Database consistency models

Marc Shapiro
Sorbonne-Université & Inria, Paris, France

Pierre Sutra
Télécom SudParis, Évry, France

18 March 2018

Synonyms

Consistency model, data consistency, consistency criterion, isolation level.

The distributed systems and database communities use the same word, consistency, with different meanings. Within this entry, and following the usage of the distributed algorithms community, "consistency" refers to the observable behaviour of a data store.

In the database community, roughly the same concept is called "isolation," whereas the term "consistency" refers to the property that application code is sequentially safe (the C in ACID).

Definition

A data store allows application processes to put and get data from a shared memory. In general, a data store cannot be modelled as a strictly sequential process. Applications observe non-sequential behaviours, called anomalies. The set of possible behaviours, and conversely of possible anomalies, constitutes the *consistency model* of the data store.

Overview

Background

A *data store*, or database system, is a persistent shared memory space, where different client application processes can store data items. To ensure scalability and dependability, a modern data store distributes and replicates its data across clusters of *servers* running in parallel. This approach supports high throughput by spreading the load, low latency by parallelising requests, and fault-tolerance by replicating data and processing; system capacity increases by simply adding more servers (scale-out).

Ideally, from the application perspective, data replication and distribution should be transparent. Read and update operations on data items would appear to execute as in a single sequential thread; reading a data item would return exactly the last value written to it in real time; and each application transaction (a grouping of operations) would take effect atomically (all at once). This ideal behaviour is called *strict serialisability*, noted SSER (Papadimitriou 1979).

In practice, exposing some of the internal parallelism to clients enables better performance. More fundamentally, SSER requires assumptions that are unrealistic at large scale, such as absence of network partitions (see Section "Fundamental results" hereafter). Therefore, data store design faces a *fundamental tension* between providing a strictly serialisable behaviour on the one hand, versus availability and performance on the other. This explains why large-scale data stores hardly ever provide the SSER model, with the notable exception of Spanner (Corbett et al. 2012).

Consistency models

Informally, a *consistency model* defines what an application can observe about the updates and reads of its data store. When the observed values of data differ from strict serialisability, this is called an *anomaly*. Examples of anomalies include *divergence*, where concurrent reads of the same item persistently return different values; *causality violation*, where updates are observed out of order; *dirty reads*,

where a read observes the effect of a transaction that has not terminated; or *lost updates*, where the effect of an update is lost. The more anomalies allowed by a store, the *weaker* its consistency model. The *strongest* model is (by definition) SSER, the baseline against which other models are compared.

More formally, a consistency model is defined by the history of updates and reads that clients can observe. A model is weaker than another if it allows more histories.

An absolute definition of "strong" and "weak consistency" is open to debate. For the purpose of this entry, we say that a consistency model is *strong* when it has consensus power, i.e., any number of failure-prone processes can reach agreement on some value by communicating through data items in the store. If this is not possible, then the consistency model is said *weak*.

In a strong model, updates are totally ordered. Some well-known strong models include SSER, serialisability (SER), or snapshot isolation (SI). Weak models admit concurrent updates to the same data item, and include Causal Consistency (CC), Strong Eventual Consistency (SEC) and Eventual Consistency (EC) (see Table 1).

Key Research Findings

Basic concepts

A *data store* is a logically-shared memory space where different application processes store, update and retrieve data *items*. An item can be very basic, such as a register with read/write operations, or a more complex structure, e.g., a table or a file system directory. An application process executes *operations* on the data store through the help of an *API*. When a process *invokes* an operation, it executes a remote call to an appropriate *end-point* of the data store. In return, it receives a *response value*. A common example of this mechanism is a POST request to an HTTP end-point.

An application consists of *transactions*. A transaction consists of any number of reads and updates to the data store. It is terminated either by an abort, whereby its writes have no effect, or by a commit, whereby writes modify the store. In what

follows, we consider only committed transactions. The transaction groups together low-level storage operations into a higher-level abstraction, with properties that help developers reason about application behaviour.

The properties of transactions are often summarised as ACID: All-or-Nothing, (individual) Correctness, Isolation, and Durability.

All-or-Nothing ensures that, at any point in time, either all of a transaction's writes are in the store, or none of them is. This guarantee is essential in order to support common data invariants such as equality or complementarity between two data items. *Individual Correctness* is the requirement that each of the application's transactions individually transitions the database from a safe state (i.e., where some application-specific integrity invariants hold over the data) to another safe state. *Durability* means that all later transactions will observe the effect of this transaction after it commits. A, C and D are essential features of any transactional system, and will be taken for granted in the rest of this entry.

