The Cap Theorem

Understanding the five database genres is an important selection criterion, but it’s not the only one.

Another recurring theme in this book is the CAP theorem, which lays bare an unsettling truth about how distributed database systems behave in the face of network instability.

CAP proves that you can create a distributed database that is *consistent* (writes are atomic and all subsequent requests retrieve the new value), *available* (the database will always return a value as long as a single server is running), or *partition tolerant* (the system will still function even if server communication is temporarily lost—that is, a network partition), but you can have only two at once.

In other words, you can create a distributed database system that is *consistent* and *partition tolerant*, a system that is *available* and *partition tolerant*, or a system that is *consistent* and *available* (but not *partition tolerant*—which basically means not distributed).

But it is not possible to create a distributed database that is consistent and available and partition tolerant at the same time.

The CAP theorem is pertinent when considering a distributed database, since you must decide what you are willing to give up.

The database you choose will lose availability or consistency.

Partition tolerance is strictly an architectural decision (will the database be distributed or not).

Eventual Consistency

Distributed databases must be partition tolerant, so the choice between availability and consistency can be difficult.

However, while CAP dictates that if you pick availability you cannot have true consistency, you can still provide *eventual consistency*.

The idea behind eventual consistency is that each node is always available to serve requests.

As a trade-off, data modifications are propagated in the background to other nodes.

This means that at any time the system may be inconsistent, but the data is still largely accurate.

The Internet’s Domain Name Service (DNS) is a prime example of an eventually consistent system.

You register a domain, and it may take a few days to propagate to all DNS servers across the Internet.

But at no time is any particular DNS server unavailable (assuming you can connect to it, that is).

CAP in the Wild

Some partition-tolerant databases can be tuned to be more or less consistent or available on a per-request basis.

Riak works like this, allowing clients to decide at request time what level of consistency they require.

The other databases in this book largely occupy one corner or another of the CAP trade- off triangle.

Redis, PostgreSQL, and Neo4J are consistent and available (CA); they don’t distribute data and so partitioning is not an issue (though arguably, CAP doesn’t make much sense in non-distributed systems).

MongoDB and HBase are generally consistent and partition tolerant (CP).

In the event of a network partition, they can become unable to respond to certain types of queries (for example, in a Mongo replica set you flag slaveok to false for reads).

In practice, hardware failure is handled gracefully—other still-networked nodes can cover for the downed server—but strictly speaking, in the CAP theorem sense, they are unavailable.

Finally, CouchDB is available and partition tolerant (AP).

Even though two or more CouchDB servers can replicate data between them, CouchDB doesn’t guarantee consistency between any two servers.

It’s worth noting that most of these databases can be configured to change CAP type (Mongo can be CA, CouchDB can be CP), but here we’ve noted their default or common behaviors.

The Latency Trade-Off

There is more to distributed database system design than CAP, however.

For example, low latency (speed) is a chief concern for many architects.

If you read the Amazon Dynamo paper, you’ll notice a lot of talk about availability but also Amazon’s latency requirements.

For a certain class of applications, even a small latency change can translate to a large costs.

Yahoo’s PNUTS database famously gives up both availability on normal operation and consistency on partitions in order to squeeze a lower latency out of its design.

It’s important to consider CAP when dealing with distributed databases, but it’s equally important to be aware that distributed database theory does not stop there.