

# The many faces of consistency

Marcos K. Aguilera                      Douglas B. Terry  
VMware Research Group      Samsung Research America

## Abstract

*The notion of consistency is used across different computer science disciplines from distributed systems to database systems to computer architecture. It turns out that consistency can mean quite different things across these disciplines, depending on who uses it and in what context it appears. We identify two broad types of consistency, state consistency and operation consistency, which differ fundamentally in meaning and scope. We explain how these types map to the many examples of consistency in each discipline.*

## 1 Introduction

Consistency is an important consideration in computer systems that *share* and *replicate* data. Whereas early computing systems had private data exclusively, shared data has become increasingly common as computers have evolved from calculating machines to tools of information exchange. Shared data occurs in many types of systems, from distributed systems to database systems to multiprocessor systems. For example, in distributed systems, users across the network share files (e.g., source code), network names (e.g., DNS entries), data blobs (e.g., images in a key-value store), or system metadata (e.g., configuration information). In database systems, users share tables containing account information, product descriptions, flight bookings, and seat assignments. Within a computer, processor cores share cache lines and physical memory.

In addition to sharing, computer systems increasingly replicate data within and across components. In distributed systems, each site may hold a local replica of files, network names, blobs, or system metadata—these replicas, called caches, increase performance of the system. Database systems also replicate rows or tables for speed or to tolerate disasters. Within a computer, parts of memory are replicated at various points in the cache hierarchy (L1, L2, L3 caches), again for speed. We use the term replica broadly to mean any copies of the data maintained by the system.

In all these systems, data sharing and replication raise a fundamental question: what should happen if a client modifies some data items and simultaneously, or within a short time, another client reads or modifies the same items, possibly at a different replica?

This question does not have a single answer that is right in every context. A consistency property governs the possible outcomes by limiting how data can change or what clients can observe in each case. For example, with DNS, a change to a domain may not be visible for hours; the only guarantee is that updates will be seen eventually—an example of a property called eventual consistency [23]. But with flight seat assignments, updates must be immediate and mutually exclusive, to ensure that no two passengers receive the same seat—an example

---

*Copyright 2016 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.*

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

---

of a strong type of consistency provided by serializability [5]. Other consistency properties include causal consistency [13], read-my-writes [21], bounded staleness [1], continuous consistency [1, 25], release consistency [10], fork consistency [16], epsilon serializability [18], and more.

Consistency is important because developers must understand the answer to the above fundamental question. This is especially true when the clients interacting with the system are not humans but other computer programs that must be coded to deal with all possible outcomes.

In this article, we examine many examples of how consistency is used across three computer science disciplines: distributed systems, database systems, and computer architecture. We find that the use of consistency varies significantly across these disciplines. To bring some clarity, we identify two fundamentally different types of consistency: state consistency and operation consistency. State consistency concerns the state of the system and establishes constraints on the allowable relationships between different data items or different replicas of the same items. For instance, state consistency might require that two replicas store the same value when updates are not outstanding. Operation consistency concerns operations on the system and establishes constraints on what results they may return. For instance, operation consistency might require that a read of a file reflects the contents of the most recent write on that file. State consistency tends to be simpler and application dependent, while operation consistency tends to be more complex and application agnostic. Both types of consistency are important and, in our opinion, our communities should more clearly disentangle them.

While this article discusses different forms of consistency, it focuses on the *semantics* of consistency rather than the *mechanisms* of consistency. Semantics refer to what consistency properties the system provides, while mechanisms refer to how the system enforces those properties. Semantics and mechanisms are closely related but it is important to understand the former without needing to understand the latter.

The rest of this article is organized as follows. We first explain the abstract system model and terminology used throughout the article in Section 2. We present the two types of consistency and their various embodiments in Section 3. We indicate how these consistency types occur across different disciplines in Section 4.

