**Introduction to NoSQL Databases**

Over the last decade, the volume and velocity with which data is generated within organizations has grown exponentially. Consequently, there has been an explosion of database technologies that have been developed to address these growing data needs. These databases have typically had distributed implementations, since the volume of data being managed far exceeds the storage capacity of a single node. In order to support the massive scale of data, these databases have provided fewer of the features that we've come to expect from relational databases.

The first generation of these so-called NoSQL databases only provided rudimentary key-value get/put APIs. They were largely schema-free and didn't require well-defined types to be associated with the values being stored in the database. Over the last decade, however, a number of features that we've come to expect from standard databases—such as type systems and SQL, secondary indices, materialized views, and some kind of concept of transactions—have come to be incorporated and overlaid over those rudimentary key-value interfaces.

Today, there are hundreds of NoSQL databases available in the world, with a few popular ones, such as MongoDB, HBase, and Cassandra, having the lion's share of the market, followed by a long list of other, less popular databases.

These databases have different data models, ranging from the document model of MongoDB, to the column-family model of HBase and Cassandra, to the columnar model of Kudu. These databases are widely deployed in hundreds of organizations and at this point are considered mainstream and commonplace.

This book covers some of the most popular and widely deployed NoSQL databases. Each chapter covers a different NoSQL database, how it is architected, how to model your data, and how to interact with the database. Before we jump into each of the NoSQL databases covered in this book, let's look at some of the design choices that should be considered when one is setting out to build a distributed database.

Knowing about some of these database principles will give us insight into why different databases have been designed with different architectural choices in mind, based on the use cases and workloads they were originally designed for.

# Consistency versus availability

A database's **consistency** refers to the reliability of its functions' performance. A consistent system is one in which reads return the value of the last write, and reads at a given time epoch return the same value regardless of where they were initiated.

NoSQL databases support a range of consistency models, such as the following:

* **Strong consistency**: A system that is strongly consistent ensures that updates to a given key are ordered and reads reflect the latest update that has been accepted by the system
* **Timeline consistency**: A system that is timeline consistent ensures that updates to a given key are ordered in all the replicants, but reads at a given replicant might be stale and may not reflect the latest update that has been accepted by the system
* **Eventual consistency**: A system that is eventually consistent makes no guarantees about whether updates will be applied in order in all the replicants, nor does it make guarantees about when a read would reflect a prior update accepted by the system

A database's **availability** refers to the system's ability to complete a certain operation. Like consistency, availability is a spectrum. A system can be unavailable for writes while being available for reads. A system can be unavailable for admin operations while being available for data operations.

As is well known at this point, there's tension between consistency and availability. A system that is highly available needs to allow operations to succeed even if some nodes in the system are unreachable (either dead or partitioned off by the network). However, since it is unknown as to whether those nodes are still alive and are reachable by some clients or are dead and reachable by no one, there are no guarantees about whether those operations left the system in a consistent state or not.

So, a system that guarantees consistency must make sure that all of the nodes that contain data for a given key must be reachable and participate in the operation. The degenerate case is that a single node is responsible for operations on a given key. Since there is just a single node, there is no chance of inconsistency of the sort we've been discussing. The downside is that when a node goes down, there is a complete loss of availability for operations on that key.

**ACID guarantees**

Relational databases have provided the traditional properties of **ACID**: **atomicity**, **consistency**, **isolation**, and **durability**:

* **Atomicity** is self-explanatory and refers to the all-or-nothing nature of a set of operations.
* **Consistency**in ACID and **consistency** in the CAP theorem refer to different things. Consistency in ACID refers to the principle that the system must be left in a consistent state while processing transactions, it either reflects the state after successful completion of the transaction or must roll back to a state prior to the start of the transaction.
* **Isolation** refers to the interaction effects between transactions. Under what conditions is the state modified by one transaction visible to other active transactions in the system? It ranges from weak isolation levels, such as read-committed, and goes all the way to linearizable.
* **Durability**indicates that once a transaction has committed, the effects of the transaction remain despite events such as errors and crashes.

NoSQL databases vary widely in their support for these guarantees, with most of them not approaching the level of strong guarantees provided by relational databases (since these are hard to support in a distributed setting).

# Hash versus range partition

Once you've decided to distribute data, how should the data be distributed?

Firstly, data needs to be distributed using a partitioning key in the data. The partitioning key can be the primary key or any other unique key. Once you've identified the partitioning key, we need to decide how to assign a key to a given shard.

One way to do this would be to take a key and apply a hash function. Based on the hash bucket and the number of shards to map keys into, the key would be assigned to a shard. There's a bit of nuance here in the sense that an assignment scheme based on a modulo by the number of nodes currently in the cluster will result in a lot of data movement when nodes join or leave the cluster (since all of the assignments need to be recalculated). This is addressed by something called consistent hashing, a detailed description of which is outside the scope of this chapter.

