

Consistency Tradeoffs in Modern Distributed Database System Design

Daniel J. Abadi, *Yale University*

The CAP theorem's impact on modern distributed database system design is more limited than is often perceived. Another tradeoff—between consistency and latency—has had a more direct influence on several well-known DDBSs. A proposed new formulation, PACELC, unifies this tradeoff with CAP.

Although research on distributed database systems began decades ago, it was not until recently that industry began to make extensive use of DDBSs. There are two primary drivers for this trend. First, modern applications require increased data and transactional throughput, which has led to a desire for elastically scalable database systems. Second, the increased globalization and pace of business has led to the requirement to place data near clients who are spread across the world. Examples of DDBSs built in the past 10 years that attempt to achieve high scalability or worldwide accessibility (or both) include SimpleDB/Dynamo/DynamoDB,¹ Cassandra,² Voldemort (<http://project-voldemort.com>), Sherpa/PNUTS,³ Riak (<http://wiki.basho.com>), HBase/BigTable,⁴ MongoDB (www.mongodb.org), VoltDB/H-Store,⁵ and Megastore.⁶

DDBSs are complex, and building them is difficult. Therefore, any tool that helps designers understand the tradeoffs involved in creating a DDBS is beneficial. The CAP theorem, in particular, has been extremely useful in helping designers to reason through a proposed system's

capabilities and in exposing the exaggerated marketing hype of many commercial DDBSs. However, since its initial formal proof,⁷ CAP has become increasingly misunderstood and misapplied, potentially causing significant harm. In particular, many designers incorrectly conclude that the theorem imposes certain restrictions on a DDBS during normal system operation, and therefore implement an unnecessarily limited system. In reality, CAP only posits limitations in the face of certain types of failures, and does not constrain any system capabilities during normal operation.

Nonetheless, the fundamental tradeoffs that inhibit DDBSs' capabilities during normal operation have influenced the different design choices of well-known systems. In fact, one particular tradeoff—between consistency and latency—arguably has been more influential on DDBS design than the CAP tradeoffs. Both sets of tradeoffs are important; unifying CAP and the consistency/latency tradeoff into a single formulation—PACELC—can accordingly lead to a deeper understanding of modern DDBS design.

CAP IS FOR FAILURES

CAP basically states that in building a DDBS, designers can choose two of three desirable properties: consistency (C), availability (A), and partition tolerance (P). Therefore, only CA systems (consistent and highly available, but not partition-tolerant), CP systems (consistent and partition-tolerant, but not highly available), and AP systems (highly available and partition-tolerant, but not consistent) are possible.

Many modern DDBSs—including SimpleDB/Dynamo, Cassandra, Voldemort, Sherpa/PNUTS, and Riak—do not

by default guarantee consistency, as defined by CAP. (Although consistency of some of these systems became adjustable after the initial versions were released, the focus here is on their original design.) In their proof of CAP, Seth Gilbert and Nancy Lynch⁷ used the definition of atomic/linearizable consistency: “There must exist a total order on all operations such that each operation looks as if it were completed at a single instant. This is equivalent to requiring requests of the distributed shared memory to act as if they were executing on a single node, responding to operations one at a time.”

Given that early DDBS research focused on consistent systems, it is natural to assume that CAP was a major influence on modern system architects, who, during the period after the theorem was proved, built an increasing number

It is wrong to assume that DDBSs that reduce consistency in the absence of any partitions are doing so due to CAP-based decision-making.

of systems implementing reduced consistency models. The reasoning behind this assumption is that, because any DDBS must be tolerant of network partitions, according to CAP, the system must choose between high availability and consistency. For mission-critical applications in which high availability is extremely important, it has no choice but to sacrifice consistency.

However, this logic is flawed and not consistent with what CAP actually says. It is not merely the partition tolerance that necessitates a tradeoff between consistency and availability; rather, it is the combination of

- partition tolerance and
- the existence of a network partition itself.

