HAT, not CAP: Towards Highly Available Transactions

Peter Bailis[†], *Alan Fekete*[◇], *Ali Ghodsi*[†,‡], *Joseph M. Hellerstein*[†], *Ion Stoica*[†]
[†] *UC Berkeley* [◇] *University of Sydney* [‡] *KTH/Royal Institute of Technology*

Abstract

While the CAP Theorem is often interpreted to preclude the availability of transactions in a partition-prone environment, we show that highly available systems can provide useful transactional semantics, often matching those of today's ACID databases. We propose Highly Available Transactions (HATs) that are available in the presence of partitions. HATs support many desirable ACID guarantees for arbitrary transactional sequences of read and write operations and permit low-latency operation.

1 Introduction

To provide high availability and low latency, many recent distributed data stores eschew strongly consistent semantics [30, 32, 37, 40, 56, 58, 62]. Indeed, the CAP Theorem shows that it is impossible to simultaneously provide single-key linearizable data "consistency" [46] and available operation in the presence of arbitrary partitions between servers [42]. In the absence of partitions, an algorithm that is not highly available may also require costly round-trip messaging between servers [19]. Accordingly, CAP sends a clear message: strong semantics have a cost, and systems designers desiring "always on" operation and low latency must choose between one of many weak consistency models.

The CAP Theorem addresses a strict form of distributed consistency (linearizability) [42] that provides strong recency guarantees on data [46]. As such, CAP does not directly apply to database transactions, which allow users to combine sequences of separately submitted operations (possibly on different items) into groups with useful properties including atomicity, (application-level) consistency, isolation, and durability [44]. While these ACID transactions are grounded in decades of database tradition [27, 28, 43], many low-latency, highly available stores eschew transactional, multi-object semantics, often under the impression that CAP pertains to transactions, that it is impossible or unrealistic to provide highly available general-purpose transactional semantics [25, 30, 31, 39, 40, 56, 58]. Today's users must often settle for transactions with restricted data types or operations (e.g., read-only transactions, entity groups) [21, 24, 50, 51, 52] or instead face unavailability and high latency [29, 33, 34, 35, 47, 48, 50, 59, 61].

In this paper, we show that many transactional semantics from ACID databases can be provided with high availability in the presence of partitions. Serializability, the classic form of ACID isolation—which, unlike linearizability, does *not* provide recency guarantees—is not achievable with high availability [36]. However, most ACID and "NewSQL" databases provide weaker forms of isolation—usually by default, and often as the only options offered (§2). Databases have provided these weak guarantees for decades [43], suggesting that they are useful to application programmers, yet traditional *implementations* of these guarantees are not available: lock-based [43], mastered [27], and several other distributed transaction implementations [28] all suffer from unavailability. We show that this need not be the case: many of these transactional semantics can be made available in partition-prone environments (e.g., geo-replicated systems), and, as a consequence, achievable with low latency—two increasingly common requirements [32, 51, 52, 59, 64].

In this work, we motivate the further study of *Highly Available Transactions* (HATs): the class of transactional guarantees achievable with high availability and low latency in partition-prone environments (§3). HATs offer the opportunity to match the formal semantics provided by today's databases without compromising low latency and "always on" operation. For the many applications written for non-serializable but still transactional databases commonly used in practice, we believe HATs can provide a useful programming model without the cost of traditional implementations. In this paper, as a starting point, we specifically demonstrate that HAT systems can provide ANSI-compliant Read Committed and Repeatable Read isolation via simple, proof-of-concept implementation strategies such as variants of lightweight client-side caching and server-side concurrent write reconciliation. We also describe guarantees that are *not* achievable, such as providing recency bounds or enforcing arbitrary global correctness criteria (§4).

