SELECTED CHAPTERS

ScyllaDB IN ACTION

Bo Ingram

Sponsored by

ScyllaDB

MANNING

contents

1 Introducing ScyllaDB

This chapter covers

- What ScyllaDB is
- How ScyllaDB compares with other databases
- How ScyllaDB takes advantage of being a distributed system

ScyllaDB is a distributed NoSQL database designed to be a higher-performance rewrite of Apache Cassandra. Although its name rhymes with *Godzilla* and it has an adorable creature as a mascot, it's designed to be nonmonstrous to operate.

Compared with relational databases, ScyllaDB brings two big weapons to the Great Database Battle Royale: scalability and fault tolerance. ScyllaDB runs as a distributed system, running multiple nodes to store and serve data. This distribution simplifies scalability; to add capacity, operators need only add nodes. By enabling users to tune the number of nodes that respond to a query, ScyllaDB also provides fault tolerance; the system can handle the loss of a configurable amount of nodes before it's unable to serve requests (figure 1.1).

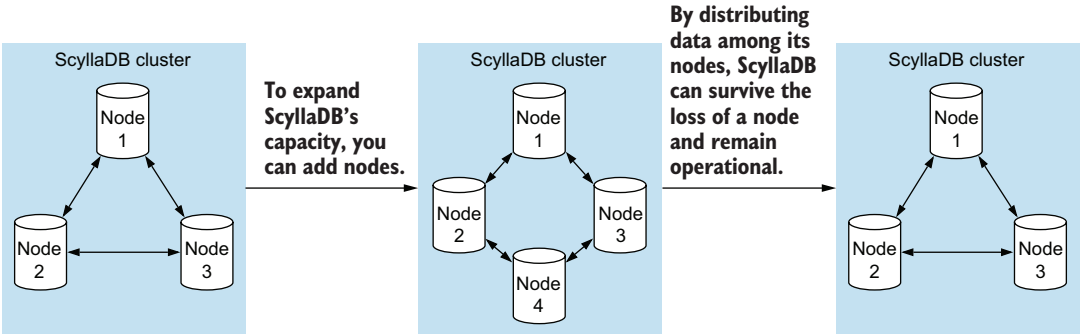


Figure 1.1 ScyllaDB is a distributed database that provides scalability and fault tolerance.

This distributed design affects everything around it, including how you design applications, how you query data, how you monitor the database, and how you recover the system during an outage. We'll explore all of these areas in this chapter to show how ScyllaDB can be practical distributed database for any application. Let's dive in!

1.1 ScyllaDB: A different database

ScyllaDB is a database; its name says so! Users give it data; the database gives that data back when asked to do so. This basic, oversimplified interface is similar to that of popular relational databases such as PostgreSQL and MySQL. ScyllaDB, however, isn't a relational database; it eschews joins and relational data modeling to provide a different set of benefits. Let's look at a fictitious example.

1.1.1 Hypothetical databases

Suppose that you've just moved to a new town. As you go to new restaurants, you want to remember what you ate so that you can order it (or avoid it) next time. You could write the info in a journal or save it in the notes app on your phone. But then you hear about a new business model in which people remember information you send them. Your friend Robert has started a similar venture called Robert's Rememberings.

ROBERT'S REMEMBERINGS

Robert's business (figure 1.2) is straightforward: you text Robert's phone number, and he remembers whatever information you send him. He also retrieves information for you, so you won't need to remember everything you've eaten in your new town. Remembering is Robert's job.

The plan works swimmingly at first, but soon, problems begin to appear. One time, you text him and he doesn't respond. He apologizes later, saying that he had a doctor's appointment. That situation isn't unreasonable; you want your friend to be healthy. Another time, you text him about a new meal, and instead of sending his usual instant response, he takes several minutes to reply. Later, he says that business is booming; he's been inundated with requests, so response time has suffered. He reassures you and tells you not to worry because he has a plan.

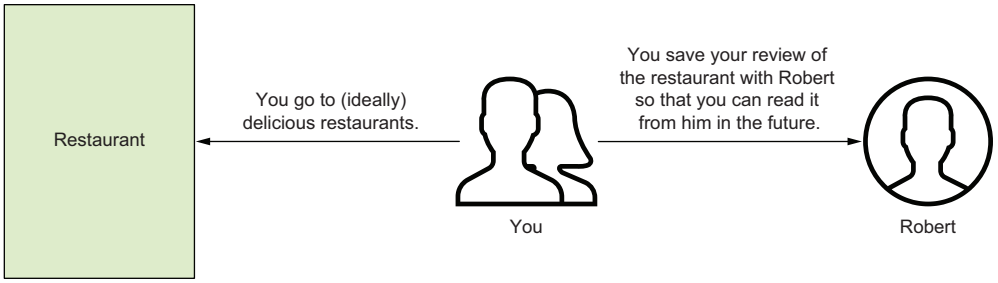


Figure 1.2 Robert's Rememberings has a seemingly simple plan.

