghemawat, et al. Bigtable: A distributed storage system for structured data. In *OSDI 2006*.

[21] G. Clarke. The Register: NoSQL's CAP theorem busters: We don't drop ACID. `http://tinyurl.com/bpsug4b`, November 2012.

[22] B. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB 2008*.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM SOCC 2010*.

[24] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, et al. Spanner: Google's globally-distributed database. In *OSDI 2012*.

[25] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SOCC 2010*, pages 163–174.

[26] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *ICDE 2004*, pages 424–435.

[27] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM CSUR*, 17(3):341–370, 1985.

[28] J. Dean. Designs, lessons and advice from building large distributed systems. Keynote at LADIS 2009.

[29] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, et al. Dynamo: Amazon's highly available key-value store. In *SOSP 2007*.

[30] A. Demers et al. Epidemic algorithms for replicated database maintenance. In *PODC*, 1987.

[31] P. Deutsch. The eight fallacies of distributed computing. `http://tinyurl.com/c6vvtzg`, 1994.

[32] R. Dillet. Update: Amazon Web Services down in North Virginia Reddit, Pinterest, Airbnb, Foursquare, Minecraft and others affected. TechCrunch `http://tinyurl.com/9r43dwt`, October 2012.

[33] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM TODS*, 30(2):492–528, June 2005.

[34] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[35] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM 2011*.

[36] J. Gray. The transaction concept: Virtues and limitations. In *VLDB 1981*.

[37] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared data base. Technical report, IBM, 1976.

[38] J. Hamilton. Stonebraker on CAP Theorem and Databases. `http://tinyurl.com/d3gtfq9`, April 2010.

[39] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR 2007*.

[40] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. 2008.

[41] R. Kallman et al. H-store: a high-performance, distributed main memory transaction processing system. In *VLDB 2008*.

[42] B. Kemme. *Database replication for clusters of workstations*. PhD thesis, EPFL, 2000.

[43] K. Kingsbury and P. Bailis. The network is reliable. June 2013. `http://aphyr.com/posts/288-the-network-is-reliable`.

[44] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental study of internet stability and backbone failures. In *FTCS 1999*.

[45] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[46] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: Fault tolerant engineered networks. In *NSDI 2013*.

[47] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI 2013*.

[48] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, convergence. Technical Report TR-11-22, CS Department, UT Austin, May 2011.

[49] A. Markopoulou et al. Characterization of failures in an operational IP backbone network. *IEEE/ACM TON*, 16(4).

[50] D. McCullagh. How Pakistan knocked YouTube offline (and how to make sure it never happens again). CNET, `http://tinyurl.com/c4pffd`, February 2008.

[51] R. McMillan. Research experiment disrupts internet, for some. Computerworld, `http://tinyurl.com/23sqpek`, August 2010.

[52] P. P. S. Narayan. Sherpa update. YDN Blog, `http://tinyurl.com/c3ljuce`, June 2010.

[53] F. Pedone and R. Guerraoui. On transaction liveness in replicated databases. In *Pacific Rim International Symposium on Fault-Tolerant Systems*, 1997.

[54] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1), Mar. 2005.

[55] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *CACM*, 39(4), 1996.

[56] L. Segall. Internet routing glitch kicks millions offline. CNNMoney, `http://tinyurl.com/cmqqac3`, November 2011.

[57] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. INRIA TR 7506, 2011.

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

[59] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, et al. Session guarantees for weakly consistent replicated data. In *PDIS 1994*.

[60] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD 2012*.

[61] D. Turner, K. Levchenko, J. C. Mogul, S. Savage, and A. C. Snoeren. On failure in managed enterprise networks. HP Labs HPL-2012-101, 2012.

[62] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: understanding the causes and impact of network failures. *SIGCOMM 2011*.

[63] W. Vogels. Eventually consistent. *CACM*, 52(1):40–44, Jan. 2009.

[64] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *SRDS 2000*.

[65] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: avoiding long tails in the cloud. In *NSDI*, 2013.

[66] M. Zawirski, A. Bieniusa, V. Balegas, N. Preguica, S. Duarte, M. Shapiro, and C. Baquero. Geo-replication all the way to the edge. Personal communication and draft under submission. `http://tinyurl.com/cp68svy`.