Although the traditional database literature does not typically consider requirements for high availability, we believe that there is a rich opportunity to do so. Instead of reinventing decades of work on useful transactional semantics, the systems community can adopt existing, accepted standards when designing emerging highly available and scalable database systems. Our goal here is not to provide high-performance algorithms, a design space exploration, or a full exploration of HAT semantics. Instead, we briefly demonstrate that several useful ACID properties *can* be implemented with high availability and advocate the adoption of practical and prevalent transactional semantics in future HAT systems.

2 CAP, ACID, and Availability

Highly available systems have attracted substantial attention in recent years. The introduction of the CAP Theorem informed systems developers of concrete trade-offs between semantic strengths and coordination cost [40]: a system without high availability risks the possibility of indefinitely stalling a client in the presence of network partitions, and, in their absence, may incur long round-trip times (RTTs) [19]. In contrast, a highly available system guarantees a response and low latency but faces associated semantic costs.[1] This balance is the subject of considerable ongoing research [33, 50, 51, 52, 59].

Unfortunately, the relationship between the CAP Theorem and ACID transactions is not always well understood. As formally proven [42], the CAP Theorem states that it is impossible to guarantee that every request to a correct server eventually receives a response—even in the presence of partitions— while ensuring linearizability [46] (informally providing the user with the illusion of a single, centralized replica for each key). While CAP is an important result with implications for many end-users, recency guarantees are orthogonal to most ACID guarantees. For example, serializability, the gold standard of ACID transactions, requires that the effect of executing a set of transactions corresponds to some serial ordering between them; this guarantees application-level integrity constraints will not be violated. However, by itself, this serializability guarantee does not say anything about real-time ordering. Indeed, several notable models couple serializability guarantees with recency guarantees like linearizability (yielding *Strong (or Strict) Serializability* [20]). Yet CAP does not address the ability to provide serializability or any weaker ACID variants.

This apparent gap raises the question: is it possible to achieve ACID semantics with high availability? In the presence of network partitions, a system can guarantee clients a response by simply aborting all transactions (i.e., causing them to fail); this is highly available but is not very useful [55]. However, aborts may be intrinsic to some transactions: for example, in a bank, a withdrawal transaction may need to abort if it would violate an integrity constraint requiring every bank account balance to remain positive. Other transactions may be caused externally—by the availability and configuration of the database servers. We refer to the formal as *internal aborts* and the latter as *external aborts*. We say that a transaction has *replica availability* if it can contact at least one replica for every item it attempts to access [42]. We say that a system provides *transactional availability* if every transaction either eventually commits or internally aborts. We say a system provides *high availability* if, given replica availability, it provides transactional availability.

Serializability and Unavailability The database community has studied distributed database design and operation for decades, and it is well known that serializability is at odds with high availability; that is, for arbitrary read-write transactions, it is impossible to provide high availability and guarantee equivalence with a serial execution [36]. Consider the following example, where $w_d(v)$ denotes a write of value v to data item d and $r_d(v)$ denotes a read of value v from data item d:

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

Suppose we have two servers, each acting as a replica for both x and y, and there is a network partition between the two servers. We can trivially achieve serializability by aborting both of these transactions. However, this lacks *liveness* and admits the possibility that the database will never make progress [55]. For transactional availability, T_1 and T_2 should eventually commit on their respective replicas. However, when they do, due to the partition, T_1 will read $a = \texttt{null}$ and T_2 will read $b = \texttt{null}$: there is no serial ordering of T_1 and T_2 under which these reads could have occurred, so serializability is violated.

Weak Isolation and the Status Quo While serializability is not achievable with high availability, databases frequently offer a wide spectrum of weaker guarantees [20, 43]. Even on a single-node database, the penalties associated with providing serializability can be severe, including decreased concurrency, reduced performance, and the possibility of deadlock. Accordingly, since the earliest database systems such as System R in 1976 [43], databases have provided a range of user-configurable "weak isolation" properties. These properties do not guarantee serializability but offer benefits such as increased concurrency and ease of implementation. The range of models has parallels to the many distributed consistency models found in the systems literature: weaker models provide performance benefits but are often more difficult to reason about.