## 2 Abstract model

We consider a setting with multiple *clients* that submit *operations* to be executed by the system. Clients could be human users, computer programs, or other systems that do not concern us. Operations might include simple read and write, read-modify-write, start and commit a transaction, and range queries. Operations typically act on data items, which could be blocks, files, key-value pairs, DNS entries, rows of tables, memory locations, and so on.

The system has a *state*, which includes the current values of the data items. In some cases, we are interested in the consistency of client caches and other replicas. In these cases, the caches and other replicas are considered to be part of the system and the system state includes their contents.

An operation execution is not instantaneous; rather, it *starts* when a client submits the operation, and it *finishes* when the client obtains its response from the system. If the operation execution returns no response, then it finishes when the system is no longer actively processing it.

Operations are distinct from operation executions. Operations are static and a system has relatively few of them, such as read and write. Operation executions, on the other hand, are dynamic and numerous. A client can execute the same operation many times, but each operation execution is unique. While technically we should separate operations from operation executions, we often blur the distinction when it is clear from the context (e.g., we might say that the read operation finishes, rather than the execution of the read operation finishes).

## 3 Two types of consistency

We are interested in what happens when shared and replicated data is accessed concurrently or nearly concurrently by many clients. Generally speaking, consistency places constraints on the allowable outcomes of operations, according to the needs of the application. We now define two broad types of consistency. One places constraints on the state, the other on the results of operations.

### 3.1 State consistency

State consistency pertains to the state of the system; it consists of properties that users expect the state to satisfy despite concurrent access and the existence of multiple replicas. State consistency is also applicable when data can be corrupted by errors (crashes, bit flips, bugs, etc), but this is not the focus of this article.

State consistency can be of many subcategories, based on how the properties of state are expressed. We explain these subcategories next.

#### 3.1.1 Invariants

The simplest subcategory of state consistency is one defined by an invariant—a predicate on the state that must evaluate to true. For instance, in a concurrent program, a singly linked list must not contain cycles. In a multiprocessor system, if the local caches of two processors keep a value for some address, it must be the same value. In a social network, if user  $x$  is a friend of user  $y$  then  $y$  is a friend of  $x$ . In a photo sharing application, if a photo album includes an image then the image’s owner is the album.

In database systems, two important examples are uniqueness constraints and referential integrity. A *uniqueness constraint* on a column of a table requires that each value appearing in that column must occur in at most one row. This property is crucial for the primary keys of tables.

*Referential integrity* concerns a table that refers to keys of another table. Databases may store relations between tables by including keys of a table within columns in another table. Referential integrity requires that the included keys are indeed keys in the first table. For instance, in a bank database, suppose that an accounts table includes a column for the account owner, which is a user id; meanwhile, the user id is the primary key in a users table, which has detailed information for each user. A referential integrity constraint requires that user ids in the accounts table must indeed exist in the users table.

Another example of state consistency based on invariants is *mutual consistency*, used in distributed systems that are replicated using techniques such as primary-backup [2]. Mutual consistency requires that replicas have the same state when there are no outstanding updates. During updates, replicas may diverge temporarily since the updates are not applied simultaneously at all replicas.

#### 3.1.2 Error bounds

If the state contains numerical data, the consistency property could indicate a maximum deviation or error from the expected. For instance, the values at two replicas may diverge by at most  $\epsilon$ . In an internet-of-things system, the reported value of a sensor, such as a thermometer, must be within  $\epsilon$  from the actual value being measured. This example relates the state of the system to the state of the world. Error bounds were first proposed within the database community [1] and the basic idea was later revived in the distributed systems community [25].

#### 3.1.3 Limits on proportion of violations

If there are many properties or invariants, it may be unrealistic to expect all of them to hold, but rather just a high percentage. For instance, the system may require that at most one user’s invariants are violated in a pool of

a million users; this could make sense if the system can compensate a small fraction of users for inconsistencies in their data.