APPENDIX

A Formal Definitions

In this section, we formally define HAT transactional semantics. Our formalism is based off of that of Adya [2]. For the reader familiar with his formalism, this is a mostly-straightforward exercise combining transactional models with distributed systems semantics. While the novel aspects of this work largely pertain to *using* these definitions (e.g., in Section 5), we believe it is instructive to accompany them by appropriate definitions to both eliminate any ambiguity in prose and for the enjoyment of more theoretically-inclined readers.

A.1 Model

Here, we briefly describe our database model. It is, with the exception of sessions, identical to that of Adya [2]. We omit a full duplication of his formalism here but highlight several salient criteria. We refer the interested reader to Adya's Ph.D. thesis, Section 3.1 (pp. 33–43).

Users submit transactions to a database system that contains multiple versions of sets of objects. Each transaction is composed of writes, which create new versions of an object, and reads, which return a written version or the initial version of the object. A transaction's last operation is either `commit` or `abort`, and there is exactly one invocation of either these two operations per transaction. Transactions can either read individual items or read based on predicates, or logical ranges of data items.

Definition 1 (Version set of a predicate-based operation). When a transaction executes a read or write based on a predicate P, the system selects a version for each tuple in Ps relations. The set of selected versions is called the *Version set* of this predicate-based operation and is denoted by Vset(P).

A history over transactions has two parts: a partial order of events reflects the ordering of operations with respect to each transaction and a (total) version order ($\ll$) on the committed versions of each object.

As a departure from Adya's formalism, to capture the use of session guarantees, we allow transactions to be grouped into sessions. We represent sessions as a partial ordering on committed transactions such that each transaction in the history appears in at most one session.

A.2 Conflict and Serialization Graphs

To reason about isolation anomalies, we use Adya's concept of a conflict graph, which is composed of dependencies between transaction. The definitions in this section are directly from Adya, with two differences. First, we expand Adya's formalism to deal with *per-item* dependencies. Second, we define session dependencies (Definition 15).

Definition 2 (Change the Matches of a Predicate-Based Read). A transaction T_i changes the matches of a predicate-based read r_j(P:Vset(P)) if T_i installs x_i, x_h immediately precedes x_i in the version order, and x_h matches P whereas x_i does not or vice-versa. In this case, we also say that x_i changes the matches of the predicate-based read. The above denition identies T_i to be a transaction where a change occurs for the matched set of r_j (P: Vset(P)).

Definition 3 (Directly Read-Depends [2, Definition 2]). T_j directly read-depends on transaction T_i if it directly item-read-depends or directly predicate-read-depends on T_i.

Definition 4 (Directly item-read-depends by x). T_j directly item-read-depends on transaction T_i if T_i installs some object version x_i and T_j reads x_i.

Definition 5 (Directly item-read-depends). T_j directly item-read-depends on transaction T_i if T_j directly item-read-depends by x on T_i for some data item x.

Definition 6 (Directly predicate-read-depends by P). Transaction T_j directly predicate-read-depends by P on transaction T_i if T_j performs an operation r_j(P: Vset(P)), $x_k \in$ Vset(P), $i = k$ or $x_i \ll x_k$, and x_i changes the matches of r_j (P: Vset(P)).

Definition 7 (Directly predicate-read-depends). T_j directly predicate-read-depends on T_i if T_j directly predicate-read-depends by P on T_i for some predicate P.

Definition 8 (Directly Anti-Depends [2, Definition 4]). Transaction T_j directly anti-depends on transaction T_i if it directly item-anti-depends or directly predicate-anti-depends on T_i.

Definition 9 (Directly item-anti-depends by x). T_j directly item-anti-depends by x on transaction T_i if T_i reads some object version x_k and T_j installs x's next version (after x_k) in the version order. Note that the transaction that wrote the later version directly item-anti-depends on the transaction that read the earlier version.

Definition 10 (Directly item-anti-depends). T_j directly item-anti-depends on transaction T_i if T_j directly item-anti-depends on transaction T_i.

Definition 11 (Directly predicate-anti-depends by P). T_j directly predicate-anti-depends by P on transaction T_i if T_j overwrites an operation $r_i(P:$ Vset(P)). That is, if T_j installs a later version of some object that changes the matches of a predicate based read performed by T_i.

Definition 12 (Directly predicate-anti-depends by P). T_j directly predicate-anti-depends on transaction T_i if T_j directly predicate anti-depends by P on T_i for some predicate P.

Definition 13 (Directly Write-Depends by x). A transaction T_j directly write-depends by x on transaction T_i if T_i installs a version x_i and T_j installs x's next version (after x_i) in the version order.