Robert hires his friend Rosa to help and sends you the new updated rules for his system. If you only want to ask a question, you can text Rosa instead. All updates are still sent to Robert; he'll send everything you save to Rosa so that she'll have an up-to-date copy. At first, you slip up a few times and still ask Robert questions, but the new system seems to work well. Robert is no longer overwhelmed by read requests, and Rosa's responses are prompt. Figure 1.3 illustrates the improved system.

When Robert's business gets overwhelmed due to its popularity, he adds Rosa to serve read requests. Although this addition does reduce the individual number of requests, it also introduces additional ways the system can fail.

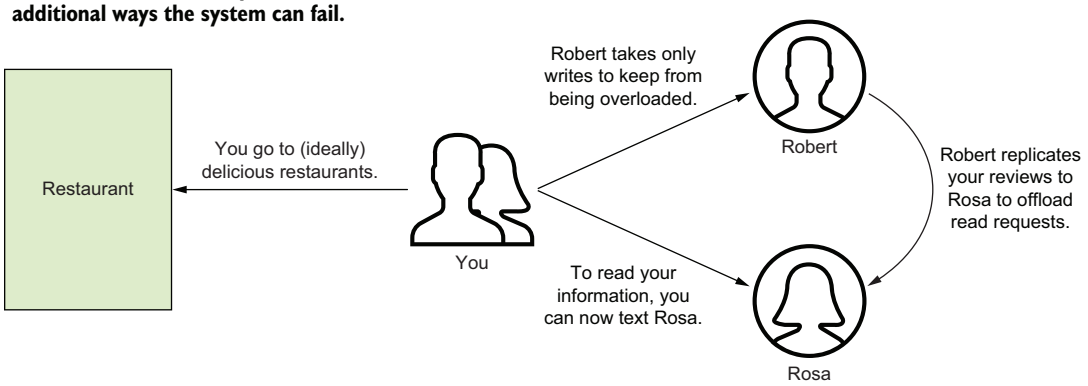


Figure 1.3 Robert adds a friend to his system to solve problems, but the update introduces complications.

One day, you realize that when you asked Rosa a question, she texted back an old review that you'd overwritten. You message Robert about this discrepancy, worried that your review of the much-improved tacos at Main Street Tacos has been lost forever. Robert tells you that there was a problem within the system; Rosa wasn't receiving messages from Robert but still got requests from customers. Your request hasn't been lost, and Robert and Rosa are reconciling the system to get back in sync.

You wanted the answer to one question: is the food at this restaurant good or not? Now you're worrying about contacting multiple people depending on whether you're

reading or writing a review, whether data is in sync, and whether your friend's system can scale to satisfy all users' requests. What happens if Robert can't handle people who are only saving their information? When you begin brainstorming intravenous energy-drink solutions, you realize that it's time to consider other options.

ABC DATA: A DIFFERENT APPROACH

Your research leads you to another business: ABC Data. ABC tells you that its system is a little different. The company has three employees—Alice, Bob, and Charlotte—and any of them can save information or answer questions. They communicate with one another to ensure that all of them have the latest data, as shown in figure 1.4. You're curious what would happen if one of them becomes unavailable, and they tell you about a cool feature: because there are three of them, they coordinate to provide redundancy for your data and increased availability. If Charlotte is unavailable, Alice and Bob will receive the request and answer. If Charlotte returns later, Alice and Bob get Charlotte up to speed on the latest changes.

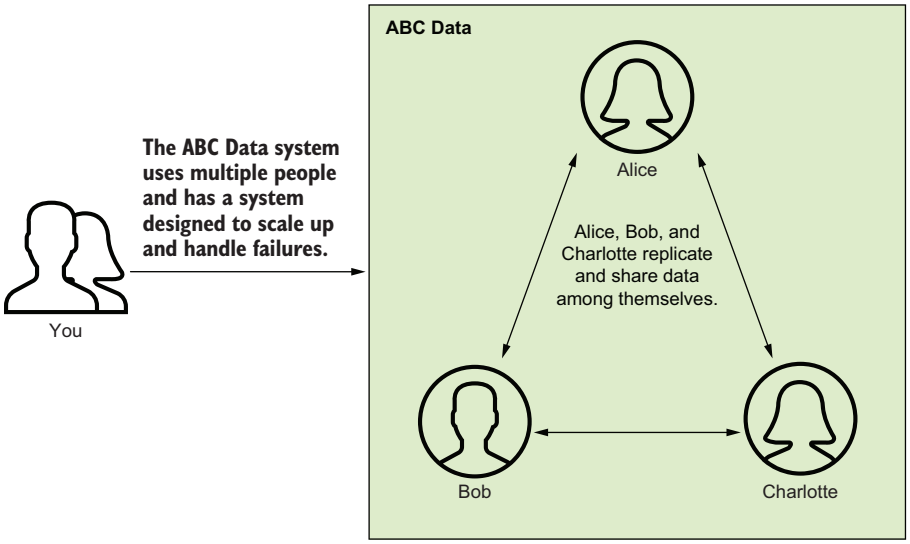


Figure 1.4 ABC Data's approach is designed to meet the scaling challenges that Robert encountered.