### 3.1.4 Importance

Properties or invariants may be critical, important, advisable, desirable, or optional, where users expect only the critical properties to hold at all times. Developers can use more expensive and effective mechanisms for the more important invariants. For instance, when a user changes her password at a web site, the system might require all replicas of the user account to have the same password before acknowledging the change to the user. This property is implemented by contacting all replicas and waiting for replies, which can be an overly expensive mechanism for less important properties.

### 3.1.5 Eventual invariants

An invariant may need to hold only after some time has passed. For example, under eventual consistency, replicas need not be the same at all times, as long as they *eventually* become the same when updates stop occurring. This eventual property is appropriate because replicas may be updated in the background or using some anti-entropy mechanism, where it takes an indeterminate amount of time for a replica to receive and process an update. Eventual consistency was coined by the distributed systems community [23], though the database community previously proposed the idea of reconciling replicas that diverge during partitions [9].

State consistency is limited to properties on state, but in many cases clients care less about the state and more about the results that they obtain from the system. In other words, what matters is the behavior that clients observe from interacting with the system. These cases call for a different form of consistency, which we discuss next.

## 3.2 Operation consistency

Operation consistency pertains to the operation executions by clients; it consists of properties that indicate whether operations return acceptable results. These properties can tie together many operation executions, as shown in the examples below.

Operation consistency has subcategories, with different ways to define the consistency property. We explain these subcategories next.

### 3.2.1 Sequential equivalence

This subcategory defines the permitted operation results of a concurrent execution in terms of the permitted operation results in a sequential execution—one in which operations are executed one at a time, without concurrency. More specifically, there must be a way to take the execution of all operations submitted by any subset of clients, and then *reduce* them to a sequential execution that is correct. The exact nature of the reduction depends on the specific consistency property. Technically, the notion of a correct sequential execution is system dependent, so it needs to be specified as well, but it is often obvious and therefore omitted.

We now give some examples of sequential equivalence.

*Linearizability* [12] is a strong form of consistency. Intuitively, the constraint is that each operation must appear to occur at an instantaneous point between its start and finish times, where execution at these instantaneous points form a valid sequential execution. More precisely, we define a partial order  $<$  from the concurrent execution, as follows:  $op_1 < op_2$  iff  $op_1$  finishes before  $op_2$  starts. There must exist a legal total order  $T$  of all operations with their results, such that (1)  $T$  is consistent with  $<$ , meaning that if  $op_1 < op_2$  then  $op_1$  appears before  $op_2$  in  $T$ , and (2)  $T$  defines a correct sequential execution. Linearizability has been traditionally used

to define the correct behavior of concurrent data structures; more recently, it has also been used in distributed systems.

*Sequential consistency* [14] is also a strong form of consistency, albeit weaker than linearizability. Intuitively, it requires that operations execute as if they were totally ordered in a way that respects the order in which each client issues operations. More precisely, we define a partial order  $<$  as follows:  $op_1 < op_2$  iff both operations are executed by the same client and  $op_1$  finishes before  $op_2$  starts. There must exist a total order  $T$  such that (1)  $T$  is consistent with  $<$ , and (2)  $T$  defines a correct sequential execution. These conditions are similar to linearizability, except that  $<$  reflects just the local order of operations at each client. Sequential consistency is used to define a strongly consistent memory model of a computer, but it could also be used in the context of concurrent data structures.

The next examples pertain to systems that support *transactions*. Intuitively, a transaction is a bundle of one or more operations that must be executed as a whole. More precisely, there are special operations to start, commit, and abort transactions; and operations on data items are associated with a transaction. The system provides an isolation property, which ensures that transactions do not significantly interfere with one another. There are many isolation properties: serializability, strong session serializability, order-preserving serializability, snapshot isolation, read committed, repeatable reads, etc. All of these are forms of operation consistency, and several of them are of the sequential equivalence subcategory. Here are some examples, all of which are used in the context of database systems.