Definition 14 (Directly Write-Depends [2, Definition 5]). A transaction T_j directly write-depends on transaction T_i if T_i directly write-depends by x on T_j for some item x.

Definition 15 (Session-Depends). A transaction T_j session-depends on transaction T_i if T_i and T_j occur in the same session and T_i precedes T_j in the session commit order.

The dependencies for a history H form a graph called its Directed Serialization Graph ($DSG(H)$). If T_j directly write-depends on T_i by x, we draw $T_i \xrightarrow{ww_x} T_j$. If T_j read-depends on T_i by x, we draw $T_i \xrightarrow{rw_x} T_j$. If T_j directly anti-depends on transaction T_j by x, we draw $T_i \overset{wr_x}{\dashrightarrow} T_j$. If T_j session-depends on T_i in session S, we draw $T_i \xrightarrow{S} T_j$ [2, Definition 8].

We also consider the Unfolded Serialization Graph ($USG(H)$) that is a variation of the DSG. The USG is specied for the transaction of interest, T_i, and a history, H, and is denoted by $USG(H, T_i)$. For the USG, we retain all nodes and edges of the DSG except for T_i and the edges incident on it. Instead, we split the node for T_i into multiple nodes—one node for every read/write event in T_i. The edges are now incident on the relevant event of T_i.

$USG(H, T_i)$ is obtained by transforming $DSG(H)$ as follows:. For each node p ($p \neq T_i$) in $DSG(H)$, we add a node to $USG(H, T_i)$. For each edge from node p to node q in $DSG(H)$, where p and q are different from T_i, we draw a corresponding edge in $USG(H, T_i)$.

Now we add a node corresponding to every read and write performed by T_i. Any edge that was incident on T_i in the DSG is now incident on the relevant event of T_i in the USG. Finally, consecutive events in T_i are connected by *order edges*, e.g., if an action (e.g., SQL statement) reads object y_j and immediately follows a write on object x in transaction T_i, we add an order-edge from $w_i(x_i)$ to $r_i(y_j)$ [2, Section 4.2.1].

A.3 Transactional Anomalies and Isolation Levels

Following Adya, we define isolation levels according to possible *anomalies*–typically represented by cycles in the serialization graphs. Definitions **??**–38 are not found in Adya but are found (albeit not in this formalism) in Berenson et al. [11] and the literature on session guarantees [59, 63].

Definition 16 (Write Cycles (G0)). A history H exhibits phenomenon G0 if $DSG(H)$ contains a directed cycle consisting entirely of write-dependency edges.

Definition 17 (Read Uncommitted). A system that provides Read Uncommitted isolation prohibits phenomenon G0.

Definition 18 (Aborted Reads (G1a)). A history H exhibits phenomenon G1a if it contains an aborted transaction T_1 and a committed transaction T_2 such that T_2 has read some object (maybe via a predicate) modied by T_1.

Definition 19 (Intermediate Reads (G1b)). A history H exhibits phenomenon G1b if it contains a committed transaction T_2 that has read a version of object x (maybe via a predicate) written by transaction T_1 that was not T_1s nal modication of x.

Definition 20 (Circular Information Flow (G1c)). A history H exhibits phenomenon G1c if $DSG(H)$ contains a directed cycle consisting entirely of dependency edges.

Definition 21 (Read Committed). A system that provides Read Committed isolation prohibits phenomenon G0, G1a, G1b, and G1c.

Definition 22 (Item-Many-Preceders (IMP)). A history H exhibits phenomenon IMP if $DSG(H)$ contains a transaction T_i such that T_i directly item-read-depends by x on more than one other transaction.

$$T_1 : w_x(1)$$
$$T_2 : w_x(2)$$
$$T_3 : r_x(1)\ r_x(2)$$

Figure 7: Example of *IMP* anomaly.

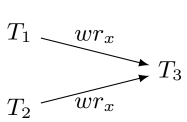

Figure 8: DSG for Figure 7.

Definition 23 (Item Cut Isolation (I-CI)). A system that provides Item Cut Isolation prohibits phenomenon IMP.

Definition 24 (Predicate-Many-Preceders (PMP)). A history H exhibits phenomenon PMP if, for all predicate-based reads $r_i(P_i : Vset(P_i))$ and $r_j(P_j : Vset(P_j)$ in T_k such that the logical ranges of P_i and P_j overlap (call it P_o), the set of transactions that change the matches of P_o for r_i and r_j differ.

Definition 25 (Predicate Cut Isolation (P-CI)). A system that provides Predicate Cut Isolation prohibits phenomenon PMP.