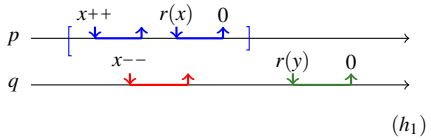
The I property, *Isolation* characterises the absence of interference between transactions. Transactions are isolated if one cannot interfere with the other, regardless of whether they execute in parallel or not.

Taken together, the ACID properties provide the *serialisability* model, a program semantics in which a transaction executes as if it was the only one accessing the database.This model restricts the allowable interactions among concurrent transactions, such that each one produces results indistinguishable from the same transactions running one after the other. As a consequence, the code for a transaction lives in the simple, familiar sequential programming world, and the developer can reason only about computations that start with the final results of other transactions. Serialisability allows concurrent operations to access the data store and still produce predictable, reproducible results.

Definitions

A *history* is a sequence of invocations and responses of operations on the data items by the application processes. It is commonly represented with timelines. For instance, in history h_1 below, processes p and q access a data store that contains

two counters (x and y).



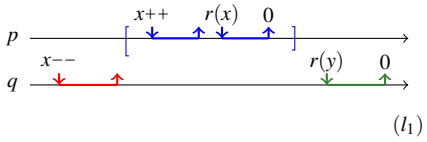
Operations $(z{+}{+})$ and $(z{-}{-})$ respectively increment and decrement counter z by 1. To fetch the content of z, a process calls $r(z)$. A counter is initialised to 0. The start and the end of a transaction are marked using brackets, e.g., transaction $T_1 = (x{+}{+}).r(x)$ in history h_1. When the transaction contains a single operation, the brackets are omitted for clarity.

As pointed above, we assume in this entry that all the transactions are committed in a history. Thus every invocation has a matching response. More involved models exist, e.g., when considering a transactional memory (Guerraoui and Kapalka 2008).

A history induces a real-time order between transactions (denoted $\prec_h$). This order holds between two transactions T and T' when the response of the last operation in T precedes in h the invocation of the first operation in T'. A history also induces a per-process order that corresponds to the order in which processes invoke their transactions. For instance in h_1, transaction $T_2 = (x{-}{-})$ precedes transaction $T_3 = r(y)$ at process q. This relation together with $(T_1 <_{h_1} T_3)$ fully defines the real-time order in history h_1.

Histories have various properties according to the way invocations and responses interleave. Two transactions are concurrent in a history h when they are not ordered by the relation $\prec_h$. A history h is *sequential* when no two transactions in h are concurrent. A sequential history is *legal* when it respects the sequential specification of each object. Two histories h and h' are *equivalent* when they contain the same set of events (invocations and responses).

A consistency model defines the histories that are allowed by the data store. In particular, serialisability (SER) requires that every history h is equivalent to some sequential and legal history l. For instance, history h_1 is serialisable, since it is equivalent to the history l_1 below. In addition, if the equivalent sequential history preserves the real-time order between transaction, history h is said *strictly*



(l_1)

serialisable (SSER) (Papadimitriou 1979). This is the case of h_1 since in l_1 the relations ($T_2 <_{h_1} T_3$) and ($T_1 <_{h_1} T_3$) also hold.

When each transaction contains a single operation, SSER boils down to linearizability (LIN) (Herlihy and Wing 1990). The data store ensures sequential consistency (SC) (Lamport 1979) when each transaction contains a single operation and only the per-process order is kept in the equivalent sequential history.

The above consistency models (SER, SSER, LIN and SC) are strong, as they allow the client application processes to reach consensus. To see this, observe that processes may agree as follows: The processes share a FIFO queue L in the data store. To reach consensus, each process enqueues some value in L which corresponds to a proposal to agree upon. Then, each process chooses the first proposal that appears in L. The equivalence with a sequential history implies that all the application processes pick the same value.

Conversely, processes cannot reach consensus if the consistency model is *weak*. A widespread model in this category is Eventual Consistency (EC) (Vogels 2008), used for instance in the Simple Storage Service (Murty 2008). EC requires that, if clients cease submitting transactions, they eventually observe the same state of the data store. This eventually-stable state may include part (or all) the transactions executed by the clients. Under EC, processes may repeatedly observe updates in different orders. For example, if the above list L is EC, each process may see its update applied first on L until it decides, preventing agreement. In fact, EC is too weak to allow asynchronous failure-prone processes to reach an agreement (Attiya et al. 2017).