This setup is impressive, but because each request can lead to additional requests, you're worried that this system might be overwhelmed even more easily than Robert's. The solution, ABC tells you, is the beauty of its system. ABC's employees take the data set and create multiple copies of it; then they divide this redundant data among them. If the company needs to expand, all it needs to do is add employees to take over some of the existing slices of data. If Diego joins the company, one customer's data might be owned by Alice, Charlotte, and Diego, whereas Bob, Charlotte, and Diego might own other data.

Because ABC Data allows you to choose how many people should respond internally for a successful request, this system gives you control of availability and correctness (figure 1.5). If you always want the most up-to-date data, you can require all three holders of data to respond. If you want to prioritize getting an answer, even if it isn't the most recent one, you can require only one holder to respond. You can balance these properties by requiring two holders to respond; you can tolerate the loss of one, but you can ensure that a majority of them have seen the most up-to-date data, so you should get the most recent information.

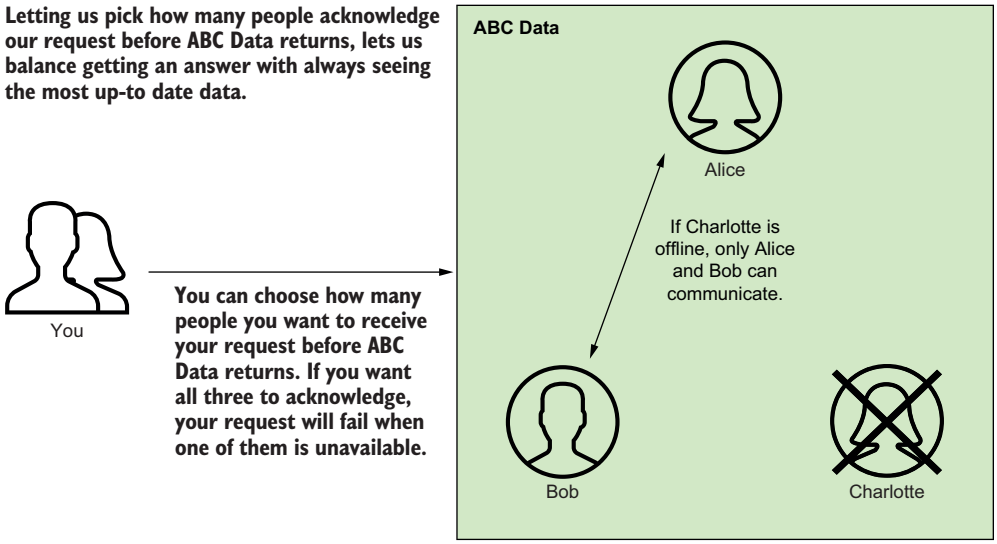


Figure 1.5 ABC Data's approach gives customers control of availability and correctness.

You've learned about two imaginary databases: one that seems to be straightforward but gains complexity as requests grow, and one with a more-complex implementation that attempts to correct the drawbacks of the first system. Before contemplating the awkwardness of telling a friend that you're leaving his business for a competitor, let's snap back to reality and translate these hypothetical databases to the real world.

1.1.2 Real-world databases

Robert's database is a metaphorical relational database, such as PostgreSQL and MySQL. These databases are relatively straightforward to run, fit a multitude of use cases, and offer high performance, and their relational data model has been used for more than 50 years. Often, a relational database is a safe, strong option. Accordingly, developers tend to default toward these systems. But as the preceding examples demonstrated, relational databases also have drawbacks:

- *Availability is often all or nothing.* Even if you run with a read replica, which in Robert's database would be Rosa, you would potentially be able to do reads only if you lost your primary instance.
- *Scalability can be tricky.* A server has a maximum amount of compute resources and memory. When you hit the limit, you're out of room to grow.

Through its approach to these drawbacks, ScyllaDB differentiates itself from relational databases. The ABC Data system is ScyllaDB. Like ABC Data, ScyllaDB is a distributed database that replicates data across its nodes to provide both scalability and fault tolerance. Scaling is straightforward: you add nodes. This elasticity in node count extends to queries. ScyllaDB lets you decide how many replicas are required to respond for a successful query, giving your application room to handle the loss of a server.

1.1.3 Unpacking the definition

ScyllaDB is commonly described as a distributed wide-column NoSQL database—a rewrite of the popular Cassandra database, which (as you might imagine) has similar properties. This definition demonstrates how Scylla differs from other databases: it aims to be more scalable than a relational database and have higher-performance than Cassandra. This positioning is typified by ScyllaDB's description as a NoSQL database. PostgreSQL and MySQL, as their names suggest, are classified as SQL databases. They use SQL (Structured Query Language) to query a relational database schema. NoSQL has become a catch-all term to describe databases that don't conform to this model. A broad array of databases fall into this category, from ScyllaDB to document stores such as MongoDB to "not-only SQL" databases such as CockroachDB.

