16.7.2 The CAP Theorem

The CAP theorem was originally formulated by Eric Brewer and states that a distributed system can exhibit at most two of these three desirable properties:

- Consistency: all nodes see the same data, and the same versions of these data, at the same time.
- Availability: every request receives a response indicating a success or failure result.
- Partition tolerance: the system continues to work even if nodes go down or are added. The distributed system can cope with it being divided into two or more disjoint network partitions due to node or network failure.

As to standalone DBMSs, the choice of which property to abandon is trivial, as there is no network involved (or at least no network connecting different database nodes). Therefore, no partitions can occur, and the standalone system can provide both data consistency and availability with ACID transactions.

Traditional tightly coupled distributed DBMSs (e.g., an RDBMS distributed over multiple nodes), will often sacrifice availability for consistency and partition tolerance. The distributed DBMS still enforces ACID properties over the participants in a distributed transaction, bringing all the data involved (including possible replicas) from one consistent state into another upon transaction execution. If individual nodes or network connections are unavailable, the DBMS would rather not perform the transaction (or not provide a query result to a read-only transaction) than yield an inconsistent result or bring the database into a temporary inconsistent state.

Many NoSQL database systems will give up on consistency instead, the reason being twofold. First, in many Big Data settings, unavailability is costlier

than (temporary) data inconsistency. For example, it is far preferable that multiple users get to see partially inconsistent versions of the same social media profile (e.g., with or without the latest status update), than having the system unavailable until all versions are synced, and all inconsistencies are resolved. In the event of node or network failure, the transaction will be executed, even if one or more nodes cannot participate, yielding a possibly inconsistent result or database state. Second, even if no failure occurs when executing the transaction, the overhead of locking all the necessary data, including replicas, and overseeing that consistency is guaranteed over all nodes involved in a distributed transaction has a severe impact on performance and transaction throughput.

Connections

NoSQL databases are discussed in Chapter 11, where the CAP theorem was also introduced.

Although the CAP theorem is widely referred to when explaining transaction paradigms for NoSQL databases, it can also be criticized. As became apparent from the previous discussion, it is not only the actual occurrence of network partitions that will result in the choice to abandon consistency. The performance degradation induced by the overhead of mechanisms that enforce transactional consistency *under normal system operation, even in the absence of network partitions* is often the true reason to abandon perpetual consistency. This overhead is further increased by the data replication that is necessary to guarantee availability and *to be prepared for failures*. This overhead exists even when no such failures occur. Also, one could argue that availability and performance are essentially the same concepts, with unavailability being an extreme case of high latency and low performance. Therefore, in many high-

volume settings, the real tradeoff is between consistency and performance. Obviously, the result of this tradeoff will be different, depending on the setting: a bank will never allow its customers to receive an inconsistent overview of their savings accounts status just for the sake of performance.

16.7.3 BASE Transactions

Although many NoSQL databases make a different tradeoff between consistency and availability/performance than more traditional DBMSs, they do not give up on consistency altogether. There would be no point in maintaining a database if the quality and consistency of its content cannot be guaranteed in the long run. Rather, they position themselves on a continuum between high availability and permanent consistency, where the exact position on this continuum can often be configured by the administrator. This paradigm is called **eventual consistency**: the results of a database transaction will eventually be propagated to all replicas and if no further transactions are executed then the system will eventually become consistent, but it is not consistent at all times, as is the case with ACID transactions.

To contrast this approach to ACID transactions (and staying within the chemical jargon), this transaction paradigm was coined as BASE transactions. BASE stands for Basically Available, Soft state, Eventually consistent:

- Basically Available: measures are in place to guarantee availability under all circumstances, if necessary at the cost of consistency.
- Soft state: the state of the database may evolve, even without external input, due to the asynchronous propagation of updates throughout the system.
- Eventually consistent: the database will become consistent over time, but may not be consistent at any moment and especially not at transaction commit.

Drill Down

ACID and BASE both refer to concepts from chemistry. In that sense, it’s not surprising that BASE was chosen as an acronym to contrast ACID.

Write operations are performed on one or at most a few of the replicas of a data item. Once these are updated, the write operation is considered as finished. Updates to the other replicas are propagated asynchronously in the background (possibly waiting for unavailable nodes or connections to become available again), so eventually all replicas receive the update. Read operations are performed on one or only a few of the replicas. If only a single replica is retrieved, there is no guarantee that this is the most recent one, but the eventual consistency guarantees it will not be too out of date either. If a read operation involves multiple replicas, these may not necessarily be consistent with one another. There are different options to resolve such inconsistencies:

- The DBMS contains rules to resolve conflicts before returning the retrieved data to the application. This often involves the use of timestamps (e.g., “last write wins”), which means that the most recently written version is returned. This approach can be used to retrieve the current session state, for example, if data pertaining to customer sessions of a web store are persisted in a NoSQL database.
- The burden of conflict resolution is shifted from the DBMS to the application. Here, the business logic can determine how conflicting replicas of the same data item can be reconciled. For example, the application may contain logic to combine the contents of two conflicting versions of the same customer’s shopping cart into a single unified version.