Definition 26 (Observed Transaction Vanishes (OTV)). A history H exhibits phenomenon OTV if $USG(H)$ contains a directed cycle consisting of exactly one read-dependency edge by x from T_j to T_i and a set of edges by y containing at least one anti-dependency edge from T_i to T_j and T_j's read from y precedes its read from x.

$$T_1 : w_x(1)\ w_y(1)$$
$$T_2 : w_x(2)\ w_y(2)$$
$$T_3 : r_x(2)\ r_y(1)$$

Figure 9: Example of *OTV* anomaly.

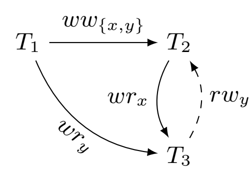

Figure 10: DSG for Figure 9.

Definition 27 (Monotonic Atomic View (MAV)). A system that provides Monotonic Atomic View isolation prohibits phenomenon OTV in addition to providing Read Committed isolation.

The following session guarantees are directly adapted from Terry et al.'s original definitions [59]:

Definition 28 (Non-monotonic Reads (N-MR)). A history H exhibits phenomenon N-MR if $DSG(H)$ contains a directed cycle consisting of a transitive session-dependency between transactions T_j and T_i with an anti-dependency edge by i from T_j and a read-dependency edge by i into T_i.

Definition 29 (Monotonic Reads (MR)). A system provides Monotonic Reads if it prohibits phenomenon N-MR.

$$T_1 : w_x(1)$$
$$T_2 : w_x(2)$$
$$T_3 : r_x(2)$$
$$T_4 : r_x(1)$$

Figure 11: Example of *N-MR* violation when $w_x(1) \ll w_x(2)$ and T_4 directly session-depends on T_3.

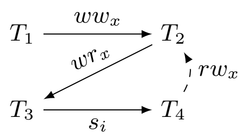

Figure 12: DSG for Figure 11. wr_x dependency from T_1 to T_4 omitted.

Definition 30 (Non-monotonic Writes (N-MW)). A history H exhibits phenomenon N-MW if $DSG(H)$ contains a directed cycle consisting of a transitive session-dependency between transactions T_j and T_i and at least one write-dependency edge.

$$T_1 : w_x(1)$$
$$T_2 : w_y(1)$$
$$T_3 : r_y(1)\ r_x(0)$$

Figure 13: Example of *N-MW* anomaly if T_2 directly session-depends on T_1.

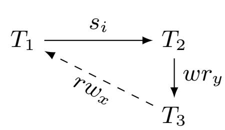

Figure 14: DSG for Figure 13.

Definition 31 (Monotonic Writes (MW)). A system provides Monotonic Writes if it prohibits phenomenon N-MW.

Definition 32 (Missing Read-Write Dependency (MRWD)). A history H exhibits phenomenon MRWD if, in $DSG(H)$, for all committed transactions T_1, T_2, T_3 such that T_2 write-depends on T_1 and T_3 write-depends on T_2, T_3 does not directly anti-depend on T_1.

$$T_1 : w_x(1)$$
$$T_2 : r_x(1) w_y(1)$$
$$T_3 : r_y(1)\ r_x(0)$$

Figure 15: Example of *MRWD* anomaly.

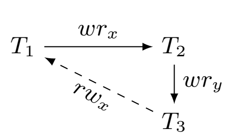

Figure 16: DSG for Figure 15.

Definition 33 (Writes Follow Reads (WFR)). A system provides Writes Follow Reads if it prohibits phenomenon MWRD.

Definition 34 (Missing Your Writes (MYR)). A history H exhibits phenomenon MYR if $DSG(H)$ contains a directed cycle consisting of a transitive session-dependency between transactions T_j and T_i, at least one anti-dependency edge, and the remainder anti-dependency or write-dependency edges.

$$T_1 : w_x(1)$$
$$T_2 : r_x(0)$$

Figure 17: Example of *MYR* anomaly if T_2 directly session-depends on T_1.

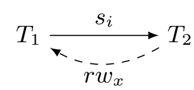

Figure 18: DSG for Figure 15.

Definition 35 (Read Your Writes (RYW)). A system provides Read Your Writes if it prohibits phenomenon MYR.

Definition 36 (PRAM Consistency). A system provides PRAM Consistency if it prohibits phenomenon N-MR, N-MW, and MYR.

Definition 37 (Causal Consistency). A system provides Causal Consistency if it provides PRAM Consistency and prohibits phenomenon MWRD.