Fundamental results

In the most general model of computation, replicas are asynchronous. In this model, and under the hypothesis that a majority of them are correct, it is possible to emulate a linearizable shared memory (Attiya et al. 1990). This number of

correct replicas is tight. In particular, if any majority of the replicas may fail, the emulation does not work (Delporte-Gallet et al. 2004).

The above result implies that, even for a very basic distributed service, such as a register, it is not possible to be at the same time consistent, available and tolerant to partition. This result is known as the CAP Theorem (Gilbert and Lynch 2002), which proves that it is not possible to provide all the following desirable features at the same time: *(C)* strong Consistency, even for a register, *(A)* Availability, responding to every client request, and *(P)* tolerate network Partition or arbitrary messages loss.

A second fundamental result, known as FLP, is the impossibility to reach consensus deterministically in presence of crash failures (Fischer et al. 1985). FLP is true even if all the processes but one are correct.

As pointed above, a majority of correct processes may emulate a shared memory. Thus, the FLP impossibility result indicates that a shared memory is not sufficient to reach consensus. In fact, solving consensus requires the additional ability to elect a leader among the correct processes (Chandra et al. 1996).

Data stores that support transactions on more than one data item are subject to additional impossibility results. For instance, an appealing property is genuine partial replication (GPR) (Schiper et al. 2010), a form of disjoint-access parallelism (Israeli and Rappoport 1994). Under GPR, transactions that access disjoint items do not contend in the data store. GPR avoids convoy effects between transactions (Blasgen et al. 1979) and ensure scalability under parallel workload. However, GPR data stores must sacrifice some form of consistency, or provide little progress guarantees (Bushkov et al. 2014; Saeida Ardekani et al. 2013a; Attiya et al. 2009).

A data store API defines the shared data structures the client application processes manipulate as well as their consistency and progress guarantees. The above impossibility results inform the application developer that some APIs require synchronisation among the data replicas. Process synchronisation is costly, thus there is a trade-off between performance and data consistency.

Acronym	Full name	Reference
EC	Eventual Consistency	Ladin et al. (1990)
SEC	Strong Eventual Consistency	Shapiro et al. (2011)
CM	Client monotonicity	Terry et al. (1994)
CS	Causal Snapshot	Chan and Gray (1985)
CC	Causal Consistency	Ahamad et al. (1995)
Causal HAT	Causal Highly-Av. Txn.	Bailis et al. (2013)
LIN	Linearisability	Herlihy and Wing (1990)
NMSI	Non-Monotonic SI	Saeida Ardekani et al. (2013b)
PSI	Parallel SI	Sovran et al. (2011)
RC	Read Committed	Berenson et al. (1995)
SC	Sequential Consistency	Lamport (1979)
SER	Serialisability	Gray and Reuter (1993)
SI	Snapshot Isolation	Berenson et al. (1995)
SSER	Strict Serialisability	Papadimitriou (1979)
SSI	Strong Snapshot Isolation	Daudjee and Salem (2006)

Table 1: Models and source references

Trade-offs

In the common case, executing an operation under strong consistency requires to solve consensus among the data replicas, which costs at least one round-trip among replicas (Lamport 2006). Sequential consistency allows to execute either read or write operations at a local replica (Attiya and Welch 1994; Wang et al. 2014). Weaker consistency models, e.g., eventual (Fekete et al. 1999) and strong eventual consistency (Shapiro et al. 2011) enable both read and write operations to be local.

A second category of trade-offs relate consistency models to metadata (Peluso et al. 2015; Burckhardt et al. 2014). They establish lower bounds on the space complexity to meet a certain consitency models. For instance, tracking causality accurately requires $O(m)$ bits of storage, where m is the number of replicas (Charron-Bost 1991).

Common models

The previous sections introduce several consistency models (namely, SER, SC, LIN, SSER and EC). This section offers a perspective on other prominent models. Table 1 recapitulates.

Read-Committed (RC). Almost all existing transactional data stores ensure that clients observe only committed data (Zemke 2012; Berenson et al. 1995). More precisely, the RC consistency model enforces that if some read r observes the state $\hat{x}$ of an item x in history h, then the transaction T_i that wrote $\hat{x}$ commits in h. One can distinguish a *loose* and a *strict* interpretation of RC. The strict interpretation requires that $r(x)$ takes place after transaction T_i commits. Under the loose interpretation, the write operation might occur concurrently.