The theorem simply states that a network partition causes the system to have to decide between reducing availability or consistency. The probability of a network partition is highly dependent on the various details of the system implementation: Is it distributed over a wide area network (WAN), or just a local cluster? What is the quality of the hardware? What processes are in place to ensure that changes to network configuration parameters are performed carefully? What is the level of redundancy? Nonetheless, in general, network partitions are somewhat rare, and are often less frequent than other serious types of failure events in DDBSs.⁸

As CAP imposes no system restrictions in the baseline case, it is wrong to assume that DDBSs that reduce consistency in the absence of any partitions are doing so due to CAP-based decision-making. In fact, CAP allows

the system to make the complete set of ACID (atomicity, consistency, isolation, and durability) guarantees alongside high availability when there are no partitions. Therefore, the theorem does not completely justify the default configuration of DDBSs that reduce consistency (and usually several other ACID guarantees).

CONSISTENCY/LATENCY TRADEOFF

To understand modern DDBS design, it is important to realize the context in which these systems were built. Amazon originally designed Dynamo to serve data to the core services in its e-commerce platform (for example, the shopping cart). Facebook constructed Cassandra to power its Inbox Search feature. LinkedIn created Voldemort to handle online updates from various write-intensive features on its website. Yahoo built PNUTS to store user data that can be read or written to on every webpage view, to store listings data for Yahoo’s shopping pages, and to store data to serve its social networking applications. Use cases similar to Amazon’s motivated Riak.

In each case, the system typically serves data for webpages constructed on the fly and shipped to an active website user, and receives online updates. Studies indicate that latency is a critical factor in online interactions: an increase as small as 100 ms can dramatically reduce the probability that a customer will continue to interact or return in the future.⁹

Unfortunately, there is a fundamental tradeoff between consistency, availability, and latency. (Note that availability and latency are arguably the same thing: an unavailable system essentially provides extremely high latency. For purposes of this discussion, I consider systems with latencies larger than a typical request timeout, such as a few seconds, as unavailable, and latencies smaller than a request timeout, but still approaching hundreds of milliseconds, as “high latency.” However, I will eventually drop this distinction and allow the low-latency requirement to subsume both cases. Therefore, the tradeoff is really just between consistency and latency, as this section’s title suggests.)

This tradeoff exists even when there are no network partitions, and thus is completely separate from the tradeoffs CAP describes. Nonetheless, it is a critical factor in the design of the above-mentioned systems. (It is irrelevant to this discussion whether or not a single machine failure is treated like a special type of network partition.)

The reason for the tradeoff is that a high availability requirement implies that the system must replicate data. If the system runs for long enough, at least one component in the system will eventually fail. When this failure occurs, all data that component controlled will become unavailable unless the system replicated another version of the data prior to the failure. Therefore, the possibility of failure, even in the absence of the failure itself, implies that the availability requirement requires some degree of

data replication during normal system operation. (Note the important difference between this tradeoff and the CAP tradeoffs: while the *occurrence* of a failure causes the CAP tradeoffs, the failure *possibility* itself results in this tradeoff.)

To achieve the highest possible levels of availability, a DDBS must replicate data over a WAN to protect against the failure of an entire datacenter due, for example, to a hurricane, terrorist attack, or, as in the famous April 2011 Amazon EC2 cloud outage, a single network configuration error. The five reduced-consistency systems mentioned above are designed for extremely high availability and usually for replication over a WAN.

DATA REPLICATION

As soon as a DDBS replicates data, a tradeoff between consistency and latency arises. This occurs because there are only three alternatives for implementing data replication: the system sends data updates to all replicas at the same time, to an agreed-upon master node first, or to a single (arbitrary) node first. The system can implement each of these cases in various ways; however, each implementation comes with a consistency/latency tradeoff.