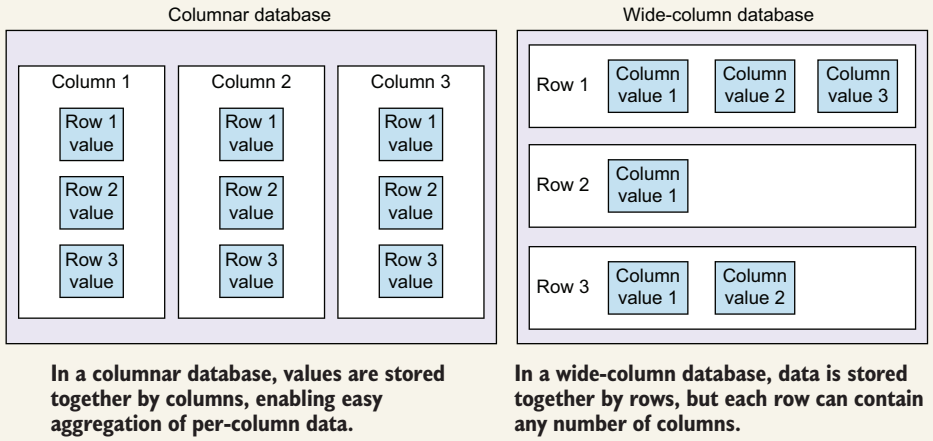
What's a wide-column database?

ScyllaDB and Cassandra are often described as wide-column databases. In this type of database, data can be thought of as a multidimensional map or a key-key value store, in which tables have columns but aren't required to have values for every column. These tables—or *column families*, as they're also called in Cassandra—are stored together on disk. This approach contrasts with that of a columnar database, in which the values for a given column are stored together.

In a columnar database, storing all values for a given column together makes it easy to perform aggregations on all values in a column. The database can easily calculate the median value of a column that stores numbers; because all the values are stored together and co-located, the database doesn't need to read every row to aggregate that data.

Here's how I try to remember the difference: in Scylla and Cassandra, column families can be arbitrarily wide, so they're wide-column stores, as shown in the following figure. All nonprimary key fields are implicitly nullable. By contrast, in a relational database such as Postgres, fields must be explicitly defined as allowing null values.

(continued)



Although they're similarly named, columnar and wide-column databases have large differences in their storage paradigms. ScyllaDB is a wide-column database.

NoSQL databases tend to emphasize scalability and fault tolerance over total correctness and accuracy of data within the database—a property called *consistency* (figure 1.6). This tradeoff may seem to be ridiculous, but you'll get to examine it closely over time. In practice, Scylla is *eventually consistent*, converging toward correctness over time. To achieve its desired scalability and fault tolerance, ScyllaDB runs multiple instances of itself within a cluster.

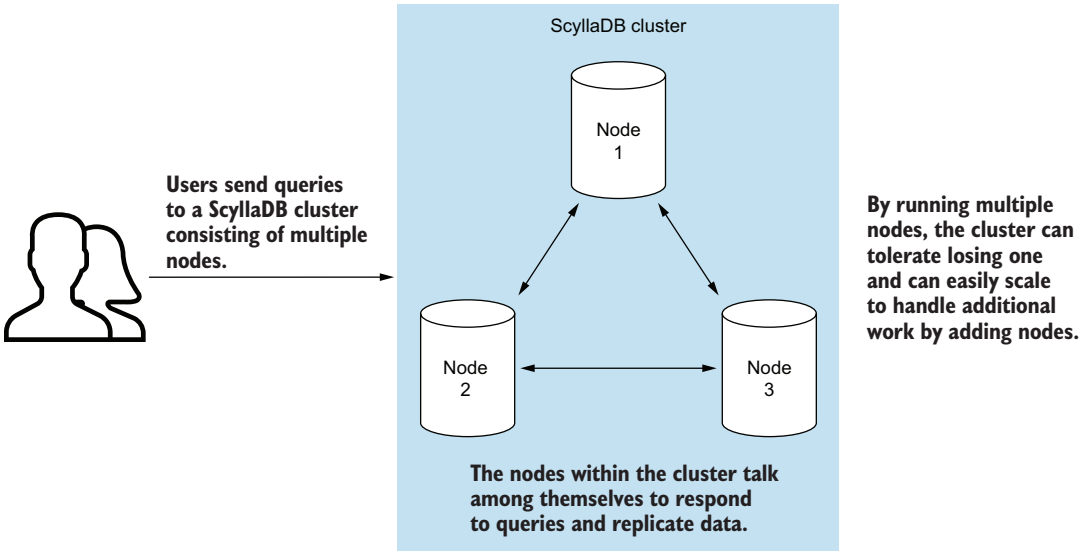


Figure 1.6 ScyllaDB is a distributed database that provides scalability and fault tolerance.