When RC, or a stricter consistency model holds, it is convenient to introduce the notion of *version*. A version is the state of a data item as produced by an update transaction. For instance, when T_i writes to some register x, an operation denoted hereafter $w(x_i)$, it creates a new version x_i of x. Versions allow to uniquely identify the state of the item as observed by a read operation, e.g., $r(x_i)$.

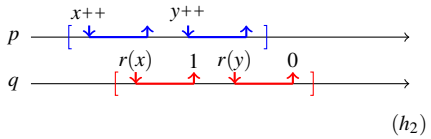
Strong Eventual Consistency (SEC). Eventual consistency (EC) states that, for every data item x in the store, if there is no new update on x, eventually clients observe x in the same state. Strong eventual consistency (SEC) further constrains the behaviour of the data replicas. In detail, a data store is SEC when it is EC and moreover, for every item x, any two replicas of x that applied the same set of updates on item x are in the same state.

Client Monotonic (CM). Client Monotonic (CM) ensures that a client always observes the results of its own past operations (Terry et al. 1994). CM enforces the following four so-called "session guarantees": *(i)* Monotonic reads (MR): if a client executes $r(x_i)$ then $r(x_{j\neq i})$ in history h, necessarily x_j follows x_i for some version order $\ll_{h,x}$ over the updates applied to x in h. *(ii)* Monotonic writes (MW): if a client executes $w(x_i)$ then $w(x_j)$, the version order $x_i \ll_{h,x} x_j$ holds;

(iii) Read-my-writes (RMW): when a client executes $w(x_i)$ followed by $r(x_{j\neq i})$, then $x_i \ll_{h,x} x_j$ holds; and *(iv)* Writes-follow-reads (WFR): if a client executes $r(x_i)$ followed by $w(x_j)$ it is true that $x_i \ll_{h,x} x_j$.

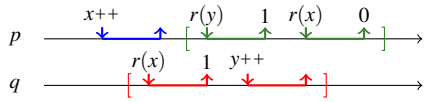
Most consistency models require CM, but this guarantee is so obvious that it might be sometimes omitted – this is for instance the case in Gray and Reuter (1992).

Read-Atomic (RA). Under RA, a transaction sees either all of the updates made by another transaction, or none of them (the All-or-Nothing guarantee). For instance, if a transaction T sees the version x_i written by T_i and transaction T_i also updates y, then T should observe at least version y_i. If history h fails to satisfies RA, a transaction in h exhibits a *fractured read* (Bailis et al. 2014). For instance, this is the case of the transaction executed by process q in history h_2 below.



Consistent Snapshot (CS). A transaction T_i *depends on* a transaction T_j when it reads a version written by T_j, or such a relation holds transitively. In other words, denoting $T_i \xrightarrow{\text{wr}} T_j$ when T_i reads from T_j, T_j is in the transitive closure of the relation $(\xrightarrow{\text{wr}})$ when starting from T_i.

When a transaction never misses the effects of some transaction it depends on, the transaction observes a *consistent snapshot* (Chan and Gray 1985). In more formal terms, a transaction T_i in a history h observes a consistent snapshot when for every object x, if *(i)* T_i reads version x_j, *(ii)* T_k writes version x_k, and *(iii)* T_i depends on T_k, then version x_k is followed by version x_j in the version order $\ll_{h,x}$. A history h belongs to CS when all its transactions observe a consistent snapshot. For instance, this is not the case of history h_3 below. In this history, transaction $T_3 = r(y).r(x)$ depends on $T_2 = r(x).(y{+}{+})$, and T_2 depends on $T_1 = (x{+}{+})$, yet T_3 does not observe the effect of T_1.

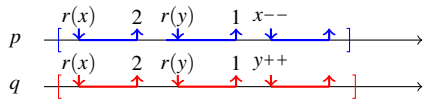


(h_3)

Causal Consistency (CC). Causal consistency (CC) holds when transactions observe consistent snapshots of the system, and the client application processes are monotonic. CC is a weak consistency model and it does not allow solving consensus. It is in fact the strongest model that is available under partition (Attiya et al. 2017). Historically, CC refers to the consistency of single operations on a shared memory (Ahamad et al. 1995). Causally consistent transactions (Causal HAT) is a consistency model that extends CC to transactional data stores (Bailis et al. 2013).