(1) Data updates sent to all replicas at the same time

If updates do not first pass through a preprocessing layer or some other agreement protocol, replica divergence—a clear lack of consistency—could ensue (assuming multiple updates to the system are submitted concurrently, for example, from different clients), as each replica might choose a different order in which to apply the updates. (Even if all updates are commutative—such that each replica will eventually become consistent, despite the fact that the replicas could possibly apply updates in different orders—Gilbert and Lynch’s strict definition of consistency⁷ still does not hold. However, generalized Paxos¹⁰ can provide consistent replication in a single round-trip.)

On the other hand, if updates first pass through a preprocessing layer or all nodes involved in the write use an agreement protocol to decide on the order of operations, then it is possible to ensure that all replicas will agree on the order in which to process the updates. However, this leads to several sources of increased latency. In the case of the agreement protocol, the protocol itself is the additional source of latency.

In the case of the preprocessor, there are two sources of latency. First, routing updates through an additional system component (the preprocessor) increases latency. Second, the preprocessor consists of either multiple machines or a single machine. In the former case, an agreement protocol to decide on operation ordering is needed across the machines. In the latter case, the system forces all updates, no matter where they are initiated—potentially anywhere in the world—to route all the way to the single

preprocessor first, even if another data replica is nearer to the update initiation location.

(2) Data updates sent to an agreed-upon location first

I will refer to this agreed-upon location as a “master node” (different data items can have different master nodes). This master node resolves all requests to update the data item, and the order that it chooses to perform these updates determines the order in which all replicas perform the updates. After the master node resolves updates, it replicates them to all replica locations.

As soon as a DDBS replicates data, a tradeoff between consistency and latency arises.

There are three replication options:

- a. The replication is synchronous: the master node waits until all updates have made it to the replica(s). This ensures that the replicas remain consistent, but synchronous actions across independent entities, especially over a WAN, increase latency due to the requirement to pass messages between these entities and the fact that latency is limited by the slowest entity.
- b. The replication is asynchronous: the system treats the update as if it were completed before it has been replicated. Typically, the update has at least made it to stable storage somewhere before the update’s initiator learns that it has completed (in case the master node fails), but there are no guarantees that the system has propagated the update. The consistency/latency tradeoff in this case depends on how the system deals with reads:
 - i. If the system routes all reads to the master node and serves them from there, then there is no reduction in consistency. However, there are two latency problems with this approach:
 1. Even if there is a replica close to the read-request initiator, the system must still route the request to the master node, which potentially could be physically much farther away.
 2. If the master node is overloaded with other requests or has failed, there is no option to serve the read from a different node. Rather, the request must wait for the master node to become free or recover. In other words, lack of load balancing options increases latency potential.

- ii. If the system can serve reads from any node, read latency is much better, but this can also result in inconsistent reads of the same data item, as different locations have different versions of a data item while the system is still propagating updates, and it could send a read to any of these locations. Although the level of reduced consistency can be bounded by keeping track of update sequence numbers and using them to implement sequential/timeline consistency or read-your-writes consistency, these are nonetheless reduced consistency options. Furthermore, write latency can be high if the master node for a write operation is geographically distant from the write requester.
- c. A combination of (a) and (b) is possible: the system sends updates to some subset of replicas synchronously, and the rest asynchronously. The consistency/

For data replication over a WAN, there is no way around the consistency/latency tradeoff.

latency tradeoff in this case again is determined by how the system deals with reads:

- i. If it routes reads to at least one node that has been synchronously updated—for example, when $R + W > N$ in a quorum protocol, where R is the number of nodes involved in a synchronous read, W is the number of nodes involved in a synchronous write, and N is the number of replicas—then consistency can be preserved. However, the latency problems of (a), (b)(i)(1), and (b)(i)(2) are all present, though to somewhat lesser degrees, as the number of nodes involved in the synchronization is smaller, and more than one node can potentially serve read requests.
- ii. If it is possible for the system to serve reads from nodes that have not been synchronously updated, for example, when $R + W \leq N$, then inconsistent reads are possible, as in (b)(ii).