The system has no overarching, all-powerful leader; each node is as important as every other node. There are multiple nodes, and data is distributed across all of them. ScyllaDB isn't a distributed database because distributed systems are cool; it's distributed because it's designed to be a more reliable, scalable database. If you distribute data across all nodes in a cluster, what happens if you lose one node? ScyllaDB stores multiple copies of the data, and because you can choose how many replicas are required to respond to a query, picking any number fewer than the maximum lets the database tolerate node failure. This distribution also helps with scalability. If one node is taking a large amount of traffic, the rest of the cluster won't be affected. Requests that don't hit one heavily-trafficked node won't be affected by any burden on another node. This fault tolerance is critical to ScyllaDB's design. Instead of putting all of your eggs in one basket, you can have many eggs in many baskets. If you lose a basket, you still have lots of eggs!

1.2 ScyllaDB: A distributed database

ScyllaDB runs multiple nodes, making it a distributed system. By spreading its data across its deployment, it achieves its desired availability and consistency, which differentiate it from other systems.

1.2.1 Distributing data

All distributed systems must deliver enough value to overcome introduced complexity. ScyllaDB, which is a distributed system, achieves its scalability and fault tolerance through this design.

When users write data to ScyllaDB, they start by contacting any node. Many systems follow a leader–follower topology; one node is designated as a leader, giving it special responsibilities within the system. If the leader dies, a new leader is elected, and the system continues operating. ScyllaDB doesn't follow this model, however; each node is as special as any other. Without a centralized coordinator deciding which node stores what data, each node must know where any given piece of data should be stored. Internally, Scylla can map a given key to the node that owns it, forwarding requests to that node.

To provide fault tolerance, ScyllaDB not only distributes data, but also replicates it across multiple nodes. The database stores a row in multiple locations; the number depends on the configured replication factor. In a perfect world, each node would acknowledge every request instantly every time, but what happens if it doesn't? To help with unexpected trouble, the database provides tunable consistency.

How you query data depends on the degree of consistency you're looking for. ScyllaDB is an eventually consistent database, and you may see inconsistent data as the system converges toward consistency. Developers must keep eventual consistency in mind when working with the database. To facilitate the various needs of consistency, ScyllaDB provides a variety of consistency levels for queries, including those listed in table 1.1.

Table 1.1 Sample of consistency level options, assuming a three-node cluster

Consistency level	Description	Number required to succeed	Failures tolerated
`ALL`	Requires all nodes to succeed	3	0
`QUORUM`	Requires a majority of replicas to succeed	2	1
`ONE`	Requires a single replica to succeed	1	2

With a consistency level of `ALL`, you can require that all replicas for a key acknowledge a query, but this setting harms availability. You can no longer tolerate the loss of a node. With a consistency level of `ONE`, you require a single replica for a key to acknowledge a query, but this setting greatly increases your chances of getting inconsistent results.

Luckily, some options aren't as extreme. ScyllaDB lets you tune consistency via the concept of quorums. A *quorum* is a majority of members in a group. Legislative bodies, such as the U.S. Senate, don't operate when the number of members present is below the quorum threshold. When applied to ScyllaDB, this concept enables you to achieve intermediate forms of consistency. With a `QUORUM` consistency level, the database requires a majority of replicas for a key to acknowledge a query. If you have three replicas, two of them must accept every read and every write. If you lose one node, you can still rely on the other two to keep serving traffic. Also, you guarantee that a majority of your nodes will get every update, making it much more challenging to read inconsistent data if you use the same consistency level when reading.

When you've picked your consistency level, you know how many replicas you need to execute a successful query. A client sends a request to a node, which serves as the coordinator for that query. Your coordinator node reaches out to the replicas for the given key, including itself if it is a replica. Those replicas return results to the coordinator, and the coordinator evaluates them according to your chosen consistency level. If the coordinator finds that the result satisfies the consistency requirements, it returns the result to the caller.

The CAP theorem classifies distributed systems by saying that they can't provide consistency, availability, and network partition tolerance, as shown in figure 1.7. For the CAP theorem's purposes, *consistency* means that every request reads the most recent write; it's a measure of correctness within the database. *Availability* is whether the system can serve requests, and *network partition tolerance* is the ability to handle a disconnected node.

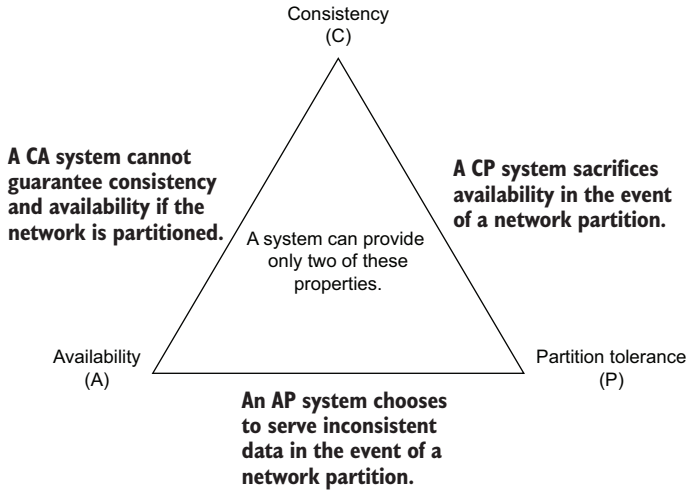


Figure 1.7 The CAP theorem says that a database can provide only two of three properties: consistency, availability, and partition tolerance. ScyllaDB is classified as an AP system.