Perhaps surprisingly, most databases today typically provide weak isolation instead of serializability. We surveyed the default and maximum isolation levels for many "ACID" and "NewSQL" databases according to vendor documentation (Table 1). Only three of 18 databases provided serializability by default, and eight—including database titans like Oracle 11g—did not provide serializability at all. In contrast, eight stores provided

[1] There is a range of intermediate failure models to consider—for example, systems such as Spanner [33] require that a majority of replicas be available. They correspondingly may provide strong semantic guarantees but requests that are routed to non-failing (minority groups of) servers are not guaranteed a response under network partitions. In the absence of network partitions, transactions can incur latency from round-trips, which may be particularly expensive over WANs. We focus on the requirement of high availability, wherein every non-failing server guarantees a response, whether partitioned or not [42].

Database	Default	Maximum
Actian Ingres 10.0/10S [1]	S	S
Aerospike [2]	RC	RC
Akiban Persistit [3]	SI	SI
Clustrix CLX 4100 [4]	RR	RR
Greenplum 4.1 [8]	RC	S
IBM DB2 10 for z/OS [5]	CS	S
IBM Informix 11.50 [9]	Depends	S
MySQL 5.6 [12]	RR	S
MemSQL 1b [10]	RC	RC
MS SQL Server 2012 [11]	RC	S
NuoDB [13]	CR	CR
Oracle 11g [14]	RC	SI
Oracle Berkeley DB [7]	S	S
Oracle Berkeley DB JE [6]	RR	S
Postgres 9.2.2 [15]	RC	S
SAP HANA [16]	RC	SI
ScaleDB 1.02 [17]	RC	RC
VoltDB [18]	S	S
RC: read committed, RR: repeatable read, SI: snapshot isolation, S: serializability, CS: cursor stability, CR: consistent read		

Table 1: Default and maximum isolation levels for ACID and NewSQL databases as of January 2013.

Read Committed by default, while three "NewSQL" data stores *only* offered Read Committed isolation.

In our investigation, we found that many databases claiming strong guarantees often offered weaker semantics. One store with an effective maximum of Read Committed isolation claimed to provide "strong consistency (ACID)" [2], while another claiming "100% ACID" and "fully support[ed] ACID transactions" uses consistent read isolation [13]. Moreover, snapshot isolation is often labeled as "serializability" [14]. We have accompanied our bibliographic references with additional detail, but it is clear that these "ACID" guarantees rarely meet serializability's goal of automatically protecting data integrity as set out by the database literature. This is especially surprising given that these databases' "stronger" semantics are often thought to substantially differentiate them from their "NoSQL" peers [30, 56, 58].

These results—and several discussions with database developers and architects—indicate that weak isolation models are viable alternatives for many applications. There are applications that either work correctly with these models or else work well enough to accept the resulting anomalies in exchange for their performance benefits [45]. A key challenge is that, while the literature provides reasonable taxonomy of the models, it considers them in either a single-node context [43] or abstractly [20, 26]—it is unclear *which* models are achievable with high availability and which are not. Indeed, most weak isolation levels today are implemented in an unavailable manner.

3 Highly Available Transactions

The large number and prevalence of "weak ACID" guarantees suggests that, although we cannot provide serializability with high availability, providing weaker guarantees still provides users with a useful programming interface. In this section, we show that two major models: Read Committed and ANSI SQL Repeatable Read are achievable in a highly available environment. This paves the way for broader theoretical and design studies of *Highly Available Transactions*: multi-operation, multi-object guarantees achievable with high availability. We will sketch algorithms solely as a proof-of-concept for high availability; further engineering is required to improve and evaluate their performance.