Snapshot Isolation (SI). SI is a widely-used consistency model (Berenson et al. 1995). This model is strong, but allows more interleavings of concurrent read transactions than SER. Furthermore, SI is causal (i.e., SI $\subseteq$ CC), whereas SER is not.

Under SI, a transaction observes a *snapshot* of the state of the data store at some point prior in time. Strong snapshot isolation (SSI) requires this snapshot to contain all the preceding transactions in real time (Daudjee and Salem 2006). Two transactions may commit under SI as long as they do not write the same item concurrent. SI avoids the anomalies listed in Section "Consistency Models", but exhibits the *write-skew* anomaly, illustrated in history h_4 below. In this history,



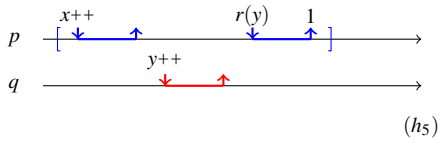
(h_4)

an application using data items x and y wishes to maintain the invariant $x \geq y$. The invariant holds initially, and each of the two transactions T_1 and T_2 guarantees the invariant individually. As illustrated in history h_4, running them concurrently under SI may violate the invariant.

An application is *robust* against a consistency model M when, it produces serialisable histories (Cerone and Gotsman 2016), despite running atop a data store

providing M, It is known (Fekete et al. 2005) that an application is robust against SI when every invariant is materialised by a data item.

Parallel / Non-Monotonic Snapshot Isolation (PSI/NMSI). Parallel and non-monotonic snapshot isolation are scalable variations of SI. These models retain two core properties of SI, namely *(i)* each transaction observes a consistent snapshot, and *(ii)* no two concurrent transactions update the same data items. PSI requires to take a snapshot at the start of the transaction. NMSI relaxes this requirement, enabling the snapshot to be computed incrementally, as illustrated in history h_5 below.



A three-dimensional view of data consistency

Shapiro et al. (2016) classify consistency models along three dimensions, to better understand and compare them. Their approach divides each operation into two parts: the *generator* reads data and computes response values, and the *effector* applies side-effects to every replica. Each of the three dimensions imposes constraints on the generators and effectors. Table 2 classifies the consistency criteria of Table 1 along these three dimensions.

- *Visibility dimension.* This dimension constrains the visibility of operations, i.e., how a generator sees the updates made by effectors. The strongest class of consistency models along this dimension is *external* visibility, which imposes that a generator sees the effectors of all the operations that precedes it in real time. Weakening this guarantee to the per-process order leads to *causal* visibility. A yet weaker class is *transitive* visibility, which only requires visibility to hold transitively. Finally, absence of constraints on generators, for instance during the unstable period of an eventually-consistent data store, is termed *rollback* visibility.

- *Ordering dimension.* This dimension constrains the ordering over generators and effectors. Four classes are of interest along this dimension: The strongest class is termed *total* order. For every history of a model in this class, there exists an equivalent serial history of all the operations. Weaker classes, below total order, constrain only effectors. The *gapless* order class requires effectors to be ordered online by natural numbers with no gaps; this requires consensus and is subject to the CAP impossibility result. The *capricious* class admits gaps in the ordering, allowing replicas to order their operations independently. A last-writer wins protocol (e.g., (Ladin et al. 1990)) produces a consistency model in this class. This class is subject to the lost-update anomaly. The weakest class along this dimension is termed *concurrent* and imposes no ordering on generators and effectors.

- *Composition dimension.* This dimension captures the fact that a transaction contains one or more operations. A model in the All-Or-Nothing class preserves the A in ACID. This means that if some effector of transaction T_1 is visible to transaction T_2, then all of T_1's effectors are visible to T_2. Typically, all the generators of a transaction read from the same set of effectors, i.e., its snapshot. The snapshot class extends the Visibility and Ordering guarantees to all generators of the transaction. For instance, in the case of a model both in the snapshot and total order classes, all the operations of a transaction are adjacent in the equivalent serial history.

Examples of Application

A key-value store (KVS) is a distributed data store that serves as building block of many cloud applications. This type of system belongs to the larger family of NoSQL databases and is used to store uninterpreted blobs of data (e.g., marshalled objects).