Definition 38 (Lost Update). A history H exhibits phenomenon Lost if $DSG(H)$ contains a directed cycle having one or more item-antidependency edges and all edges are by the same data item x.

Definition 39 (Write Skew (Adya G2-item)). A history H exhibits phenomenon Write Skew if $DSG(H)$ contains a directed cycle having one or more item-antidependency edges.

For Snapshot Isolation, we depart from Adya's recency-based definition (see Adya Section 4.3). Nonetheless, implementations of this definition will still be unavailable due to reliance of preventing Lost Update.

Definition 40 (Snapshot Isolation). A system that provides Snapshot Isolation prevents phenomena G0, G1a, G1b, G1c, PMP, OTV, and Lost Update.

For Repeatable Read, we return to Adya.

Definition 41 (Repeatable Read). A system that provides Repeatable Read Isolation prohibits phenomena G0, G1a, G1b, G1c, and Write Skew.

For definitions of safe, regular, and linearizable register semantics, we refer the reader to Herlihy's textbook [40].

B Monotonic Atomic View Isolation

In this section, we provide additional information on our two-phase algorithm for providing Monotonic Atomic View isolation. This section is effectively a work-in-progress; we are actively investgating alternative algorithms (particularly with respect to metadata overheads) and stronger (but still HAT) semantics that complement the techniques presented here. The interested reader (or would-be systems implementer) should contact Peter Bailis for more details.

The below pseudocode describes a straightforward implementation of the algorithm from Section 5.1.2. Replicas maintain two sets of data items: `pending` (which contains writes that are not yet pending stable) and `good` (which contains writes that are pending stable). Incoming writes are added to `pending` and moved to `good` once they are stable. Stable pending calculation is performed by counting the number of acknowledgments (`acks`) of writes from other replicas, signifying their receipt of the transaction's writes in their own `pending`. Accordingly, the algorithm maintains the invariant that the transactional siblings (or suitable replacements) for every write in `good` are present on their respective replicas. Clients use the RC algorithm from Section 5.1.1 and keep a map of data items to timestamps (`required`)—effectively a vector clock whose entries are data items (not processes, as is standard).

Example. To illustrate this algorithm consider an execution of the following transactions:

$$T_1 : w_x(1)\ w_y(1)$$
$$T_2 : r_x(1)\ r_y(a)$$

For MAV, the system must ensure that $a = 1$. Consider a system with one replica for each of x and y (denoted R_x and R_y). Say client c_1 executes T_1 and commits. The client chooses timestamp 10001 (whereby the last 4 digits are reserved for client ID to ensure uniqueness) and sends values to $w_x(1)$ to R_x and R_y with $tx\_keys = \{x, y\}$ and $timestamp = 10001$. R_x and R_y place their respective writes into their individual `pending` sets and send acknowledgment for transaction timestamped 10001 to the other. Consider the case where r_x has seen r_y's acknowledgment but not vice versa: $w_x(1)$ will be in `good` on R_x but `pending` on R_y. Now, if another client c_2 executes T_2 and reads x from R_x, it will read $x = 1$. c_2 will update its `required` vector so that `required` = $\{x : 10001, y : 10001\}$. When c_2 reads y from R_y, it will specify $ts = 10001$ and R_y will not find a suitable write in `good`. R_y will instead return $y = 1$ from `pending`.

Overheads. MAV requires overhead both on client and server. First, clients must store a vector (`required`) for the duration of a transaction in order to determine what values to read. Second, each write contains metadata (tx_keys) linear in the number of writes in the transaction. (Note that we could have stored a different timestamp with every write, but a single timestamp suffices and is more efficient) These overheads are modest for short transaction lengths but they increase value sizes on disk, in memory, and over the network. Third, on the server side, each client `PUT` incurs two backends `puts`: one into `pending` and `good`. If keeping a single value for `good`, the second `put` into `good` (as is the case with normal, single-valued eventually consistent operation) is really a `get` then a possible `put`: the server first checks if a higher-timestamped value has been written and, if not, it overwrites the current value. The `put` into `pending` is simpler: if message delivery is exactly-once, then `puts` can be made unconditionally.

Optimizations. The use of `pending` and `required` is effectively to handle the above race condition, where some but not all replicas have realized that a write is pending stable. With this in mind, there are several optimizations we can perform:

If a replica serves a read from `pending`, it must mean one of two things: either the write is stable pending or the writing client for the write is requesting the data item (this second case does not occur in the below pseudocode since own-writes are served out of `write_buffer`). In the former case, the replica can safely install the write into `good`.