Technically, simply using a quorum protocol is not sufficient to guarantee consistency to the level defined by Gilbert and Lynch. However, the protocol additions needed to ensure complete consistency¹¹ are not relevant here. Even without these additions, latency is already inherent in the quorum protocol.

(3) Data updates sent to an arbitrary location first

The system performs updates at that location, and then propagates them to the other replicas. The difference be-

tween this case and (2) is that the location the system sends updates to for a particular data item is not always the same. For example, two different updates for a particular data item can be initiated at two different locations simultaneously.

The consistency/latency tradeoff again depends on two options:

- a. If replication is synchronous, then the latency problems of (2)(a) are present. Additionally, the system can incur extra latency to detect and resolve cases of simultaneous updates to the same data item initiated at two different locations.
- b. If replication is asynchronous, then consistency problems similar to (1) and (2)(b) are present.

TRADEOFF EXAMPLES

No matter how a DDBS replicates data, clearly it must trade off consistency and latency. For carefully controlled replication across short distances, reasonable options such as (2)(a) exist because network communication latency is small in local datacenters; however, for replication over a WAN, there is no way around the consistency/latency tradeoff.

To more fully understand the tradeoff, it is helpful to consider how four DDBSs designed for extremely high availability—Dynamo, Cassandra, PNUTS, and Riak—replicate data. As these systems were designed for low-latency interactions with active Web clients, each one sacrifices consistency for improved latency.

Dynamo, Cassandra, and Riak use a combination of (2)(c) and (3). In particular, the system sends updates to the same node and then propagates these synchronously to W other nodes—that is, case (2)(c). The system sends reads synchronously to R nodes, with $R + W$ typically being set to a number less than or equal to N , leading to the possibility of inconsistent reads. However, the system does not always send updates to the same node for a particular data item—for example, this can happen in various failure cases, or due to rerouting by a load balancer. This leads to the situation described in (3) and potentially more substantial consistency shortfalls. PNUTS uses option (2)(b)(ii), achieving excellent latency at reduced consistency.

A recent study by Jun Rao, Eugene Shekita, and Sandeep Tata¹² provides further evidence of the consistency/latency tradeoff in these systems' baseline implementation. The researchers experimentally evaluated two options in Cassandra's consistency/latency tradeoff. The first option, "weak reads," allows the system to service reads from any replica, even if that replica has not received all outstanding updates for a data item. The second option, "quorum reads," requires the system to explicitly check for inconsistency across multiple replicas before reading data. The

second option clearly increases consistency at the cost of additional latency relative to the first option. The difference in latency between these two options can be a factor of four or more.

Another study by Hiroshi Wada and colleagues¹³ seems to contradict this result. These researchers found that requesting a consistent read in SimpleDB does not significantly increase latency relative to the default (potentially inconsistent) read option. However, the researchers performed these experiments in a single Amazon region (US West), and they speculate that SimpleDB uses master-slave replication, which is possible to implement with a modest latency cost if the replication occurs over a short distance. In particular, Wada and colleagues concluded that SimpleDB forces all consistent reads to go to the master in charge of writing the same data. As long as the read request comes from a location that is physically close to the master, and as long as the master is not overloaded, then the additional latency of the consistent read is not visible (both these conditions were true in their experiments).

If SimpleDB had replicated data across Amazon regions, and the read request came from a different region than the master's location, the latency cost of the consistent read would have been more apparent. Even without replication across regions (SimpleDB does not currently support replication across regions), official Amazon documentation warns users of increased latency and reduced throughput for consistent reads.


All four DDBSs allow users to change the default parameters to exchange increased consistency for worse latency—for example, by making $R + W$ more than N in quorum-type systems. Nonetheless, the consistency/latency tradeoff occurs during normal system operation, even in the absence of network partitions. This tradeoff is magnified if there is data replication over a WAN. The obvious conclusion is that reduced consistency is attributable to runtime latency, not CAP.