According to the CAP theorem, a distributed system must have partition tolerance, so it ultimately chooses between consistency and availability. If a system is consistent, reading inconsistent data must be impossible. To achieve consistency, the system must ensure that all nodes receive all necessary copies of data, so it can't tolerate the loss of a node, therefore losing availability.

> **NOTE** In practice, systems aren't as rigidly classified as the CAP theorem suggests. For a more nuanced discussion of these properties, research the PACELC theorem, which illustrates how systems make partial tradeoffs between latency and consistency.

ScyllaDB is classified as an AP system. When it encounters a network partition, it chooses to sacrifice consistency and maintain availability, as you can see in its design. ScyllaDB repeatedly makes choices, via quorums and eventual consistency, to keep the system up and running in exchange for potentially weaker consistency. Emphasizing availability is one of ScyllaDB's key differences from its most popular competitors: relational databases.

1.2.2 ScyllaDB vs. relational databases

I've introduced ScyllaDB by describing its features in comparison with relational databases, but in this section, I'll describe the differences in more detail. Relational databases such as PostgreSQL and MySQL are the standard for data storage in software applications, and they're almost always the default choice for new developers who want to build applications. Relational databases are a strong option for many use cases, but not for every use case.

ScyllaDB is a distributed NoSQL wide-column store. By distributing data across a cluster, ScyllaDB unlocks better availability when nodes go awry than a single-node all-or-nothing relational database can provide. PostgreSQL and MySQL can run in distributed mode, but that mode is powered through extensions or newer storage engines, not by the primary native mode of the database. This distribution is native to ScyllaDB and the bedrock of its design.

By running as a distributed system, ScyllaDB empowers horizontal scalability. Many relational databases are only vertically scalable: you can add resources only by running on a bigger server. With horizontal scalability, you can add nodes to a system to increase its capacity. ScyllaDB supports this expansion. Administrators can add nodes, and the cluster will rebalance itself, offloading data to the new cluster member. In a relational database, horizontal scaling is possible but often manual. Operators need to shard data manually among multiple nodes to achieve this behavior.

ScyllaDB doesn't provide a relational database's atomicity, consistency, isolation, and durability (ACID) guarantees, instead opting for a softer model called *BASE* (basic availability, soft state, and eventual consistency), in which the database has basic availability and is eventually consistent. This decision leads to faster writes than in a relational database, which has to validate the consistency of the database after every write. By contrast, ScyllaDB only has to save the write because it doesn't promise that degree of correctness. The tradeoff, though, is that developers need to consider ScyllaDB's weaker consistency.

ACID vs. BASE

ACID provides a set of guarantees for *transactions*—one or more statements applied to a database. When developers refer to a transaction in a database, they're almost always referring to ACID transactions. ACID provides

- *Atomicity*—All statements in the transaction succeed together or fail together.
- *Consistency*—The database is in a valid state after every transaction.
- *Isolation*—A transaction can't interfere with a concurrently executing transaction.
- *Durability*—Any change in a transaction will be persisted.

I like to think of ACID as being how you expect a database to run. You want a consistent database, and you'd like your writes to be durable. You'd be quite dismayed if you wrote data to the database and it didn't persist.

ScyllaDB provides a softer set of guarantees called BASE. Softer isn't bad, though; these guarantees make it easier for ScyllaDB to provide scalability and fault tolerance. BASE provides

- *Basic availability*—The database is basically available. Some portions of the database may be down, but overall, the system is available.
- *Soft state*—Every node in the database doesn't have to be consistent at every moment.
- *Eventually consistent*—The database will be consistent at some moment.

(continued)

Although I remain convinced that the designer of BASE named the property *soft state* only to make the acronym work, the acronym does describe ScyllaDB's benefits accurately. The database can tolerate the loss of a node and remain available, but to do so, it has to weaken consistency. Nevertheless, it should strive and converge toward consistency.

Ultimately, ScyllaDB versus relational databases is a foundational and philosophical decision. The systems operate so differently and provide such varying guarantees to their clients that picking one over the other has large effects on an application. If you're looking for availability and scalability in your database, ScyllaDB is a strong option.

1.2.3 ScyllaDB vs. Cassandra

ScyllaDB is a rewrite of Apache Cassandra, frequently described as a higher-performance Cassandra or Cassandra in C++. ScyllaDB is designed to be compatible with Cassandra: it uses a compatible API, query language, on-disk storage format, and hash ring architecture. Being like Cassandra but better is ScyllaDB's goal, and the database features some improvements to accomplish this goal.