*Serializability* [5] intuitively guarantees that each transaction appears to execute in series. More precisely, serializability imposes a constraint on the operations in a system: the schedule corresponding to those operations must be equivalent to a serial schedule of transactions. The serial schedule is called a serialization of the schedule.

*Strong session serializability* [8] addresses an issue with serializability. Serializability allows transactions of the same client to be reordered, which can be undesirable at times. Strong session serializability imposes additional constraints on top of serializability. More precisely, each transaction is associated with a session, and the constraint is that serializability must hold (as defined above) and the serialization must respect the order of transactions within every session: if transaction  $T_1$  occurs before  $T_2$  in the same session, then  $T_2$  is not serialized before  $T_1$ .

*Order-preserving serializability* [24], also called strict serializability [6, 17] or strong serializability [7], requires that the serialization order respect the real-time ordering of transactions. More precisely, the constraint is that serializability must hold and the serialization must satisfy the requirement that, if transaction  $T_1$  commits before  $T_2$  starts, then  $T_2$  is not serialized before  $T_1$ .

### 3.2.2 Reference equivalence

Reference equivalence is a generalization of sequential equivalence. It defines the permitted operation results by requiring the concurrent execution to be equivalent to a given reference, where the notion of equivalence and the reference depend on the consistency property. We now give some examples for systems with transactions. These examples occur often in the context of database systems.

*Snapshot isolation* [4] requires that transactions behave identically to a certain reference implementation, that is, transactions must have the same outcome as in the reference implementation, and operations must return the same results. The reference implementation is as follows. When a transaction starts, it gets assigned a monotonic start timestamp. When the transaction reads data, it reads from a snapshot of the system as of the start timestamp. When a transaction  $T_1$  wishes to commit, the system obtains a monotonic commit timestamp and verifies whether there is some other transaction  $T_2$  such that (1)  $T_2$  updates some item that  $T_1$  also updates, and (2)  $T_2$  has committed with a commit timestamp between  $T_1$ 's start and commit timestamp. If so, then  $T_1$  is aborted; otherwise,  $T_1$  is committed and all its updates are applied instantaneously as of the time of  $T_1$ 's commit timestamp.

Interestingly, the next two properties are examples of reference equivalence where the reference is itself defined by another consistency property. This other property is in the serial equivalence subcategory in the first example, and it is in the reference equivalence subcategory in the second example.

*One-copy serializability* [5] pertains to a replicated database system. The replicated system must behave like a reference system, which is a system that is not replicated and provides serializability.

*One-copy snapshot isolation* [15] also pertains to a replicated system. The requirement is that it must behave like a system that is not replicated and that provides snapshot isolation.

### 3.2.3 Read-write centric

The above subcategories of operation consistency apply to systems with arbitrary operations. The read-write centric subcategory applies to systems with two very specific operations: read and write. These systems are important because they include many types of storage systems, such as block storage systems, key value storage systems, and processors accessing memory. By focusing on the two operations, this subcategory permits properties that directly evoke the semantics of the operations. In particular, a write operation returns no information other than an acknowledgment or error status, which has no consistency implications. Thus, the consistency properties focus on the results of reads. Common to these properties is the notion of a read *seeing* the values of a set of writes, as we now explain. Each read is affected by some writes in the system; if every write covers the entire data item, then writes overwrite each other and the read returns the value written by one of them. But if the writes update just part of a data item, the read returns a combination of the written values in some appropriate order. In either case, the crucial consideration is the set of writes that *could* have potentially affected the read, irrespective of whether the writes are partial or not; we say that the read *sees* those writes. This notion is used to define several known consistency properties, as we now exemplify.

*Read-my-writes* [21] requires that a read by a client sees at least all writes previously executed by the same client, in the order in which they were executed. This property is relevant when clients expect to observe their own writes, but can tolerate delays before observing the writes of others. Typically, read-my-writes is combined with another read-write consistency property, such as bounded staleness or operational eventual consistency, defined below. By combined we mean that the system must provide both read-my-writes and the other property. Read-my-writes was originally defined in the context of distributed systems [21], then used in computer architecture to define memory models [19].