PNUTS offers the clearest evidence that CAP is not a major reason for reduced consistency levels in these systems. In PNUTS, a master node owns each data item. The system routes updates to that item to the master node, and then propagates these updates asynchronously to replicas over a WAN. PNUTS can serve reads from any replica, which puts the system into category (2)(b)(ii): it reduces consistency to achieve better latency. However, in the case of a network partition, where the master node becomes unavailable inside a minority partition, the system by default makes the data item unavailable for updates. In other words, the PNUTS default configuration is actually CP: in the case of a partition, the system chooses consistency over availability to avoid the problem of resolving conflicting updates initiated at different master nodes.

Therefore, the choice to reduce consistency in the baseline case is more obviously attributable to the continuous

consistency/latency tradeoff than to the consistency/availability tradeoff in CAP that only occurs upon a network partition. Of course, PNUTS's lack of consistency in the baseline case is also helpful in the network partition case, as data mastered in an unavailable partition is still accessible for reads.

CAP arguably has more influence on the other three systems. Dynamo, Cassandra, and Riak switch more fully to data replication option (3) in the event of a network partition and deal with the resulting consistency problems using special reconciliation code that runs upon detection of replica divergence. It is therefore reasonable to assume that these systems were designed with the possibility of a network partition in mind. Because these are AP systems, the reconciliation code and abil-



Ignoring the consistency/latency tradeoff of replicated systems is a major oversight, as it is present at all times during system operation.

ity to switch to (3) were built into the code from the beginning. However, once that code was there, it is convenient to reuse some of that consistency flexibility to choose a point in the baseline consistency/latency tradeoff as well. This argument is more logical than claims that these systems' designers chose to reduce consistency entirely due to CAP (ignoring the latency factor).

In conclusion, CAP is only one of the two major reasons that modern DDBSs reduce consistency. Ignoring the consistency/latency tradeoff of replicated systems is a major oversight, as it is present at all times during system operation, whereas CAP is only relevant in the arguably rare case of a network partition. In fact, the former tradeoff could be more influential because it has a more direct effect on the systems' baseline operations.

PACELC

A more complete portrayal of the space of potential consistency tradeoffs for DDBSs can be achieved by rewriting CAP as PACELC (pronounced "pass-elk"): if there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?

Note that the latency/consistency tradeoff (ELC) only applies to systems that replicate data. Otherwise, the system suffers from availability issues upon any type of failure or overloaded node. Because such issues are just instances of extreme latency, the latency part of the ELC tradeoff can incorporate the choice of whether or not to replicate data.

The default versions of Dynamo, Cassandra, and Riak are PA/EL systems: if a partition occurs, they give up consistency for availability, and under normal operation they give up consistency for lower latency. Giving up both Cs in PACELC makes the design simpler; once a system is configured to handle inconsistencies, it makes sense to give up consistency for both availability and lower latency. However, these systems have user-adjustable settings to alter the ELC tradeoff—for example, by increasing $R + W$, they gain more consistency at the expense of latency (although they cannot achieve full consistency as defined by Gilbert and Lynch, even if $R + W > N$).

Fully ACID systems such as VoltDB/H-Store and Megastore are PC/EC: they refuse to give up consistency, and will pay the availability and latency costs to achieve it. BigTable and related systems such as HBase are also PC/EC.

MongoDB can be classified as a PA/EC system. In the baseline case, the system guarantees reads and writes to be consistent. However, MongoDB uses data replication option (2), and if the master node fails or is partitioned from the rest of the system, it stores all writes that have been sent to the master node but not yet replicated in a local rollback directory. Meanwhile, the rest of the system elects a new master to remain available for reads and writes. Therefore, the state of the old master and the state of the new master become inconsistent until the system repairs the failure and uses the rollback directory to reconcile the states, which is a manual process today. (Technically, when a partition occurs, MongoDB is not available according to the CAP definition of availability, as the minority partition is not available. However, in the context of PACELC, because a partition causes more consistency issues than availability issues, MongoDB can be classified as a PA/EC system.)