The choice of language in the rewrite immediately unlocks better performance. Cassandra is written in Java, which uses a garbage collector to perform memory management. Because objects get loaded into memory, at some point they need to be removed. Java's garbage collection algorithms handle this removal, but at the cost of compute. Time spent collecting garbage is time that Cassandra can't spend executing queries. If garbage collection reaches a certain threshold, the Java virtual machine will pause all execution briefly while it cleans up memory—a period referred to as a *stop-the-world pause*. Even if it lasts only milliseconds, that pause can be painful to clients. Although Java exposes many configuration knobs and improves the garbage collector with each release, the pause is a tax that all Java-based applications have to pay, whether in garbage-collection time or time spent mitigating it.

ScyllaDB avoids this tax because it's implemented in C++ and uses manual memory management. Because it has full control of memory allocation and cleanup, ScyllaDB doesn't need to let a garbage collector perform this function on an applicationwide scale. It avoids stop-the-world pauses and can dedicate its compute time to executing queries.

ScyllaDB's biggest architectural difference is its shard-per-core architecture (figure 1.8). Both Cassandra and ScyllaDB shard a data set across its various nodes via placement in a hash ring, which you'll learn more about in chapter 3. ScyllaDB takes this process further by using the Seastar framework (https://seastar.io) to shard data within a node, splitting it per CPU core and giving each shard its own CPU, memory, and network bandwidth allocation.

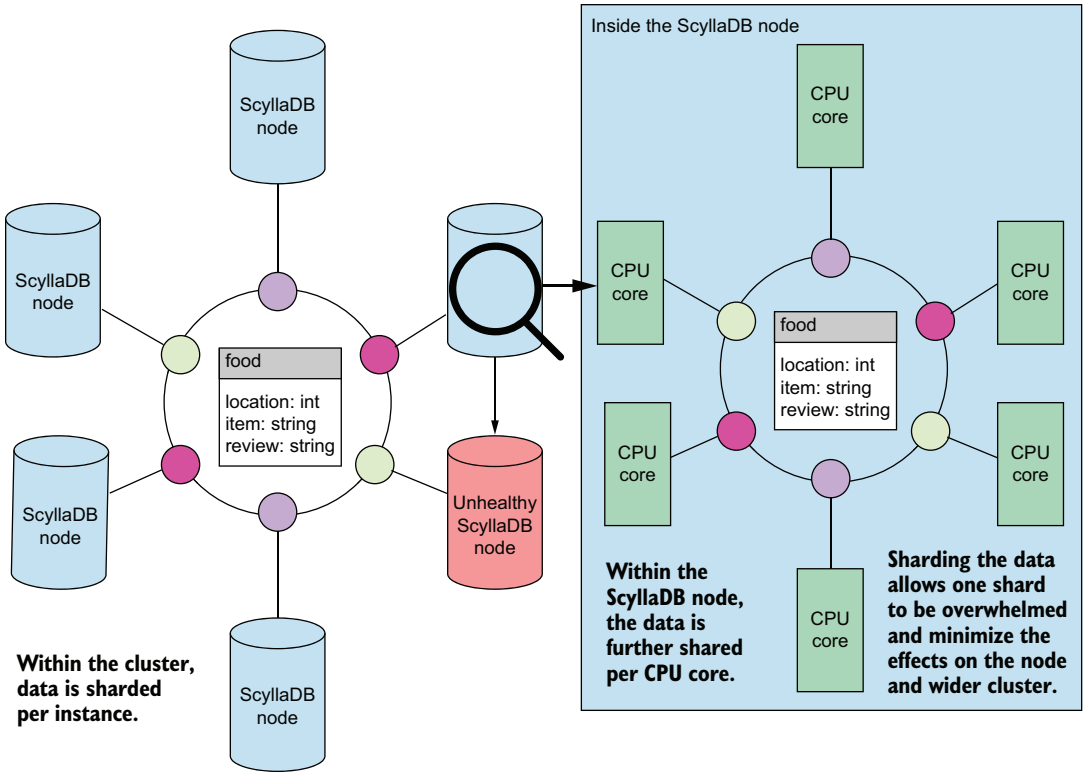


Figure 1.8 ScyllaDB shards data not only within the cluster, but also within each instance.

This sharding further limits the blast radius due to hot traffic patterns; the damage is limited to that shard on that node. Cassandra doesn't follow this paradigm, however; it limits the sharding to that node. If a data partition gets a large amount of requests, it can overwhelm the node, leading to clusterwide struggles.

Performance justifies the rewrite. Both in benchmarks (https://thenewstack.io/benchmarking-apache-cassandra-40-nodes-vs-scylladb-4-nodes) and in the wild (https://discord.com/blog/how-discord-stores-trillions-of-messages), ScyllaDB is faster and more consistent than Cassandra, and it requires fewer servers to operate.