Read Committed We first consider Read Committed isolation—a particularly widely used isolation model in our survey. Read Committed is often the lowest level of isolation provided in a database beyond "No Isolation." It requires that transactions do not read uncommitted data items, which would result in "Dirty Reads phenomena (i.e., ANSI $P1$ [22] and Adya $G1\{a,b,c\}$ [20]). In the example below, T_3 should never see $a = 1$, and, if T_2 aborts, T_3 will never see $a = 3$:

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

Read Committed is a useful property because it ensures that transactions will not read intermediate versions of a given data item or read data from transactions that will eventually be rolled back (and thus will never have "existed" in the database).

Read Committed also disallows "Dirty Write" phenomena (Adya's $G0$ [20]), so the database will "consistently" order writes from concurrent transactions . Effectively, the database induces a total order on transactions, and the replicas of the database should apply writes in this order. For example, if T_1, T_2 commit, T_3 can *eventually* only read $a = b = 1$ or $a = b = 2$:

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

This is useful because it effectively guarantees cross-item convergence, or eventual consistency. "Dirty Write" occurs when a database chooses different "winning" transactions across simultaneously written keys.

We can implement Read Committed isolation with high availability. If servers never reveal dirty data to clients, then clients will never experience "Dirty Read" phenomena. To ensure this, servers should only serve data that they are sure has been committed. Servers can explicitly buffer incoming writes until they receive a commit message from clients. Alternatively, clients can

buffer their writes until they decide to commit and then send the servers the final value for every data item that they write. To prevent "Dirty Write" phenomena, servers need to pick which version of each item to store in a way that is consistent with some transaction ordering. To provide an ordering, clients can choose a unique integer ID (e.g., client ID and timestamp) at the start of each transaction and attach it to all writes that they send to servers. In turn, servers can ensure that they only store values for each data item with increasing IDs (Thomas Write Rule [28]). Transaction ID generation is coordination-free and, if a transaction can access a server, the server can safely apply its updates and provide a response.

ANSI Repeatable Read ANSI Repeatable Read captures the spirit that each transaction's view of the database should not change as it executes. Under ANSI Repeatable Read, each transaction can only read one version of each data item that the transaction did not itself produce. In the example below, T_3 must read $a = 1$:

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

ANSI Repeatable Read also subsumes Read Committed guarantees (preventing "Dirty Read" and "Dirty Write"). It is a useful property because it captures the essence of database "isolation." It does not restrict changes to data items while a transaction is running but only guarantees that each transaction's view of the database will be "frozen" along an arbitrary cut of data items.

ANSI Repeatable Read is achievable by a HAT system. To ensure that transactions read from a stable cut across items, clients can cache versions of items that they read from servers. For each read, the client checks its cache and, if there is a cache miss, it fetches the latest value from a suitable server. For each write, the client updates its cache. Upon commit or abort, the cache is cleared. This does not require any coordination among clients or servers and is highly available. Moreover, the earlier algorithms can simultaneously provide Read Committed guarantees.

Discussion We have shown that two frequently used ACID isolation guarantees are achievable with high availability (via near-trivial implementations). This is perhaps surprising because traditional implementations of these guarantees are not highly available; for example, if we were to use locking to ensure that we continued to read the same data item (in ANSI Repeatable Read), then, in the presence of partitions, mutual exclusion might cause operations to stall indefinitely. Key to our HAT implementations is a focus on high availability as a first class concern, leveraging multi-versioning and, where applicable, caching. However, while these results are encouraging, we must be careful of several subtleties in existing implementations and additional guarantees.

Varying guarantees. While we have stayed faithful to implementation-agnostic interpretations in the literature, actual implementations of these guarantees can provide varying additional properties. For example, lock-based implementations typically provide recency guarantees: if a client holds a read lock on a data item, it can access the last written version. Similarly, while ANSI Repeatable Read describes the spirit of "isolation," Gray's traditional lock-based implementation provides serializability except for predicate-based reads [43, 26]. This problem is further complicated by real-world database implementations. For example, Oracle 11g will attempt to read the latest version of each data item as of the wall-clock transaction start time. On a single-site system, this property holds. However, in a distributed setting, Oracle does not properly synchronize the timestamps across replicas and accordingly may serve stale reads [54].