*Bounded staleness* [1], intuitively, bounds the time it takes for writes to be seen by reads. More precisely, the property has a parameter  $\delta$ , such that a read must see at least all writes that complete  $\delta$  time before the read started. This property is relevant when inconsistencies are tolerable in the short term as defined by  $\delta$ , or when time intervals smaller than  $\delta$  are imperceptible by clients (e.g.,  $\delta$  is in the tens of milliseconds and clients are humans). Bounded staleness was originally defined in the context of database systems [1] and has been used more recently in the context of cloud distributed systems [20].

*Operational eventual consistency* is a variant of eventual consistency (a form of state consistency) defined using operation consistency. The requirement is that each write be eventually seen by all reads, and if clients stop executing writes then eventually every read returns the same latest value [22].

*Cache coherence* originates from computer architecture to define the correct behavior of a memory cache. Intuitively, cache coherence requires that reads and writes to an individual data item (a memory location) satisfy some properties. The properties vary across the literature. One possibility [11] is to require that, for each data item: (1) a read by some client returns the value of the previous write by that client, unless another client has written in between, (2) a read returns the value of a write by another client if the write and read are sufficiently separated in time and if no other write occurred in between, and (3) writes are serialized.

### 3.3 Discussion

We now compare state consistency and operation consistency in terms of their level of abstraction, complexity, power, and application dependence.

#### 3.3.1 Level of abstraction

Operation consistency is an end-to-end property, because it deals with results that clients can observe directly. This is in contrast to state consistency, which deals with system state that clients observe indirectly by executing operations. In other words, operation consistency is at a higher level of abstraction than state consistency. As a result, a system might have significant state inconsistencies, but hide these inconsistencies externally to provide a strong form of operation consistency.

An interesting example is a storage system with three servers replicated using majority quorums [3], where (1) to write data, the system attaches a monotonic timestamp and stores the data at two (a majority of) servers, and (2) to read, the system fetches the data from two servers; if the servers return the same data, the system returns the data to the client; otherwise, the system picks the data with the highest timestamp, stores that data and its timestamp in another server (to ensure that two servers have the data), and returns the data to the client. This system violates mutual consistency, because when there are no outstanding operations, one of the servers deviates from the other two. However, this inconsistency is not observable in the results returned by reads, since a read filters out the inconsistent server by querying a majority. In fact, this storage system satisfies linearizability, one of the strongest forms of operation consistency.

#### 3.3.2 Complexity

Operation consistency is more complex than state consistency. With state consistency, developers gain a direct understanding of what states they can expect from the system. Each property concerns specific data items that do not depend on the execution. As a result, state consistency is intuitive and simple to express and understand. Moreover, state consistency can be checked by analyzing a snapshot of the system state, which facilitates debugging.

By contrast, operation consistency properties establish relations between operations that are spread over time and possibly over many clients, which creates complexity. This complexity makes operation consistency less intuitive and harder to understand, as can be observed from the examples in Section 3.2. Moreover, checking operation consistency requires analyzing an execution log, which complicates debugging.

#### 3.3.3 Power

Operation consistency and state consistency have different powers. Operation consistency can see all operations in the system, which permits constraining the ordering and results of operations. If the system is deterministic, operation consistency properties can reconstruct the state of the system from the operations, and thereby indirectly constrain the state much like state consistency. But doing so is not generally possible when the system is non-deterministic (e.g., due to concurrency, timing, or external events).

State consistency, on the other hand, can see the entire state of the system, which permits constraining operations that might break the state. If the system records all its operations in its state, then state consistency can indirectly constrain the results of operations much like operation consistency.<sup>1</sup> However, it is often prohibitive to record all operations so this is only a theoretical capability.

---

<sup>1</sup>It is even possible to constrain all operations of the entire execution, though enforcing such constraints would be hard.