As we mention in Section 5.1.2, the size of `good` can be limited to a single item. Recall that any item can be served from `good` as long as its timestamp is greater than the required timestamp. Accordingly, if replicas store multiple writes in `good`, any time they can serve a write from `good`, the highest-timestamped write is always a valid response candidate (and, indeed, returning the highest-timestamped write provides the best visibility). Note, however, that serving a higher-than-requested timestamp from `pending` is dangerous: doing so might mean returning a write that is not pending stable and, via appropriate unfortunate partition behavior, force a client to stall while an available replica waits for the write to arrive in `pending`.

The below pseudocode implements a push-based acknowledgment scheme: replicas eagerly notify other replicas about `pending` writes. This need not be the case and, indeed, it is wasteful: if a write in `pending` has lower timestamp than the highest timestamped write in `good`, it can safely be discarded. Especially for data items with high temporal locality or data stores that are replicated over WAN, this `pending` invalidation may be a frequent event. Instead, replicas can periodically poll other replicas for acknowledgments (i.e., asking for either a match in `pending` or a higher timestamp in `good`); this reduces both network utilization and, in the case of durable `pending` and `good` sets, disk utilization.

There are ample opportunities for batching of pending and good acknowledgments. Batching is easier in a push-based acknowledgment scheme, as pull-based acknowledgments may require invalidation to actually reduce network transfers. Moreover, if a replica is responsible for several writes within a transaction, notify messages can be coalesced. We have also observed that batching multiple background operations together (particularly for anti-entropy) allows for more efficient compression—especially important in network-constrained environments.

Implementation note. Our implementation from Section 6 implements the below pseudocode, with batching between notify operations and a single-item good. We have not yet investigated the efficiency of a pull-based approach (mostly due to our choice of RPC library, which unfortunately does not currently handle multiple outstanding requests per socket). We have also not experimentally determined the visibility effect of the side-channel pending stable notifications as in the first optimization.

Algorithm HAT Read Atomic Isolation

Shared Data Types

`timestamp`: unique, per transaction identifier
`write` : [`key` : k, `value` : v, `timestamp` : ts, `set<key>` : $sibs$]

Server-side Data Structures and Methods

```
set<write>:pending
set<write>:good
map<timestamp, int>:acks
```

procedure PUT(`write`:w)
    `pending`.add(w)
    for key $k_t \in w.sibs$ **do**
        {all replicas for sib}.notify($w.ts$)
    asynchronously send w *to other replicas via anti-entropy*
    return

procedure NOTIFY(`timestamp`:ts)
    `acks`.get(ts).increment()
    if all `acks` received for all replicas for ts's $sibs$ **then**
        `good`.add(w)
        `pending`.remove(w)

procedure GET(`key`:k, `timestamp`:$ts_{required}$)
    if $ts_{required} = \bot$ **then**
        return $w \in$ `good` s.t. $w.key = key$ with highest timestamp
    else if $\exists\ w \in$ `good` s.t. $w.key = key, w.ts \geq ts_{required}$ **then**
        return w
    else
        return $w \in$ `pending` s.t. $w.key = key, w.ts = ts_{required}$

Client-side Data Structures and Methods

```
int:cur_txn
map<key, value>:write_buffer
map<key, timestamp>:required
```

procedure BEGIN_TRANSACTION
    `cur_txn` = *new txn ID* `// (e.g., clientID+logicalClock)`

procedure PUT(`key`:k, `value`:v)
    `write_buffer`.put(k, v)

procedure GET(`key`:k)
    if $k \in$ `write_buffer` **then**
        return `write_buffer`.get(k) `// if we want per-TxN RYW`
    w_ret = (available replica for k).get(k, `required`.get(k))
    for $(tx_{key} \in w_{ret}.sibs$ **do**
        if $w_{ret}.ts >$ `required`.$get(tx_{key})$ **then**
            `required`.put$(tx_{key}, w_{ret}.ts)$
    return $w\_ret.value$

procedure COMMIT
    for $(tx_{key}, v) \in$ `write_buffer` **do**
        r = (available replica for tx_{key})
        r.put($[tx_{key}, v,$ `cur_txn`, `write_buffer`.$keys()]$)
    CLEANUP()

procedure ABORT
    CLEANUP()

procedure CLEANUP
    `write_buffer`.clear()
    `required_map`.clear()