Usability. Database idiosyncrasies aside, one might wonder whether these isolation models are actually meaningful to program against. On the one hand, the prevalence of their use suggests that programmers have managed to code around the possible anomalies they permit. On the other hand, they both allow particularly odd behavior such as reading all `null` values. We can generously say that the properties discussed here are underspecified. Yet, when combined with existing properties (often from the distributed systems literature), they yield useful transactional models. For example, coupling ANSI Repeatable Read with causal consistency yields a model in which causality between transactions is respected and isolation properties are guaranteed (called *PL-2L* in the database literature [20]). Coupled with intelligent conflict resolution [60], this yields a semantics that is substantially stronger than today's production highly available systems and "NoSQL" stores.

Implications. Collectively, these issues indicate that there is need for future exploration of the space of highly available models. We have characterized two, showing that real-world ACID is not synonymous with unavailability. However, perhaps unfortunately, ACID is not easily captured by a single model (or even three)—rather, just like the plethora distributed consistency models, there is a range of possible semantics, each varying from store to store. While it is tempting to jump straight to the top of the hierarchy and adopt strong serializability [33], doing so will sacrifice many of the benefits of high availability for general-purpose transactions. We omit a full discussion, but we believe that the many other ACID properties available to HAT systems will improve programmability (§6).

4 Unsupported Semantics

Highly Available Transactions cannot "beat CAP." The requirement for high availability prohibits a HAT system from providing several useful guarantees.

Recency As a corollary to the CAP Theorem, given the possibility of indefinitely long partitions, HAT systems cannot guarantee any finite bounds on recency; they are easily violated by a partition of length greater than the stated bound. However, HAT systems *can* limit staleness of reads to the length of partitions, and, when the system is not partitioned, it is possible to execute more coordination-intensive protocols than we have discussed here. Additionally, it is possible to provide a notification of "potential data staleness" via protocols often requiring heartbeats and negative acknowledgments [57, 63].

Global Integrity Constraints As a corollary to the unavailability of serializability, a highly available system cannot enforce arbitrary global integrity constraints (i.e., single-copy semantics) over data [36]. As an example, general-purpose uniqueness constraints on a set of values are not achievable: two clients might try to write the same value on different sides of a partition. This is a limitation of data stores operating at any consistency level weaker than serializability, whether Repeatable Read or even *PL-2L*. We can, however, enforce some local constraints, such as checking for a given value like `null` and use well known database techniques such as Sagas [41], compensating actions [45], and limited forms of escrow [53] to provide consistency despite partitions. Additionally, clients can perform constraint checking on possibly out-of-date values (e.g., conditional modifications, which are often supported in weakly consistent stores like Riak)—useful for monotonic logic [21].

5 Related Work

We note a recent resurgence of interest in distributed multi-object semantics, both in academia [34, 35, 47, 48, 50, 51, 52, 59, 61, 64] and industry [24, 29, 33]. This is a strong indication that transactional semantics are useful to end-users and that, even in a distributed environment, their additional design and algorithmic complexity is worthwhile. However, few—if any—highly available systems have seriously pursued database ACID semantics. Rather, today's users must often settle for operations over restricted data types (e.g., commutative and monotonic operations) [21, 50] or with restricted semantics (e.g., read-only operations, entity groups) [24, 51, 52] or instead face unavailability and high latency [29, 33, 34, 35, 47, 48, 50, 59, 61] due to dependencies on unavailable protocols like Paxos or 2PC or weakened assumptions about failure.

There are several systems that provide hints at what is possible with HAT semantics. Bayou has support for performing atomic multi-read and multi-write transactions as well as conditional multi-write operation. Thus, Bayou is an example of an early system with highly available transactional support [60]. Several recent stores provide *PL-2L* variants, including Eiger [52] for multi-read and multi-write transactions, and Swift [64] and Bolt-on Causal Consistency [23] for highly available read/write transactions. Moreover, Kraska [47] has proposed an architecture to prevent "Dirty Read" by placing highly available queues between clients and data storage. These systems all provide useful reference points for what can be achieved in HAT systems. Our current focus is on achieving well-established transactional isolation levels for read-write transactions: here, Read Committed and ANSI Repeatable Read guarantees.