### 3.3.4 Application dependence

State consistency tends to be application dependent, because the properties concern state, and the correct state of a system varies significantly from application to application. As a result, developers need to figure out the right properties for each system, which takes time and effort. Moreover, in some cases there are no general mechanisms to enforce state consistency and developers must write application code that is closely tied to each property. There are two noteworthy exceptions: mutual consistency and eventual consistency. These properties apply broadly to any replicated system, by referring to the replicated state irrespective of the application, and there are general replication mechanisms to enforce such properties.

Operation consistency is often application independent. It achieves application independence in two ways. First, some properties factor out the application-specific behavior, by reducing the behavior of the system under concurrent operations to behavior under sequential operations (as in the sequential equivalence subcategory), or behavior under a reference (as in the reference equivalence subcategory). Second, some properties focus on specific operations, such as read and write, that apply to many systems (as in the read-write centric subcategory). Theoretically, operation consistency *can* be highly application dependent, but this is not common. An example might be an email system accessible by many devices, where each operation (read, delete, move) might have different constraints on their response according to their semantics and the expectations of users.

### 3.3.5 Which type to use?

To decide what type of consistency to use, we suggest taking a few things into consideration. First, think about the negation of consistency: what are the inconsistencies that must be avoided? If the answer is most easily described by an undesirable state (e.g., two replicas diverge), then use state consistency. If the answer is most easily described by an incorrect result to an operation (e.g., a read returns stale data), then use operation consistency.

A second important consideration is application dependency. Many operation consistency and some state consistency properties are application independent (e.g., serializability, linearizability, mutual consistency, eventual consistency). We recommend trying to use such properties, before defining an application-specific one, because the mechanisms to enforce them are well understood. If the system requires an application specific property, and state and operation consistency are both natural choices, then we recommend using state consistency due to its simplicity.

## 4 Consistency in different disciplines

We now discuss what consistency means in each discipline, why it is relevant in that discipline, and how it relates to the two types of consistency in Section 3. We also point out concepts that are considered to be consistency in one discipline but not in another.

### 4.1 Distributed systems

In distributed systems, consistency refers to either state or operation consistency. Early replication protocols focused on providing mutual consistency while many cloud distributed systems provide eventual consistency. These are examples of state consistency. Some systems aim at providing linearizability or various flavors of read-write centric consistency. These are examples of operation consistency.

Consistency is an important consideration in distributed systems because such systems face many concerns that preclude or hinder consistency: clients separated by a slow network, machines that fail, clients that disconnect from each other, scalability of the system to a large number of clients, and high availability. These concerns can make it hard to provide strong levels of consistency, because consistency requires client coordination that

may not be possible. As a result, distributed systems may adopt weaker levels of consistency, chosen according to the needs of applications.

Cloud systems, an interesting type of distributed system, face all of the above concerns with intensity: the systems are geo-distributed (distributed around the globe) with significant latency separating data centers; machines fail often because there are many of them; clients disconnect from remote data centers due to problems or congestion in wide-area links; many clients are active and the system must serve all of them well; and the system must be available whenever possible since businesses lose money during downtime. Because of these challenges, cloud systems often resort to weak levels of consistency.

## 4.2 Database systems

In database systems, consistency refers to state consistency. For example, consider the ACID acronym that describes the guarantees of transactions. The “C” stands for consistency, which in this case means that the database is always in a state that developers consider valid: the system must preserve invariants such as uniqueness constraints, referential integrity, and application-specific properties (e.g.,  $x$  is a friend of  $y$  iff  $y$  is a friend of  $x$ ). These are flavors of state consistency.

The “A” stands for atomicity and the “I” stands for isolation. Interestingly, atomicity and isolation are examples of operation consistency. Atomicity requires that a transaction either executes in its entirety or does not execute at all, while isolation requires that transactions appear to execute by themselves without much interference. There are many different levels of isolation (serializability, snapshot isolation, read committed, repeatable reads, etc), but they all constrain the behavior of operations.