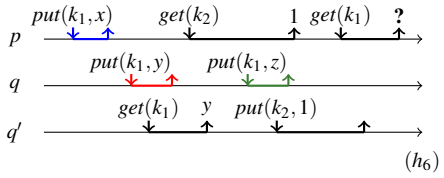
A KVS implements a map, that is a mutable relation from a set of *keys* to a set of *values*. In detail, the API of a key-value store consists of two operations: Operation $put(k,v)$ adds the pair (k,v) to the mapping, updating it if necessary.

Acronym	Ordering	Visibility	Composition
EC	Capricious	Rollbacks	Single Operation
CM	Concurrent	Monotonic	Single Operation
CS	Concurrent	Transitive	All-or-Nothing + Snapshot
CC	Concurrent	Causal	Single Operation
Causal HAT	Concurrent	Causal	All-or-Nothing + Snapshot
LIN	Total	External	Single Operation
NMSI	Gapless	Transitive	All-or-Nothing + Snapshot
PSI	Gapless	Causal	All-or-Nothing + Snapshot
RC	Concurrent	Monotonic	All-or-Nothing
SC	Total	Causal	Single Operation
SER	Total	Transitive	All-or-Nothing + Snapshot
SI	Gapless	Transitive	All-or-Nothing + Snapshot
SSER	Total	External	All-or-Nothing + Snapshot
SSI	Gapless	External	All-or-Nothing + Snapshot

Table 2: Three-dimension features of consistency models and systems

Depending on the KVS, this operation may return the previous value of the key k, or simply *nil*. Operation $get(k)$ retrieves the current value stored under key k.

The notions of "current" and "previous" values depend on the consistency model of the KVS. History h_6 below illustrates this point for an operation $get(k_1)$ by process p. When the KVS is RC, operation $get(k_1)$ returns the initial value



of k_1, or any value written concurrently or before this call. Denoting $\perp$ the initial value of k_1, this means that operation $get(k_1)$ may return any value in $\{\perp, x, y, z\}$.

If the KVS guarantees RMW, at least the last value written by p should be returned. As a consequence, the set of possible values reduces to $\{x, y, z\}$.

Now, let us consider that the KVS guarantees CC. Process p observes the op-

eration $put(k_2, 1)$ by r. This operation causally follows an observation by q' of y. Therefore, p should observe either y, or z.

If the KVS is linearizable, the value stored under key k_1 is the last value written before $get(k_1)$ in any sequential history equivalent to h_6. Every such history should preserve the real-time precedence of h_6. Clearly, the last update in h_6 sets the value of k_1 to z. Thus, if the KVS is linearizable, z is the only allowed response of operation $get(k_1)$ in h_6.

Future Directions for Research

Consistency models are formulated in various frameworks and using different underlying assumptions. For instance, some works (ANSI 1992; Berenson et al. 1995) define a model in terms of the anomalies it forbids. Others rely on specific graphs to characterise a model (Adya 1999), or predicates over histories (Viotti and Vukolić 2016). The existence of a global time (Papadimitriou 1979) is sometimes taken for granted. This contrasts with approaches (Lamport 1986) that avoid to make such an assumption. A similar observation holds for concurrent operations which may (Guerraoui and Kapalka 2008) or not (Ozsu and Valduriez 1991) overlap in time.

This rich literature makes difficult an apples-to-apples comparison between consistency models. Works exist (Chrysanthis and Ramamritham 1994) that attempt to bridge this gap by expressing them in a common framework. However, not all the literature is covered and it is questionable whether their definitions is equivalent to the ones given in the original publications.

The general problem of the implementability a given model is also an interesting avenue for research. One may address this question in term of the minimum synchrony assumptions to attain a particular model. In distributed systems, this approach has lead to the rich literature on failure detectors (Freiling et al. 2011). A related question is to establish lower and upper bounds on the time and space complexity of an implementation (when it is achivable). As pointed out in Section “Trade-offs”, some results already exist, yet the picture is incomplete.

From an application point of view, three questions are of particular interest.

First, the robustness of an application against a particular consistency model (Fekete et al. 2005; Cerone and Gotsman 2016). Second, the relation between a model and a consistency control protocol. These two questions are related to the grand challenge of synthesising concurrency control from the application specification (Gotsman et al. 2016). A third challenge is to compare consistency models in practice (Kemme and Alonso 1998; Wiesmann and Schiper 2005; Saeida Ardekani et al. 2014), so as to understand their pros and cons.