6 Towards HATs for All

In this paper, we have investigated the relationship between the CAP Theorem and ACID transactions—whether ACID transactions can be made highly available in the presence of network partitions. We demonstrated that transactions do not preclude high availability, but unavailable designs do. Accordingly, we have proposed Highly Available Transactions (HATs), which always provide high availability and typically provide low latency. We have shown how several standardized weak isolation properties from widely deployed ACID databases can be achieved in HAT systems.

According to implementation-independent definitions [20, 26], we believe that HATs can match the default (and sometimes strongest) semantics of several existing "ACID" and "NewSQL" stores without compromising always on operation. Our preliminary results (omitted for brevity) indicate that a combination of transactional atomicity (even under partial replication), causal consistency, Phantom prevention, ANSI Repeatable Read, and replica convergence are achievable with HATs but preventing Lost Update (as in Snapshot Isolation) and Write Skew are not. While the sample algorithms here serve as feasibility proofs, there is substantial work to be done improving them and in further mapping the HAT design space.

Acknowledgments The authors would like to thank Peter Alvaro, Neil Conway, Evan Jones, and the HotOS reviewers for insightful feedback on earlier revisions of this paper.

This research is supported in part by National Science Foundation grants CCF-1139158, CNS-0722077, IIS-0713661, IIS-0803690, and IIS-0917349, DARPA awards FA8650-11-C-7136 and FA8750-12-2-0331, Air Force Office of Scientific Research Grant FA9550-08-1-0352, the National Science Foundation Graduate Research Fellowship under Grant DGE-1106400. and gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Clearstory Data, Cloudera, EMC, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, NTT Multimedia Communications Laboratories, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!.

References

[1] Actian documentation: Isolation levels. http://docs.actian.com/ingres/10s/database-administrator-guide/2349-isolation-levels, January 2013.

[2] Aerospike: Home > Performance > ACID. http://www.aerospike.com/performance/acid-compliance/, January 2013. Note: "multi-key operations may not be serialized with each other"; single-key immediate consistency roughly translates to Read Committed isolation.

[3] Akiban Documentation: Transactions. http://www.akiban.com/ak-docs/admin/persistit/Transactions.html, January 2013.

[4] Clustrix System Administrator's Guide CLX 4100 Series, Version 4.1. http://www.clustrix.com/Portals/146389/docs/Clustrix_System_Administrators_Guide_v4.1.pdf, January 2013.

[5] DB2 10 for z/OS: Choosing an ISOLATION option. http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=%2Fcom.ibm.db2z10.doc.perf%2Fsrc%2Ftpc%2Fdb2z_chooseisolationoption.htm, January 2013. Note: DB2's "Repeatable Read" isolation matches serializability—the only instance in which we have found a stronger level than named.

[6] Getting Started with Berkeley DB, Java Edition Transaction Processing: Isolation. http://docs.oracle.com/cd/E17277_02/html/TransactionGettingStarted/isolation.html, January 2013.

[7] Getting Started with Berkeley DB, Transaction Processing: Isolation. http://docs.oracle.com/cd/E17076_02/html/gsg_txn/JAVA/isolation.html, January 2013.

[8] Greenplum database 4.2 database administrator guide rev: A01. http://media.gpadmin.me/wp-content/uploads/2012/11/GPDBAGuide.pdf, January 2013.

[9] IBM Informix 11.50: SET ISOLATION statement. http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp?topic=%2Fcom.ibm.sqls.doc%2Fids_sqs_1161.htm, January 2013.

[10] MemSQL 1b documentation: Transactions and isolation levels. http://developers.memsql.com/docs/1b/isolationlevel.html, January 2013.