Although the database systems community separates transaction isolation from consistency and atomicity, in the distributed systems community, transaction isolation is seen as a form of consistency, while in the computer architecture community, a concept analogous to isolation is called atomicity. We do not know exactly why these terms have acquired different meanings across communities. But we suspect that a reason is that there are intertwined ideas across these concepts, which is something we try to identify and clarify in this article.

Consistency is important in database systems because data is of primary concern; in fact, data could be even more important than the result of operations in such systems (e.g., operations can fail as long as data is not destroyed). Different types of consistency arise because of the different classes of invariants that exist in the database, each with its own enforcement mechanism. For example, uniqueness constraints are enforced by an index and checks in the execution engine; application-specific constraints are enforced by the application logic; and mutual consistency is enforced by the replication manager.

## 4.3 Computer architecture

In computer architecture, consistency refers to operation consistency. A similar concept called coherence is also a form of operation consistency. Consistency and coherence have a subtle difference. Consistency concerns the entire memory system; it constrains the behavior of reads and writes—called loads and stores—across all the memory locations; an example is the sequential consistency property. Coherence concerns the cache subsystem; it can be seen as consistency of the operation of the various caches responsible for a given memory location. Thus, coherence constrains the behavior of loads and stores to an individual memory location.

Coherence and consistency are separated to permit a modular architecture of the system: a cache coherence protocol ensures the correct behavior of the caching subsystem, while the rest of the system ensures consistency across memory accesses without worrying about the cache subsystem.

Consistency and coherence arise as issues in computer architecture because increasingly computer systems have many cores or processors sharing access to a common memory: in such systems, there are concurrent operations on memory locations and data replication across many caches, which lead to problems of data sharing.

## 5 Conclusion

Consistency is a concern that spans many disciplines, as we briefly described here. This concern stems from the rise of concurrency and replication across these disciplines, a trend that we expect to continue. Unfortunately, consistency is subtle and hard to grasp, and to make matters worse, it has different names and meanings across communities. We hope to have shed some light on this subject by identifying two broad and very different types of consistency—state consistency and operation consistency—that can be seen across the disciplines.

## References

- [1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, Sept. 1990.
- [2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *International Conference on Software Engineering*, pages 562–570, Oct. 1976.
- [3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, Jan. 1995.
- [4] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O’Neil, and P. E. O’Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD International Conference on Management of Data*, pages 1–10, May 1995.
- [5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [6] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, 5(3):203–216, May 1979.
- [7] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2):181–239, Oct. 1992.
- [8] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *International Conference on Data Engineering*, pages 424–435, Mar. 2004.
- [9] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, Sept. 1985.
- [10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *International Symposium on Computer Architecture*, pages 15–26, June 1990.
- [11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fifth edition, Sept. 2011.
- [12] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
- [13] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *International Conference on Distributed Computing Systems*, pages 302–309, May 1990.
- [14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.
- [15] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *ACM SIGMOD International Conference on Management of Data*, pages 419–320, June 2005.
- [16] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *ACM Symposium on Principles of Distributed Computing*, pages 108–117, July 2002.
- [17] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979.

- [18] K. Ramamritham and C. Pu. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):997–1007, Dec. 1995.
- [19] A. Tanenbaum and M. V. Steen. *Distributed systems*. Pearson Prentice Hall, 2007.
- [20] D. B. Terry. Replicated data consistency explained through baseball. *Communications of the ACM*, 56(12):82–89, Dec. 2013.
- [21] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *International Conference on Parallel and Distributed Information Systems*, pages 140–149, Sept. 1994.
- [22] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *ACM Symposium on Operating Systems Principles*, pages 309–324, Nov. 2013.
- [23] 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 *ACM Symposium on Operating Systems Principles*, pages 172–183, Dec. 1995.
- [24] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2009.
- [25] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, Aug. 2002.