Another way to do assignments would be to take the entire keyspace and break it up into a set of ranges. Each range corresponds to a shard and is assigned to a given node in the cluster. Given a key, you would then do a binary search to find out the node it is meant to be assigned to. A range partition doesn't have the churn issue that a naive hashing scheme would have. When a node joins, shards from existing nodes will migrate onto the new node. When a node leaves, the shards on the node will migrate to one or more of the existing nodes.

What impact do the hash and range partitions have on the system design? A hash-based assignment can be built in a decentralized manner, where all nodes are peers of each other and there are no special master-slave relationships between nodes. Ceph and Cassandra both do hash-based partition assignment.

On the other hand, a range-based partitioning scheme requires that range assignments be kept in some special service. Hence, databases that do range-based partitioning, such as Bigtable and HBase, tend to be centralized and peer to peer, but instead have nodes with special roles and responsibilities.

# In-place updates versus appends

Another key difference between database systems is how they handle updates to the physical records stored on disk.

Relational databases, such as MySQL, maintain a variety of structures in both the memory and disk, where writes from in-flight transactions and writes from completed transactions are persisted. Once the transaction has been committed, the physical record on disk for a given key is updated to reflect that. On the other hand, many NoSQL databases, such as HBase and Cassandra, are variants of what is called a **log-structured merge** (**LSM**) database.

In such an LSM database, updates aren't applied to the record at transaction commit. Instead, updates are applied in memory. Once the memory structure gets full, the contents of the memory are flushed to the disk. This means that updates to a single record can be fragmented and located within separate flush files that are created over time. This means that when there is a read for that record, you need to read in fragments of the record from the different flush files and merge the fragments in reverse time order in order to construct the latest snapshot of the given record. We will discuss the mechanics of how an LSM database works in the [Chapter 6](https://learning.oreilly.com/library/view/seven-nosql-databases/9781787288867/b7c57962-56bc-40cc-9ab2-57484d6d4039.xhtml), HBase.

# Row versus column versus column-family storage models

When you have a logical table with a bunch of rows and columns, there are multiple ways in which they can be stored physically on a disk.

You can store the contents of entire rows together so that all of the columns of a given row would be stored together. This works really well if the access pattern accesses a lot of the columns for a given set of rows. MySQL uses such a row-oriented storage model.

On the other hand, you could store the contents of entire columns together. In this scheme, all of the values from all of the rows for a given column can be stored together. This is really optimized for analytic use cases where you might need to scan through the entire table for a small set of columns. Storing data as column vectors allows for better compression (since there is less entropy between values within a column than there is between the values across a column). Also, these column vectors can be retrieved from a disk and processed quickly in a vectorized fashion through the SIMD capabilities of modern processors. SIMD processing on column vectors can approach throughputs of a billion data points/sec on a personal laptop.

Hybrid schemes are possible as well. Rather than storing an entire column vector together, it is possible to first break up all of the rows in a table into distinct row groups, and then, within a row group, you could store all of the column vectors together. Parquet and ORC use such a data placement strategy.

Another variant is that data is stored row-wise, but the rows are divided into row groups such that a row group is assigned to a shard. Within a row group, groups of columns that are often queried together, called column families, are then stored physically together on the disk. This storage model is used by HBase and is discussed in more detail in [Chapter 6](https://learning.oreilly.com/library/view/seven-nosql-databases/9781787288867/b7c57962-56bc-40cc-9ab2-57484d6d4039.xhtml), HBase.

# Strongly versus loosely enforced schemas

Databases can decide up-front how prescriptive they want to be about specifying a schema for the data.

When NoSQL databases came to the fore a decade ago, a key point was that they didn't require a schema. The schema could be encoded and enforced in the application rather than in the database. It was thought that schemas were a hindrance in dealing with all of the semi structured data that was getting produced in modern enterprise. So because the early NoSQL systems didn't have a type system, they didn't enforce the standard that all rows in the table have the same structure, they didn't enforce a whole lot.

However, today, most of these NoSQL databases have acquired an SQL interface. Most of them have acquired a rich type system. One of the reasons for this has been the realization that SQL is widely known and reduces the on-board friction in working with a new database. Getting started is easier with an SQL interface than it is with an obscure key-value API. More importantly, having a type system frees application developers from having to remember how a particular value was encoded and to decode it appropriately.

Hence, Cassandra deprecated the Thrift API and made CQL the default. HBase still doesn't support SQL access natively, but use of HBase is increasingly pivoting towards SQL interfaces over HBase, such as Phoenix.

# Summary

In this chapter, we introduced the notion of a NoSQL database and considered some of the principles that go into the design of such a database. We now understand that there are many trade-offs to be considered in database design based on the specific use cases and types of workloads the database is being designed for. In the following chapters, we are going to be looking in detail at seven popular NoSQL databases. We will look at their architecture, data, and query models, as well as some practical tips on how you can get started using these databases, if they are a fit for your use case.