References

Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.d., MIT, March 1999.

Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, March 1995. doi: 10.1007/BF01784241. URL http://dx.doi.org/10.1007/BF01784241.

ANSI. American National Standard for Information Systems – Database Language – SQL, November 1992.

Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994. ISSN 0734-2071.

Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. Technical Report MIT/LCS/TM-423, Massachusetts Institute of Technology, Lab. for Comp. Sc., Cambridge, MA (USA), February 1990.

Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 69–78, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-606-9. doi: 10.1145/1583991.1584015. URL http://doi.acm.org/10.1145/1583991.1584015.

Hagit Attiya, Faith Ellen, and Adam Morrison. Limitations of highly-available eventually-consistent data stores. *IEEE Trans. on Parallel and Dist. Sys. (TPDS)*, 28(1):141–155, January 2017. doi: 10.1109/TPDS.2016.2556669. URL https://doi.org/10.1109/TPDS.2016.2556669.

Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, November 2013. doi: 10.14778/2732232.2732237. URL http://dx.doi.org/10.14778/2732232.2732237.

Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable atomic visibility with ramp transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 27–38, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2376-5. doi: 10.1145/2588555.2588562. URL http://doi.acm.org/10.1145/2588555.2588562.

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995. ISSN 0163-5808. doi: 10.1145/568271.223785. URL http://doi.acm.org/10.1145/568271.223785.

Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. The convoy phenomenon. *ACM SIGOPS Operating Systems Review*, 13(2):20–25, April 1979.

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 271–284, 2014. doi: 10.1145/2535838.2535848. URL http://doi.acm.org/10.1145/2535838.2535848.

Victor Bushkov, Dmytro Dziuma, Panagiota Fatourou, and Rachid Guerraoui. The pcl theorem: Transactions cannot be parallel, consistent and live. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*,

SPAA '14, pages 178–187, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2821-0. doi: 10.1145/2612669.2612690. URL http://doi.acm.org/10.1145/2612669.2612690.

Andrea Cerone and Alexey Gotsman. Analysing snapshot isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 55–64, 2016. doi: 10.1145/2933057.2933096. URL http://doi.acm.org/10.1145/2933057.2933096.

A. Chan and R. Gray. Implementing Distributed Read-Only Transactions. *IEEE Transactions on Software Engineering*, SE-11(2):205–212, February 1985. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1701989.

Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996. ISSN 0004-5411. doi: 10.1145/234533.234549. URL http://doi.acm.org/10.1145/234533.234549.

Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, 1991. doi: 10.1016/0020-0190(91)90055-M. URL https://doi.org/10.1016/0020-0190(91)90055-M.

Panos K. Chrysanthis and Krithi Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Trans. Database Syst.*, 19(3):450–491, September 1994. ISSN 0362-5915. doi: 10.1145/185827.185843. URL http://doi.acm.org/10.1145/185827.185843.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Symp. on Op. Sys. Design and Implementation (OSDI)*,

pages 251–264, Hollywood, CA, USA, October 2012. Usenix. URL https://www.usenix.org/system/files/conference/osdi12/osdi12-final-16.pdf.

Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 715–726. VLDB Endowment, 2006. URL http://dl.acm.org/citation.cfm?id=1182635.1164189.

Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 338–346, New York, NY, USA, 2004. ACM. ISBN 1-58113-802-4. doi: 10.1145/1011767.1011818. URL http://doi.acm.org/10.1145/1011767.1011818.

Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220:113–156, 1999. Special issue on Distributed Algorithms.

Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *Trans. on Database Systems*, 30 (2):492–528, June 2005. doi: http://doi.acm.org/10.1145/1071610.1071615.

Michael J. Fischer, Nancy A. Lynch, and Michael S. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985. doi: http://doi.acm.org/10.1145/3149.214121.

Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9:1–9:40, February 2011. ISSN 0360-0300. doi: 10.1145/1883612.1883616. URL http://doi.acm.org/10.1145/1883612.1883616.

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. ISSN 0163-5700. doi: http://doi.acm.org/10.1145/564585.564601.

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 371–384, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837625. URL http://doi.acm.org/10.1145/2837614.2837625.

Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992. ISBN 1558601902.

Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco CA, USA, 1993. ISBN 1-55860-190-2.

Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345233. URL http://doi.acm.org/10.1145/1345206.1345233.

Maurice Herlihy and Jeannette Wing. Linearizability: a correcteness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. URL http://doi.acm.org/10.1145/78969.78972.

Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 151–160, New York, NY, USA, 1994. ACM. ISBN 0-89791-654-9. doi: 10.1145/197917.198079. URL http://doi.acm.org/10.1145/197917.198079.

Bettina Kemme and Gustavo Alonso. A Suite of Database Replication Protocols based on Group Communication Primitives. In *International Conference on Distributed Computing Systems*, pages 156–163. IEEE, IEEE Comput. Soc,

1998. ISBN 0818682922. doi: 10.1109/ICDCS.1998.679498. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=679498.

Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploiting the semantics of distributed services. In *IEEE Computer Society Technical Committee on Operating Systems and Application Environments*, volume 4, pages 4–7. IEEE, IEEE Computer Society, 1990.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979. ISSN 0018-9340. doi: 10.1109/TC.1979.1675439. URL http://dx.doi.org/10.1109/TC.1979.1675439.

Leslie Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, Oct 2006. ISSN 1432-0452. doi: 10.1007/s00446-006-0155-x. URL https://doi.org/10.1007/s00446-006-0155-x.

James Murty. *Programming Amazon Web Services*. O'Reilly, first edition, 2008. ISBN 9780596515812.

M. Tamer Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. ISBN 0-13-691643-0.

Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322154.322158.

Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Binoy Ravindran, and Francesco Quaglia. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 217–226, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3617-8. doi: 10.1145/2767386.2767438. URL http://doi.acm.org/10.1145/2767386.2767438.

Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. On the Scalability of Snapshot Isolation. In *Proceedings of the 19th International Euro-Par Conference (EUROPAR)*, August 2013a.

Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 163–172, Braga, Portugal, October 2013b. IEEE Comp. Society. doi: 10.1109/SRDS.2013.25. URL http://dx.doi.org/10.1109/SRDS.2013.25.

Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. G-DUR: A Middleware for Assembling, Analyzing, and Improving Transactional Protocols. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 13–24, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2785-5. doi: 10.1145/2663165.2663336. URL http://doi.acm.org/10.1145/2663165.2663336.

Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *Proceedings of the 29th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, September 2010.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Dist. Sys. (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, October 2011. Springer-Verlag. doi: 10.1007/978-3-642-24550-3_29. URL http://www.springerlink.com/content/3rg39l2287330370/.

Marc Shapiro, Masoud Saeida Ardekani, and Gustavo Petri. Consistency in 3D. In Josée Desharnais and Radha Jagadeesan, editors, *Int. Conf. on Concurrency Theory (CONCUR)*, volume 59 of *Leibniz Int. Proc. in Informatics (LIPICS)*, pages 3:1–3:14, Québec, Québec, Canada, August 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany. doi: 10.4230/LIPIcs.CONCUR.2016.3. URL http://drops.dagstuhl.de/opus/volltexte/2016/6188/pdf/LIPIcs-CONCUR-2016-3.pdf.

Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pages 385–400, Cascais, Portugal, October 2011. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/2043556.2043592.

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*, pages 140–149, Austin, Texas, USA, September 1994.

Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19:1–19:34, June 2016. ISSN 0360-0300. doi: 10.1145/2926965. URL http://doi.acm.org/10.1145/2926965.

Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, October 2008. doi: http://doi.acm.org/10.1145/1466443.x.

J. Wang, E. Talmage, H. Lee, and J. L. Welch. Improved time bounds for linearizable implementations of abstract data types. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 691–701, May 2014. doi: 10.1109/IPDPS.2014.77.

Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):551–566, 2005. ISSN 10414347. doi: 10.1109/TKDE.2005.54. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1401893.

Fred Zemke. What.s new in sql:2011. *SIGMOD Rec.*, 41(1):67–73, April 2012. ISSN 0163-5808. doi: 10.1145/2206869.2206883. URL http://doi.acm.org/10.1145/2206869.2206883.

Cross-References

- Achieving LowLatency Transactions for Geo-Replicated Storage with Blotter
- Conflict-free Replicated Data Types (CRDTs)
- Coordination Avoidance
- Data replication and encoding
- Databases as a service
- Geo-replication Models
- Geo-scale Transaction Processing
- In-memory Transactions
- NoSQL Database Principles
- Storage hierarchies for big data
- TARDiS: A Branch-and-Merge Approach To Weak Consistency
- Transactional Middleware
- Weaker Consistency Models/Eventual Consistency.