PNUTS is a PC/EL system. In normal operation, it gives up consistency for latency; however, if a partition occurs, it trades availability for consistency. This is admittedly somewhat confusing: according to PACELC, PNUTS appears to get more consistent upon a network partition. However, PC/EL should not be interpreted in this way. PC does not indicate that the system is fully consistent; rather it indicates that the system does not reduce consistency beyond the baseline consistency level when a network partition occurs—instead, it reduces availability.

The tradeoffs involved in building distributed database systems are complex, and neither CAP nor PACELC can explain them all. Nonetheless, incorporating the consistency/latency tradeoff into modern DDBS design considerations is important enough to warrant bringing the tradeoff closer to the forefront of architectural discussions. **■**

Acknowledgments

This article went through several iterations and modifications thanks to extremely helpful and detailed feedback from Samuel Madden, Andy Pavlo, Evan Jones, Adam Marcus, Daniel Weinreb, and three anonymous reviewers.

References

1. G. DeCandia et al., “Dynamo: Amazon’s Highly Available Key-Value Store,” *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles (SOSP 07)*, ACM, 2007, pp. 205–220.
2. A. Lakshman and P. Malik, “Cassandra: Structured Storage System on a P2P Network,” *Proc. 28th ACM Symp. Principles of Distributed Computing (PODC 09)*, ACM, 2009, article no. 5; doi:10.1145/1582716.1582722.
3. B.F. Cooper et al., “PNUTS: Yahoo!’s Hosted Data Serving Platform,” *Proc. VLDB Endowment (VLDB 08)*, ACM, 2008, pp. 1277–1288.
4. F. Chang et al., “Bigtable: A Distributed Storage System for Structured Data,” *Proc. 7th Usenix Symp. Operating Systems Design and Implementation (OSDI 06)*, Usenix, 2006, pp. 205–218.
5. M. Stonebraker et al., “The End of an Architectural Era (It’s Time for a Complete Rewrite),” *Proc. VLDB Endowment (VLDB 07)*, ACM, 2007, pp. 1150–1160.
6. J. Baker et al., “Megastore: Providing Scalable, Highly Available Storage for Interactive Services,” *Proc. 5th Biennial Conf. Innovative Data Systems Research (CIDR 11)*, ACM, 2011, pp. 223–234.
7. S. Gilbert and N. Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services,” *ACM SIGACT News*, June 2002, pp. 51–59.
8. M. Stonebraker, “Errors in Database Systems, Eventual Consistency, and the CAP Theorem,” *blog, Comm. ACM*, 5 Apr. 2010; <http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem>.
9. J. Brutlag, “Speed Matters for Google Web Search,” unpublished paper, 22 June 2009, Google; <http://code.google.com/speed/files/delayexp.pdf>.
10. L. Lamport, *Generalized Consensus and Paxos*, tech. report MSR-TR-2005-33, Microsoft Research, 2005; <ftp://ftp.research.microsoft.com/pub/tr/TR-2005-33.pdf>.
11. H. Attiya, A. Bar-Noy, and D. Dolev, “Sharing Memory Robustly in Message-Passing Systems,” *JACM*, Jan. 1995, pp. 124–142.
12. J. Rao, E.J. Shekita, and S. Tata, “Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore,” *Proc. VLDB Endowment (VLDB 11)*, ACM, 2011, pp. 243–254.
13. H. Wada et al., “Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: The Consumers’ Perspective,” *Proc. 5th Biennial Conf. Innovative Data Systems Research (CIDR 11)*, ACM, 2011, pp. 134–143.

Daniel J. Abadi is an assistant professor in the Department of Computer Science at Yale University. His research interests include database system architecture, cloud computing, and scalable systems. Abadi received a PhD in computer science from Massachusetts Institute of Technology. Contact him at dna@cs.yale.edu; he blogs at <http://dbmsmusings.blogspot.com> and tweets at @daniel_abadi.