1.2.4 ScyllaDB vs. Amazon and Google systems

This section lumps together a few similar systems: Amazon Aurora, Amazon DynamoDB, Google Cloud Spanner, and Google AlloyDB. These systems can generally be described as scalable cloud-hosted databases; they aim to take a relational data model and provide greater scalability than out-of-the-box PostgreSQL or MySQL. This effort accentuates market demand for scalable databases, showing the value of ScyllaDB.

These systems have two related drawbacks: vendor lock-in and cost. As cloud providers provide these databases, they run in only that specific vendor's cloud environment.

You can't run Google Cloud Spanner in Amazon Web Services, for example. If your application depends heavily on one of these systems, you'll incur a high engineering cost if you decide to switch cloud providers because you'll need to migrate data to a different system with a potentially different storage paradigm. If you're not using that provider (or any provider), those options aren't even on the table for you. Companies that use a cloud provider pay for these services. Operating and maintaining a database is challenging, and although these cloud vendors provide solutions that potentially make the process simpler, they can be quite expensive. Operating a database yourself can also be costly, of course.

ScyllaDB, however, can be run anywhere. Companies run it on-premises or within various cloud providers. ScyllaDB provides a scalable, fault-tolerance database that you can take to any hosting solution.

1.2.5 ScyllaDB vs. document stores

I'm not talking about Google Drive, but about databases that store unstructured documents by a given key, such as MongoDB. These systems support querying these documents, allowing users to access arbitrary document fields without defining a database schema.

ScyllaDB eschews this flexibility to provide (relatively) predictable performance. By requiring users to define their schema up front, it clarifies to both users and the system how data is distributed across the cluster. By forcing users to query data in patterns that match this distribution, ScyllaDB can limit the number of nodes involved in a query, preventing surprisingly expensive queries.

Document stores, on the other hand, tend to be biased toward initial ease of use. In MongoDB, no schema definition is required, but users still need to consider the design of their data to query it effectively. MongoDB runs as a distributed system, but unlike ScyllaDB, it doesn't attempt out of the box to minimize inefficient queries that hit more than the expected number of nodes, leading to potential performance surprises.

In the CAP theorem, MongoDB is a CP (consistent and partition-tolerant) system. Writes require presence of a primary node and are blocked until a new primary node is elected in the event of a network partition. ScyllaDB, however, prioritizes availability, keeping the system running and relying on its tunable consistency.

1.2.6 ScyllaDB versus distributed relational databases

One interesting development for databases over the past few years is the growth of distributed transactional databases. These systems—such as CockroachDB, TiDB, and YugabyteDB—focus on improving the availability of a traditional relational database such as PostgreSQL while offering strong consistency. In the CAP theorem's classifications, they're CP systems, preferring consistency to availability. By emphasizing correctness, they need a quorum of nodes to respond in order to complete a query successfully; if a quorum is lost, the database loses availability. ScyllaDB, however, provides

tunable consistency to dodge this problem. By allowing weaker consistency levels, such as `ONE`, Scylla can handle a greater loss of availability to preserve functionality.

In a relational database, writes are the computationally intensive operation. The database needs to validate its consistency on every write. Scylla, on the other hand, skips this verification, opting for speed and simplicity when writing data. The tradeoff, however, is that reads in Scylla are slower than writes because you need to gather data from multiple nodes that have data stored in different places on disk.

1.2.7 When to prefer other databases

I've described ScyllaDB's benefits relative to those of other databases, but sometimes, I admit, ScyllaDB isn't the best tool for the job. I can't describe it as being unique database because of the Cassandra-rewrite approach, but it does trade operational and design complexity for more graceful failure modes. Choosing Scylla requires you to design applications differently and adds more complexity than something like a cloud-hosted PostgreSQL server. If you don't need ScyllaDB's horizontal scalability and nuanced availability, the increased operational overhead may not be worthwhile. If your application is small, makes few requests, and isn't expected to grow over time, ScyllaDB might be overkill. A database backing comments on your blog probably doesn't need a ScyllaDB cluster unless (like many of us) you want that excuse to try it.

Operating and maintaining a ScyllaDB cluster isn't a hands-off exercise. If you can't dedicate time to operating and maintaining a cluster, that is another signal that a managed offering might be preferable for you. Teams must choose wisely how they spend their time and their money; choosing a less hands-on is a valid decision.

One thing you'll see about Scylla is that with data modeling, your database's design can be inflexible. Adding new query patterns that don't fit with your initial design can be challenging. Although you have ways to work around this problem, other databases may give you more flexibility when you're in the prototyping and learning stages of building features for an application.

Last, some use cases might need a stronger transactional model like ACID. If you're working with financial data, you probably want to use a relational database so that you can have isolation in your operations. One popular example that demonstrates the importance of ACID transactions is concurrent access to bank accounts. Without isolation, you run the risk that concurrent operations will cause a mismatch between how much money the database thinks you have and how much money you actually have. Accountants traditionally prefer accuracy in these areas, so you may prefer a relational database for working with something that needs stronger database transactions. Although scaling a relational database has challenges, taking them on may be preferable to surrendering ACID's guarantees.