
# White Paper Beating the CAP Theorem

In this paper, we examine the implications of Brewer's CAP theorem for the builders and users of fault-tolerant database systems, and present a practical approach to overcoming its seemingly insurmountable restrictions on providing consistent, available, partition-tolerant databases

## Introduction

Brewer's CAP Theorem is widely known and accepted in the distributed systems community. And rightly so; it was formally proved to be correct[1]. The theorem itself is simple - it states that any form of distributed system with state, of which a distributed database is the canonical example, can exhibit at most two of the following desirable properties:

- Consistency - Operations which modify the state of the system should appear to happen "instantaneously" from all viewpoints; every reader in the system should see the same updates happen in the same sequence. In other words, the system provides a view of the distributed state which is consistent between observers.
- Availability - The system as a whole should continue functioning (although potentially with degradations in quality of service, such as being slower), even if servers should fail or be unreachable due to network failures
- Partition tolerance - The system as a whole should continue to function, potentially with degradations in service, even if the network can fail in arbitrary ways. Of course, as it is impossible to communicate without a working network, properties such as consistency are only considered within the scope of a group of communicating servers. In effect, if your system is chopped in half, both halves of the system should continue to operate independently until they are rejoined, then get back to some consistent state as soon as possible.

The unarguable truth of the CAP theorem is bad news for builders of distributed systems such as large Web sites. It means that at least one property of the system has to be done away with. This may represent an unacceptable compromise.

Earlier distributed databases, particularly those based on SQL update semantics choose to abandon partition tolerance, by building replicated systems that use a quorum algorithm to consistently update the database. In the event of a network partition, the smaller group of servers realises they are a minor-

ity group, and become read-only, rejecting all writes. This ensures that writes can be occurring in at most one group of connected servers - so when the network is restored, the isolated servers can just bring in the changes they missed, without needing any means of handling conflicts. However, if there is no partition that is larger than half the system (which, if you have two identical racks, and the cable between them breaks, is inevitable), then *no* partition will be able to handle writes. This is arguably no longer an 'available' system. SQL update semantics, in particular, are very hostile to merging arbitrary sets of updates after a partition has healed.

More recent distributed databases, particularly those of the NoSQL variety have opted to abandon consistency[2]. Servers share idempotent update operations with all the servers they can currently contact and, when servers that have been unreachable reappear, they send them all the missed updates. So updates propagate across the servers in a "best-effort" manner; quite likely arriving in different orders on different servers. This is why the updates must be idempotent. This provides excellent tolerance of both server and network failures, at the cost of loosely-defined "Eventually Consistent"[3] update semantics; in effect, pushing some of the burden of producing a distributed system to the application developer, who must make sure their application does not make any assumptions about the timing or ordering of updates.

There has also been interest in systems that are consistent, but not available or partition tolerant. Memcache is the primary example. As a fully distributed system with no replication, it loses data whenever a server dies. As each server is responsible for a distinct shard of the data, network failures mean that data is unreachable for some observers. Not only can it not be read, it cannot even be updated. However, this is fine: memcache *is a cache*. Consistency is important for caches, but it is fine if caches lose things as the data should always be available elsewhere.

In this white paper we will take a step back and look at the fundamental assumptions of the CAP theorem. We will not disprove it, merely work around it.

## Having your cake and eating it

Once it was believed that heavier-than-air flight was impossible, due to the eminently sensible observation that an object more dense than air would experience an inevitable downwards force. Nobody changed that. Instead, they attacked the implicit assumption that there was no way an object could generate a continuous upward force on itself without contact with a solid surface. Wings and an engine provided adequate lift and the aircraft was born.

The assumption we will kick out from beneath the CAP theorem is that you only have a single system. After all, if a sharded system can be C but not A or P and an eventually consistent NoSQL replication system can be A and P but not C, perhaps we can combine them in some way?

What happens if you put your data into both a sharded database and the eventually consistent replicator? When an update occurs in a system with no currently failing components, it will be seen in the sharded database immediately, and in the replicated store soon after. Eventually it will be evicted from the sharded database, but it will stay in the replicated store which is backed by persistent disk storage.

This suggests that if we satisfy reads from the sharded database where possible, and fall back to the local replica when the record is not found in the sharded database, we can have the consistency of the sharded database with the persistence of the replicated database. We can evict records from the sharded database once they are safely distributed in the replicated database, too.

But what happens when servers fail? The replicated database deals with that effortlessly; but when a shard server fails, two things happen.

- A section of the key space is lost. In our hybrid system, this means that updates which are currently in the process of being replicated, and which were residing on that particular shard server, will simply be eventually consistent rather than immediately consistent. However, bounding replication lag limits the scope of the "eventual". Thankfully, shard server can

be a relatively simple piece of software and has a limited amount of state (as records only need to be kept while they are being replicated), meaning that a failed shard server can be very quickly replaced. This means that the duration of a shard outage will hopefully be small.

- A section of the key space is unusable in the sharded layer until the server returns. This means that not only will updates to records hosted on the missing shard server in the course of replication at the point of failure become eventually consistent, so will further updates to those records until the shard server returns. Again, we fall back to eventual consistency, rather than data loss or unavailability.

So while a purely sharded database (even when given a persistent backing store, like memcachedb[4] loses read access to data when nodes go down or are unreachable and loses write access to data during a network failure, we can use a fully-replicated eventually consistent store as a "fallback". We read potentially outdated records from the replicated store when the sharded store cannot provide current state. Potentially outdated records are almost as good as the latest consistent state of the records, particularly when the amount of outdatedness is bounded by controlled replication lag.

## Conclusion

We have not disproved the CAP theorem but we have worked around it by combining two systems that, together, cover all three desirable properties. This paper shows how it is possible to produce a system that gives us C, then seamlessly falls back to one that provides A and P when C becomes impossible. The end result is a system that is available even in the face of network failures and provides consistent semantics when it is possible to do so.

Finding ways to work around unassailable limitations is, perhaps, the most sublimely satisfying aspect of engineering.

## References & Further Reading


[1] CAP Theorem: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.1495&rep=rep1&type=pdf

[2]Examples include CouchDB, Cassandra, and Voldemort

[3]http://en.wikipedia.org/w/index.php?title=Eventual_consistency&oldid=351460979

[4]http://memcachedb.org/

[5]Brewers Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant web services by Seth Gilbert and Nancy Lynch, 2002, ISSN:0163-5700

[6]Informal introduction to the CAP Theorem: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

[7]UK Patent Application 0920644.2 ("Consistency buffering")



GENIEDB®

**31920 Del Obispo**
**Ste 260, San Juan Capistrano**
**CA, 92675**
**+1 (855) GENIEDB**

**Units 3-4 Orchard Mews**
**42 Orchard Rd**
**London N6 5TR**
**+44 208 347 5700**

**info@geniedb.com**