[11] Microsoft SQL Server 2012: SET TRANSACTION ISOLATION LEVEL (Transact-SQL). http://msdn.microsoft.com/en-us/library/ms173763.aspx, January 2013.

[12] MySQL 5.6 Reference Manual: 13.3.6 SET TRANSACTION Syntax. http://dev.mysql.com/doc/refman/5.0/en/set-transaction.html, January 2013.

[13] NuoDB: Transactions and Isolation Levels. http://www.nuodb.com/nuodb-online-documentation/references/r_Lang/r_Transactions.html, January 2013.

[14] Oracle Database Concepts 11g Release 1 (11.1): 13 Data Concurrency and Consistency. http://docs.oracle.com/cd/B28359_01/server.111/b28318/consist.htm#autoId8, January 2013. Note: several sources note confirm that Oracle's "serializable isolation" is actually Snapshot Isolation [38, 49].

[15] PostgreSQL 9.2.2 Documentation: 13.2. Transaction Isolation. http://www.postgresql.org/docs/9.2/static/transaction-iso.html, January 2013.

[16] SAP HANA Reference: SET TRANSACTION. http://help.sap.com/hana/html/sql_set_transaction.html, January 2013. Note: the described "'SERIALIZABLE" isolation level is actually a description of Snapshot Implementation, like [14].

[17] ScaleDB Cluster Manual: For versions 1,02 and higher. http://www.scaledb.com/pdfs/ScaleDB_Cluster_Manual.pdf, 23 December 2012.

[18] The VoltDB FAQ. http://voltdb.com/dig-deeper/faq.php, also verified with VoltDB stakeholders, January 2013.

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

[20] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

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

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

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

[24] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR 2011*, pages 223–234.

[25] barrkel. Hacker News comment thread. https://news.ycombinator.com/item?id=1163140, 2009. (Anecdotal but unequivocal.).

[26] 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*.

[27] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

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

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

[30] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[31] G. Clarke. The Register: NoSQL's CAP theorem busters: We don't drop ACID. `http://www.theregister.co.uk/2012/11/22/foundationdb_fear_of_cap_theorem/`, November 2012.

[32] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. In *VLDB 2008*.

[33] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI 2012*.

[34] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *USENIX ATC 2012*.

[35] 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.

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

[37] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP 2007*.

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

[39] D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *ACM SIGMOD Record*, 38(1):43–48, 2009.

[40] A. Fox and E. A. Brewer. Harvest, yield, and scalable tolerant systems. In *HotOS 1999*.

[41] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD 1987*.

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

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

[44] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

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

[46] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[47] T. Kraska. *Building Database Applications in the Cloud*. PhD thesis, 2010.

[48] T. Kraska, G. Pang, M. Franklin, and S. Madden. Mdcc: Multi-data center consistency. In *Eurosys 2013*.

[49] T. Kyte. *Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions*. Apress, 2 edition, 2010. p. 253.

[50] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI 2012*.

[51] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP 2011*.

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

[53] P. E. O'Neil. The escrow transactional method. *ACM TODS*, 11(4):405–430.

[54] Oracle Database Administrator's Guide 11g Release 2 (11.2): Managing Read Consistency. `http://docs.oracle.com/cd/E11882_01/server.112/e25789/consist.htm`.

[55] F. Pedone and R. Guerraoui. On transaction liveness in replicated databases. In *Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 104–109. IEEE, 1997.

[56] D. Pritchett. BASE: An Acid Alternative. *Queue*, 6(3):48–55, 2008.

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

[58] S. Sakr, A. Liu, D. M. Batista, and M. Alomari. A survey of large scale data management approaches in cloud environments. *IEEE Communications Surveys & Tutorials*, 13(3):311–336.

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

[60] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP 1995*.

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

[62] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.

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

[64] 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://asc.di.fct.unl.pt/~nmp/swiftcomp/swiftcloud.html`).